dreamtech
PRESS

- An all-inclusive book to teach you everything about Visual Basic 2008
- Easy, Effective, and Reliable

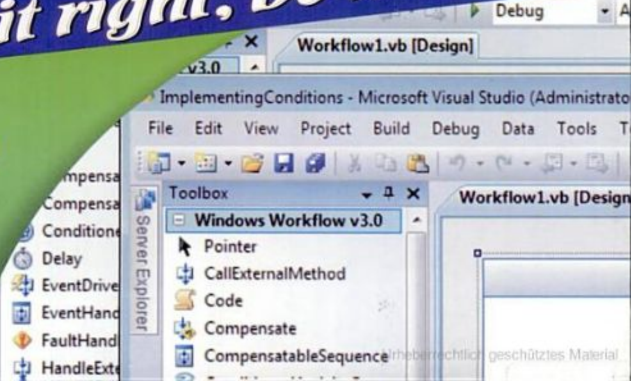
- Quick and Easy learning in Simple Steps
- Most preferred choice worldwide for learning Visual Basic 2008

Visual Basic 2008

IN SIMPLE STEPS

Easy to learn, with hundreds of illustrations

Do it right, Do it fast!



KOGENT
Solutions Inc.

Limits of Liability/disclaimer of Warranty: The author and publisher have used their best efforts in preparing this book. The author make no representation or warranties with respect to the accuracy or completeness of the contents of this book, and specifically disclaim any implied warranties of merchantability or fitness of any particular purpose. There are no warranties which extend beyond the descriptions contained in this paragraph. No warranty may be created or extended by sales representatives or written sales materials. The accuracy and completeness of the information provided herein and the opinions stated herein are not guaranteed or warranted to produce any particulars results, and the advice and strategies contained herein may not be suitable for every individual. Neither Dreamtech Press nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

Trademarks: All brand names and product names used in this book are trademarks, registered trademarks, or trade names of their respective holders. Dreamtech Press is not associated with any product or vendor mentioned in this book.

ISBN: 978-81-7722-918-9

Edition: 2009

Printed at: Printman India, Patparganj, Delhi.

CONTENTS

Chapter 1 ■ Introducing .NET Framework 3.5 1

Introduction 2
Versions of .NET Framework 2
Benefits of .NET Framework 3
Architecture of .NET Framework 3.5 3
Common Language Runtime 4
.NET Framework Class Library 5
Common Type System 5
Common Language Specification 5
Windows Forms 5
ASP.NET and ASP.NET AJAX 6
ADO.NET 6
Windows Presentation Foundation 6
Windows Communication Foundation 6
Windows Workflow Foundation 6
Windows CardSpace 6
LINQ 7
Installing Visual Studio 2008 7
Opening Visual Studio 2008 11
Exploring Visual Studio 2008 IDE 13
Menu Bar 13
Toolbar 13
Design Window 14
Code Editor 14
Server Explorer 15
Solution Explorer 15
Toolbox 16
Properties Window 16
Object Browser 17
Class View Window 18
Summary 18

Chapter 2 ■ Getting Started with Visual Basic 2008 19

Introduction 20
Creating a Visual Basic 2008 Console Application 20
New Features of Visual Basic 2008 21
Query Expressions 22
Local Type Inference 22
Object Initializers 22
Extension Methods 23
Lambda Expressions 23
Anonymous Types 23
Nullable Types 23
Partial Methods 24
Support for XML 24

- Visual Basic 2008 Keywords 24
- Visual Basic 2008 Operators 26
  - Arithmetic Operators 26
  - Assignment Operators 26
  - Concatenation Operators 27
  - Comparison Operators 27
  - Logical Operators 28
- Operator Precedence 30
- Data Types 31
  - Data Type Conversion 32
- Variables 32
- Constants 34
- Selection Statements 34
  - If Else Statement 35
  - Select Case Statement 36
- Iteration Statements 37
  - While Statement 37
  - Do While Statement 38
  - For Statement 39
  - For Each Statement 40
- Arrays 41
- Summary 41

- Working with Classes and Objects 44
  - Access Modifiers 45
  - Methods in Visual Basic 45
  - Constructors and Destructors 46
  - Partial Classes 47
  - Shared Methods 48
  - Extension Methods 49
- Encapsulation 52
- Inheritance 54
  - Defining a Derived Class 55
  - Accessing Members of the Base Class 56
  - Working with Abstract Classes 57
  - Working with Sealed Classes 58
- Implementing Polymorphism 59
  - Implementing Compile Time Polymorphism 59
  - Runtime Polymorphism 62
- Structures 63
  - Using a Structure 64
- Working with Properties 65
  - Using a Property 65

- Using an Anonymous Type for Read-Only Properties 66

Interfaces 67

- Defining an Interface 67
- Implementing an Interface 67
- Inheriting an Interface 68

Namespaces 69

- Creating Namespaces 70
- Referencing Namespaces 70

Summary 72

Chapter 4 ■ Working with Windows Forms 73

Introduction 74

Creating a Visual Basic 2008 Windows Forms Application 74

Performing Some Basic Operations on Windows Forms 75

- Setting the Title of a Form 75
- Adding Controls to a Form 76
- Handling the Click Event of a Button 79
- Docking and Anchoring Controls 79
- Setting the Tab Order of Controls 81
- Enabling and Disabling Controls 84

Working with Multiple Forms 86

- Setting the Startup Form 88

Creating Message Boxes 91

Creating Input Boxes 94

Creating Dialog Boxes 96

Summary 99

Chapter 5 ■ Working with Windows Controls 101

Introduction 102

The Label Control 102

- Formatting the Text in Labels 102
- Handling the Click Event of Label 104

The TextBox Control 104

The Button Control 107

- Formatting the Text in Buttons 107
- Setting the Background and Foreground Colors of Buttons 108

The RadioButton Control 111

The CheckBox Control 114

The ComboBox Control 116

The ListBox Control 120

The GroupBox Control 122

The Panel Control 124

The PictureBox Control 126

The ProgressBar and Timer Controls 128

Summary 130

Chapter 6 ■ Introducing Windows Presentation Foundation 131

Introduction 132
The Architecture of WPF 3.5 132
  The PresentationFramework Component 133
  The PresentationCore Component 133
  The WindowsBase Component 133
  The MIL or Milcore Component 133
Types of WPF Applications 133
  Standalone WPF Applications 133
  XAML Browser Applications 135
The WPF 3.5 Designer 137
  The Design View 137
  The XAML View 140
  The Split View Bar 141
  The Tag Navigator 142
XAML and WPF 143
  XAML Elements and Attributes 143
  Namespaces and XAML 145
  Markup Extensions 145
Common Controls in WPF 3.5 146
  Using the Grid Control 147
  Using the Button Control 148
  Using the TextBox Control 149
  Using the PasswordBox Control 151
  Using the TextBlock Control 153
  Using the Border Control 154
  Using the GridSplitter Control 155
  Using the Canvas Control 157
  Using the StackPanel Control 158
Resources and Styles 159
  Using a Static Resource 160
  Using a Dynamic Resource 161
  Setting Style Through a Resource 163
Summary 164

Chapter 7 ■ Windows Workflow Foundation 165

Introduction 166
Workflow Principles 166
  Workflows Coordinate Work Performed by People and Software 166
  Workflows are Long Running and Stateful 166
  Workflows are based on Extensible Models 166
  Workflows are Transparent and Dynamic throughout their Lifecycle 166
Components of Windows Workflow Foundation 167
  Workflow 167
  Base Activity Library 168
  Custom Activities 169
  Host Process 170

Runtime Engine 170

Runtime Services 170

Developing a Simple Workflow Application 170

Implementing Conditions in Workflows 172

Using Workflows with Other Applications 175

Summary 179

Chapter 8 ■ Working with Data and ADO.NET 181

Introduction 182

Introducing ADO.NET 182

New Features in ADO.NET 182

Components of ADO.NET 183

Basic Operations in ADO.NET 185

Types of Data Binding in Windows Forms 198

Simple Data Binding 198

Complex Data Binding 201

Data Binding in Windows Presentation Foundation 203

Data Flow Directions 203

Declaration of Data Binding in WPF 204

Binding Sources in WPF 209

Summary 215

Chapter 9 ■ Introducing Language-Integrated Query 217

Introduction 218

LINQ Queries 218

Executing a Simple LINQ Query 218

The Standard Query Operators 220

The Sorting Operators 220

The Set Operators 222

Filtering Operator 223

The Quantifier Operators 224

The Projection Operators 225

The Partitioning Operators 226

The Join Operators 228

The Grouping Operators 230

The Generation Operators 231

The Element Operators 231

The Conversion Operators 232

The Aggregate Operators 232

LINQ to ADO.NET 232

LINQ to SQL 232

LINQ to DataSet 236

Anonymous Types 237

Lambda Expressions 237

Summary 238

Chapter 10 ■ Deploying Visual Basic 2008 Applications 239
Introduction ... 240
Deploying Applications by Using Windows Installer ... 240
Deploying Applications by Using ClickOnce ... 252
Summary ... 256

Chapter 11 ■ Web Service and WCF Service 257
Introduction ... 258
Using a Web Service ... 258
Using a WCF Service ... 265
Summary ... 270

Introduction

The .NET Framework is one of the most widely used software development environment in today's programming world. Before its introduction, programmers had to face a lot of difficulties to integrate the code written using different programming languages. This was due to the reason that each language used a different execution environment to execute the code written using that language. For example, code written using Visual Basic 6.0 requires a different execution environment for execution than that is required by code written using Visual C++. With the .NET Framework, Microsoft has provided programmers a single platform for developing applications using different programming languages, such as Visual Basic, Visual C#, and Visual C++.

The .NET Framework 3.5 is shipped with the Microsoft Visual Studio 2008. Microsoft Visual Studio is a set of development tools designed to help software developers to develop complex applications more quickly and easily. It provides the necessary environment in which developers can create and execute various types of applications, including Console applications, Windows Forms applications, WPF applications, Web applications, and Web services. It has improved the process of development and made it easier.

In this chapter, we learn about versions of .NET Framework, benefits of .NET Framework, and architecture of .NET Framework. We also learn how we can install Visual Studio 2008 and how we can open it. Finally, we take a look at Visual Studio 2008 IDE.

Let's first start by taking an overview of the different versions of .NET Framework.

Versions of .NET Framework

The .NET Framework has seen many upgrades since the release of its first version in 2002. All the versions of the .NET Framework that have been released till now are described as follows:

- **.NET Framework 1.0:** The .NET Framework 1.0 is the first version of the .NET Framework and was released by Microsoft on February 13, 2002. It is available for download in the form of a redistributable package as well as a Software Development Kit (SDK). It is also a part of Visual Studio .NET 2002, which is the first version of Visual Studio .NET.
- **.NET Framework 1.1:** The first major upgrade of the .NET Framework, the .NET Framework 1.1, was released on April 3, 2003. It is available for download in the form of a redistributable package as well as a Software Development Kit (SDK). It is also a part of Visual Studio .NET 2003, which is the second version of Visual Studio .NET. In contrast to the .NET Framework 1.0, the .NET Framework 1.1 has in-built support for mobile ASP.NET controls and Open Database Connectivity (ODBC) and Oracle databases. It also has support for Internet Protocol version 6 (IPv6).
- **.NET Framework 2.0:** The second major upgrade of the .NET Framework, the .NET Framework 2.0, was released on January 22, 2006. It is available for download in the form of a redistributable package as well as a Software Development Kit (SDK). It is also a part of Visual Studio 2005 and Microsoft SQL Server 2005. The .NET Framework 2.0 is the last version of the .NET Framework that has support of Windows 2000. The .NET Framework 2.0 has many changes and enhancements as compared to the .NET Framework 1.1. It has a number of Application Programming Interface (API) changes. It contains many new ASP.NET Web controls and data controls. It also contains new personalization features for ASP.NET, for example support for themes, skins, and WebParts.
- **.NET Framework 3.0:** The third major upgrade of the .NET Framework, the .NET Framework 3.0, was released on November 21, 2006. It contains a set of managed code APIs, which are an integral part of Windows Vista and Windows Server 2008. Managed code is the code that runs under Common Language Runtime (CLR). We discuss CLR in detail later in this chapter. The .NET Framework 3.0 uses the same version of CLR that was incorporated with .NET Framework 2.0. The .NET Framework 3.0 includes the following four new components:
  - Windows Presentation Foundation (WPF)
  - Windows Communication Foundation (WCF)

  - Windows Workflow Foundation (WF)
  - Windows CardSpace (WCS)
- **.NET Framework 3.5:** The fourth major upgrade of the .NET Framework, the .NET Framework 3.5, was released on November 19, 2007. Similar to the .NET Framework 3.0, the .NET Framework 3.5 also uses the same version of CLR. The .NET Framework 3.5 also installs the .NET Framework 2.0 SP1 and the .NET Framework 3.0 SP1, which includes methods and properties that are required for the .NET Framework 3.5 features, such as Language Integrated Query (LINQ). In addition to LINQ, the .NET Framework 3.5 includes many other new features, such as extension methods, lambda expressions, anonymous types, and built-in support for ASP.NET AJAX.

After having a quick overview of the versions of the .NET Framework, let's move on to discuss the benefits of the .NET Framework.

Benefits of .NET Framework

The .NET Framework offers many benefits to the programmers in developing applications. Some of these benefits are as follows:

- **Consistent programming model:** The .NET Framework provides a consistent object-oriented programming model across different languages. You can use this model to create programs for performing different tasks, such as connecting to and retrieving data from databases, and reading from and writing to files.
- **Language interoperability:** Language interoperability is a feature that enables code written in different languages to interact with each other. This allows reusability of code and improves the efficiency of the development process. For example, you can inherit a class created in C# in Visual Basic and vice-versa. The CLR has built-in support for language interoperability. However, there is no assurance that the code written using one programming language will work properly in programs developed using another programming language. Therefore, to ensure multi-language code interoperability, a set of language features and rules, called Common Language Specification (CLS), is defined. The components that follow these rules and expose only CLS features are said to be CLS-compliant.
- **Automatic management of resources:** When you create a .NET application, you do not need to manually free application resources, such as files, memory, network and database connections. The CLR automatically tracks the resource usage and saves you from the task of manual resource management.
- **Ease of deployment:** The .NET Framework makes the task of deployment easier. In most cases, to install an application, you need to copy the application along with its components, on the target computer. The .NET Framework provides easy deployment of applications by installing new applications or components that do not have an adverse effect on the existing applications. In .NET, applications are deployed in the form of assemblies; therefore, registry entries are not required to store information about components and applications. In addition, problems that used to arise due to different versions of an assembly are also overcome or eliminated in .NET Framework since assemblies also store information about different versions of the components used by an application.

Architecture of .NET Framework 3.5

The .NET Framework 2.0 and the .NET Framework 3.0, along with their service packs, form the foundation of the .NET Framework 3.5. In other words, the architecture of the .NET Framework 3.5, besides its new features and enhancements, includes components of the .NET Framework 2.0 and the .NET Framework 3.0. Architecture of the .NET Framework 3.5 is shown in Fig.VB-1.1:

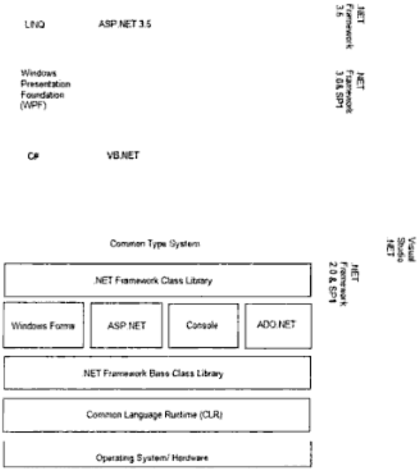


Fig.VB-1.1

As shown in Fig.VB-1.1, the main components of the .NET Framework 2.0 are CLR, .NET Framework Base Class Library, Windows Forms, ASP.NET, Common Type System (CTS), CLS, and .NET languages, such as C# and Visual Basic. The .NET Framework 3.0 adds four major components — Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF), Windows Workflow Foundation (WF), and Windows CardSpace—to the .NET Framework 2.0. Similarly, the .NET Framework 3.5 adds few more components and features, including LINQ, ASP.NET 3.5, and ActiveX Data Objects .NET (ADO.NET) Entity Framework and Data services, to the .NET Framework 3.0.

Let's now discuss the major components of the .NET Framework 3.5, one by one.

Common Language Runtime

One of the most important components of the .NET Framework is CLR, better known as the runtime. It provides functionalities, such as memory management, exception handling, debugging, security, thread execution, code execution, code safety, code verification, compilation. The CLR can host a variety of languages and provides common tools to these languages; thereby, ensuring interoperability between code written in different languages. The managed environment of the runtime eliminates many common software issues. For example, the runtime automatically releases the objects when they are no longer in use. This automatic memory management resolves the issue of memory leaks and invalid memory references.

CLR is the module that actually runs your .NET applications. When you run a .NET application, the language compiler compiles the source code into an intermediate code, called Microsoft Intermediate Language (MSIL) code. The MSIL code is similar to Java's bytecode. The MSIL code is later converted by the Just-In-Time (JIT) compiler into native machine code, which is the final executable code. Fig.VB-1.2 explains the functioning of CLR:

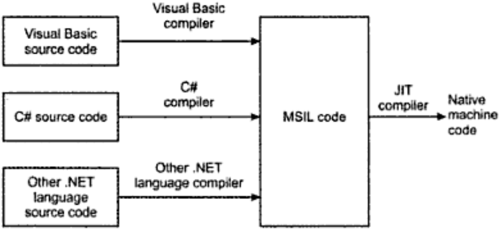


Fig.VB-1.2

.NET Framework Class Library

The .NET Framework Class Library is a huge library made up of a hierarchy of namespaces. Each namespace in the .NET Framework Class Library is a collection of different .NET types, such as classes, structures, interfaces, enumerations, and delegates. The namespaces are logically defined by their functionality. For example, the **System.Data** namespace contains all the functionalities available for accessing databases. This namespace is further broken down into other namespaces, such as **System.Data.SqlClient**, which exposes functionality required to work with Structured Query Language (SQL) Server databases and **System.Data.OleDb**, which exposes functionality required to work with Object Linking and Embedding Database (OLE DB) databases. Grouping types in namespaces also solves the problem of name collisions as we can have members (types) with the same name in more than one namespace.

The System namespace contains the most basic classes, structures, interfaces, delegates, and enumerations. Some important classes of the System namespace are **Console**, **Math**, **Object**, **String**, **Array**, **Enum**, and **Delegate**. Some important structures of the System namespace are **Boolean**, **Byte**, **Char**, **Decimal**, **Single**, **Double**, and **Int32**.

Common Type System

Common Type System (CTS) is the component of CLR through which the .NET Framework provides support for multiple languages. The CTS specification describes all possible data types and programming constructs supported by the runtime and specifies how these entities can interact with each other; therefore, calling one language from another does not require type conversions. CTS provides a base set of data types for all the languages supported by .NET Framework; however, each language uses aliases for the base data types provided by CTS. For example, CTS uses the data type **System.Int32** to represent a 4 byte integer value; however, Visual Basic uses the alias **Integer** for the same. This is done for the sake of clarity and simplicity.

Common Language Specification

The Common Language Specification (CLS), a subset of CTS, defines the common types and programming constructs that are supported by all .NET programming languages. CLS enables interoperability on the .NET platform; therefore, languages supporting the CLS can easily use each other's class libraries. Application Programming Interfaces (APIs) that are designed by following the rules defined in CLS can be used by all .NET-compliant languages.

Windows Forms

A Windows form is similar to a blank slate on which we can add controls to perform various functions. Windows forms are used to develop Windows Forms applications. The .NET Framework provides you the facility to develop Windows Forms applications using a .NET-compliant language. A Windows form can be used to accept input from a user or display information to the user. You can add controls to a Windows form and develop responses to the user actions, such as mouse clicks or key presses.

ASP.NET and ASP.NET AJAX

ASP.NET is a Web development model, which is used to develop interactive, data-driven Web applications over the Internet. ASP.NET Web applications can be created using any CLR-compliant language, such as Visual Basic, Visual C#, and Visual C++.

AJAX, formerly code-named as Atlas, is an extension of ASP.NET for developing and implementing AJAX functionality. ASP.NET AJAX includes both client-side and server-side components that allows developers to create Web applications, which does not require complete reload of the page while making any modifications to the page. It enables you to send only parts of a Web page to the Web server by allowing you to make asynchronous calls to the Web server. This decreases network traffic as well as processing on the Web server.

ADO.NET

ActiveX Data Objects .NET (ADO.NET) is a technology for working with data and databases of all types. It provides access to various data sources, such as Microsoft SQL Server, and data sources exposed through OLE DB and eXtensible Markup Language (XML). You can use ADO.NET to connect to data sources for retrieving, manipulating, and updating data. The most important feature of ADO.NET is disconnected data architecture. In this architecture, applications are connected to the databases only till data is retrieved or modified.

Windows Presentation Foundation

Apart from Windows Forms, Windows client applications can also be developed through WPF (formerly codenamed as Avalon). WPF also facilitates building various kinds of interfaces, such as documents, media, two and three-dimensional graphics, animations. It helps in creating Windows client applications of superior quality. You can use WPF for creating both standalone and browser-hosted applications. WPF introduces a new language called eXtensible Application Markup Language (XAML), which is a language based on XML.

Windows Communication Foundation

Windows Communication Foundation (WCF) (formerly codenamed as Indigo) is a service-oriented technology introduced by Microsoft for building and running connected systems. The service-oriented design results in a distributed system that runs between the services and clients. You can understand WCF more easily if you are familiar with concepts, such as Web services, remoting, distributed transactions, and message queuing.

WCF based applications are interoperable with any process as these communicate through Simple Object Access Protocol (SOAP) messages. When a WCF process connects with a non-WCF process, it uses XML-based encoding for SOAP messages, but when it connects with another WCF process, the SOAP messages are encoded into a binary format.

Windows Workflow Foundation

Windows Workflow Foundation (WF) is a technology introduced by Microsoft that provides a programming model for building workflow based applications on Windows. The components of WF include activities, workflow runtime, workflow designer, and a rules engine. WF is a part of .NET Framework 3.0 and 3.5.

The most important feature of WF is the separation between the business process code and the actual implementation code. Before WF was introduced, both the business logic and the actual implementation code were written together while developing applications.

Windows CardSpace

Windows CardSpace (WCS) is a client software provided by Microsoft that makes the process of securing resources easier and also makes sharing personal information on the Internet more secure. It helps programmers to develop Web sites and software that are less prone to identity related attacks such as phishing. WCS solves the problems of traditional online security mechanisms by reducing dependence on user names and passwords. It, instead, uses a separate desktop and cryptographically strong authentication to ensure secure online transactions.

LINQ

Language Integrated Query (LINQ) is a component of the .NET Framework 3.5 that adds native data querying capabilities to .NET languages using the syntax similar to SQL. This implies that with LINQ, you can write statements similar to SQL statements, in a .NET language, such as Visual Basic.

Though LINQ queries resemble SQL, they are not restricted to accessing only relational databases. LINQ enabled data access components are as follows:

- **LINQ to ADO.NET:** Includes two options, LINQ to SQL, which translates a query into a SQL query, and then issues it against tables in a SQL Server database, and LINQ to DataSet, which executes a query on the contents of a DataSet.
- **LINQ to Object:** Allows querying objects in a collection. LINQ to Objects is not dynamic. Once you create a result set and use it, any changes made to the source collection do not automatically update the result set.
- **LINQ to XML:** Allows querying of XML data. In addition, it also helps in creating and manipulating XML data. This option has a different syntax, but the basic organization of the LINQ query remains the same.

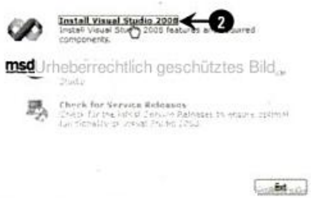
Here, we discussed the architecture of .NET framework 3.5. In the next section, we learn how to install Visual Studio 2008.

Installing Visual Studio 2008

In order to develop a .NET application, you need to have the required software installed on your computer, such as Visual Studio 2008 or Visual Web Developer. Visual Studio 2008 is a software development product that enables programmers to develop various types of applications including Console applications, Windows applications, Web applications, and Web services. Before installing Visual Studio 2008, you need to install the hardware and software components given in Table 1.1 on your computer:

After installing all these components on your computer, perform the following steps to install Visual Studio 2008:

1. *Insert* the DVD-ROM of Visual Studio 2008 in the DVD-ROM drive. The **Visual Studio 2008 Setup** wizard begins, as shown in Fig.VB-1.3:



Install Visual Studio 2008
Install Visual Studio 2008 features and required components.
2
Check for Service Releases
Exit

Fig. 1.3

2. *Click* the **Install Visual Studio 2008** link, as shown in Fig.VB-1.3. The setup loads the installation components, as shown in Fig.VB-1.4:

Fig.VB-1.4

After the loading process is completed, the **Next** button becomes enabled, as you can see in Fig.VB-1.5:

Fig.VB-1.5

3. *Click* the **Next** button, as shown in Fig.VB-1.5. The **Microsoft Visual Studio 2008 Setup – Start Page** page appears, as shown in Fig.VB-1.6:

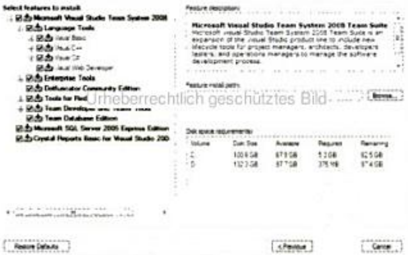
Fig.VB-1.6

4. *Select* the **I have read and accept the license terms** radio button and then *click* the **Next** button to continue, as shown in Fig.VB-1.6. The **Microsoft Visual Studio 2008 Setup – Options Page** page appears, as shown in Fig.VB-1.7:

Fig.VB-1.7

In the left side of the **Microsoft Visual Studio 2008 Setup – Options Page** page, you are presented with three radio buttons—**Default**, **Full**, and **Custom**—that permit you to choose the features of Visual Studio 2008 to install.

5. *Select* a radio button, as shown in Fig.VB-1.7. In this case, we have selected the **Custom** radio button as we are going to install a customized version of Visual Studio 2008.
6. *Click* the **Next** button, as shown in Fig.VB-1.7. The next page appears where you can select the features of Visual Studio 2008 that you want to install, from the list displayed on the left side of the page, as shown in Fig.VB-1.8:



Select features to install:

- ☑ Microsoft Visual Studio Team System 2008
  - ☑ Language Tools
    - ☑ Visual Basic
    - ☑ Visual C++
    - ☑ Visual C#
    - ☑ Visual Web Developer
  - ☑ Enterprise Tools
  - ☑ Dotfuscator Community Edition
  - ☑ Tools for Red[illegible]
  - ☑ Team Developer [illegible]
  - ☑ Team Database Edition
- ☑ Microsoft SQL Server 2005 Express Edition
- ☑ Crystal Reports Basic for Visual Studio 200

Feature description:

Microsoft Visual Studio Team System 2008 Team Suite
Microsoft Visual Studio Team System 2008 Team Suite is an expansion of the Visual Studio product line to include new lifecycle tools for project managers, architects, developers, testers, and operations managers to manage the software development process.

Feature install path:

Disk space requirements:

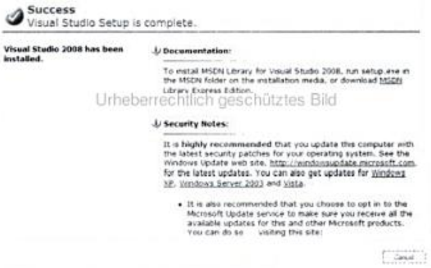
Volume	Disk Size	Available	Required	Remaining
C:	100.9 GB	87.9 GB	5.3 GB	82.5 GB
D:	132.2 GB	87.7 GB	375 MB	87.4 GB

Fig.VB-1.8

7. *Select* the features of Visual Studio 2008 and *click* the **Install** button, as shown in Fig.VB-1.8. The installation of Visual Studio 2008 starts, as shown in Fig.VB-1.9:

Fig.VB-1.9

After a few minutes, the final setup page, **Microsoft Visual Studio 2008 Setup – Finish Page**, appears indicating that the Visual Studio 2008 setup has been completed successfully, as shown in Fig.VB-1.10:



Success
Visual Studio Setup is complete.

Visual Studio 2008 has been installed.

Documentation:

To install MSDN Library for Visual Studio 2008, run setup.exe in the MSDN folder on the installation media, or download MSDN Library Express Edition.

Security Notes:

It is **highly recommended** that you update this computer with the latest security patches for your operating system. See the Windows Update web site, http://windowsupdate.microsoft.com, for the latest updates. You can also get updates for Windows XP, Windows Server 2003 and Vista.

- It is also recommended that you choose to opt in to the Microsoft Update service to make sure you receive all the available updates for this and other Microsoft products. You can do so visiting this site:

Fig.VB-1.10

8. *Click* the **Finish** button to end the **Visual Studio 2008 Setup** wizard, as shown in Fig.VB-1.10.

After learning how to install Visual Studio 2008, let's now learn how we can open (or start) Visual Studio 2008.

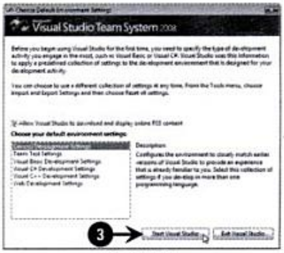
Opening Visual Studio 2008

After you have installed Visual Studio 2008 on your computer, you can open it by performing the following steps:

1. *Click* the **Start** button on the task bar, as shown in Fig.VB-1.11:

Fig.VB-1.11

2. Then *click* **All Programs→Microsoft Visual Studio 2008→Microsoft Visual Studio 2008**, as shown in Fig.VB-1.11. The **Choose Default Environment Settings** dialog box appears, as shown in Fig.VB-1.12:



Choose Default Environment Settings

Visual Studio Team System 2008

Before you begin using Visual Studio for the first time, you need to specify the type of development activity you engage in the most, such as Visual Basic or Visual C#. Visual Studio uses this information to apply a predefined collection of settings to the development environment that is designed for your development activity.

You can choose to use a different collection of settings at any time. From the Tools menu, choose Import and Export Settings and then choose Reset all settings.

☑ Allow Visual Studio to download and display online RSS content

Choose your default environment settings:

General Development Settings
Team Test Settings
Visual Basic Development Settings
Visual C# Development Settings
Visual C++ Development Settings
Web Development Settings

Description:
Configures the environment to closely match earlier versions of Visual Studio to provide an experience that is already familiar to you. Select this collection of settings if you develop in more than one programming language.

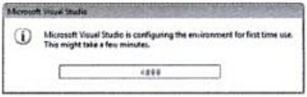
Fig.VB-1.12

Note

The Choose Default Environment Settings dialog box is displayed only when you open Visual Studio 2008 for the first time.

The **Choose Default Environment Settings** dialog box allows you to select the default environment settings for Visual Studio 2008 installed on your computer.

3. *Select* one of the available options and *click* the **Start Visual Studio** button, as shown in Fig.VB-1.12. In this case, we have selected **General Development Settings**. This configures the environment for Visual Studio 2008, as shown in Fig.VB-1.13:



Microsoft Visual Studio

Microsoft Visual Studio is configuring the environment for first time use. This might take a few minutes.

Fig.VB-1.13

After a few minutes, the Start Page of Visual Studio 2008 appears, as shown in Fig.VB-1.14:

Fig.VB-1.14

In this section, we have learned how we can open Visual Studio 2008. Let's now explore Visual Studio 2008 IDE.

Exploring Visual Studio 2008 IDE

The Visual Studio 2008 Integrated Development Environment (IDE) provides development and execution environment for various kinds of applications, such as console applications, Windows applications, and Web applications. It contains a number of menu bars, toolbars, and windows that help you throughout the development of an application. Fig.VB-1.15 shows the Visual Studio 2008 IDE:

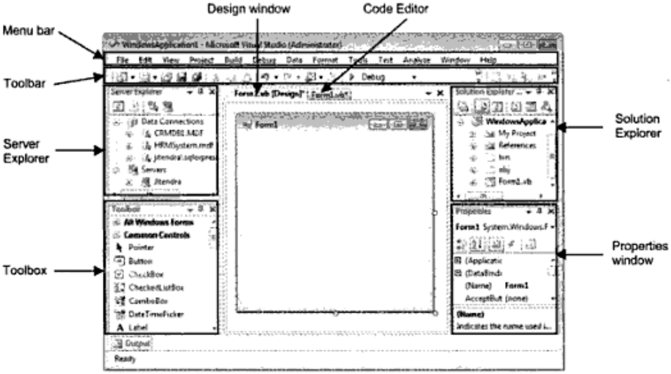


Fig.VB-1.15

Let's now discuss some important components of Visual Studio 2008 IDE, one by one.

Menu Bar

Menu bar is a collection of menus, each of which contains a set of options for performing various tasks. These menus include File menu, Edit menu, Build menu, Debug menu, and so on. Each menu in the menu bar contains options for performing a specific category of tasks. For example, the File menu contains options for performing the file management tasks, such as creating a new project, opening an existing project, saving a project, and closing an opened project. Fig.VB-1.16 shows the different menus present in the menu bar:

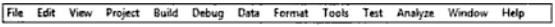


Fig.VB-1.16

Toolbar

Toolbar works as a container for the commands used to perform various tasks while developing applications in Visual Studio 2008. Many of these commands are the shortcuts to the options present in the various menus of the menu bar. The commands present in the Toolbar include commands for creating a new project, opening an existing project, and saving a project. Fig.VB-1.17 shows the various commands available in the Toolbar:

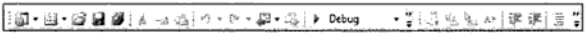


Fig.VB-1.17

Design Window

Design window is the place where we design the user interface for our application. It occupies the middle portion of the Visual Studio 2008 IDE. The user interface for a form can be designed by adding and organizing controls on the form, in the Design window. You can open a form in the Design window by double-clicking it in the Solution Explorer. Fig.VB-1.18 shows the Design window of `Form1`:

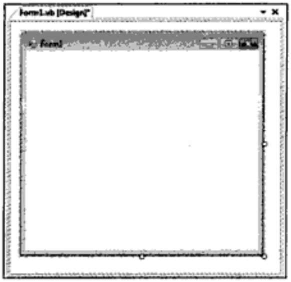


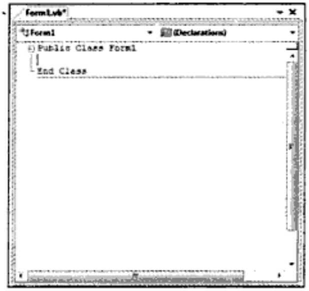
Fig.VB-1.18

Code Editor

Code Editor is the place where we can add the code for handling a form and the various controls added to it. Generally, we write the code in the Code Editor in the form of event handlers, which are methods that tell the computer how to respond when an event occurs. For example, we can open a message box displaying a message, when the user clicks a button added to a form. You can open the Code Editor using any of the following ways:

- ❑ By right-clicking a form in the Solution Explorer and selecting the View Code option from the context menu
- ❑ By double-clicking the form or any control added to the form in the Design window
- ❑ By selecting the form in the Solution Explorer and pressing the F7 key on the keyboard

Fig.VB-1.19 shows the default code of `Form1` in the Code Editor:



```
Public Class Form1

End Class
```

Fig.VB-1.19

Server Explorer

Server Explorer is a window that allows your application to communicate with a database server. Using the Server Explorer, you can create a data connection for connecting to a database server. The database file you specify while creating a database connection gets added to the Server Explorer along with all the tables it contains. You can use the data contained in these tables in your application and also make changes in the table data using the Server Explorer. You can open the Server Explorer by either clicking View→Server Explorer on the menu bar or pressing the **CTRL+ALT+S** key combination on the keyboard. Fig.VB-1.20 shows the Server Explorer with a data connection added to it:

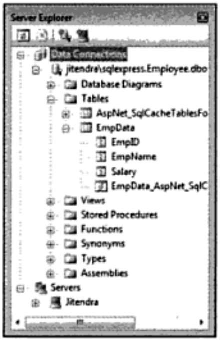


Fig.VB-1.20

Solution Explorer

Solution Explorer enables you to view all the files related to an application. You can open the Solution Explorer by either clicking View→Solution Explorer on the menu bar or pressing the **CTRL+ALT+L** key combination on the keyboard. Fig.VB-1.21 shows the different files of an application in the Solution Explorer:

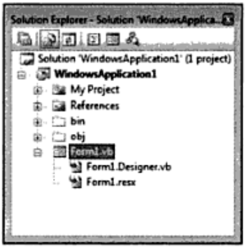

Solution Explorer - Solution 'WindowsApplica...
Solution 'WindowsApplication1' (1 project)
WindowsApplication1
My Project
References
bin
obj
Form1.vb
Form1.Designer.vb
Form1.resx

Fig.VB-1.21

The Properties button (first button from the left) in the toolbar of the Solution Explorer (Fig.VB-1.21) can be clicked to display the properties of the currently selected item in the Solution Explorer. By default, Solution Explorer do not show all the files included in a solution. You can make all of them displayed in the Solution Explorer by clicking the Show All Files button (second button from the left) in the toolbar of the Solution Explorer. You can also use the View Code and View Designer buttons present in the toolbar of the Solution Explorer to switch between the Design window and Code Editor of a form.

Toolbox

Toolbox is a window that provides you a set of controls for designing the user interface for a form of an application. It appears in the left side of the Design window on the Visual Studio 2008 IDE. The controls in the Toolbox are grouped under different tabs, such as Common Controls tab, Menus & Toolbars tab, and Data tab. Each tab stores controls related to a specific category, for example, the Data tab stores the controls, which are used in the applications that require interaction with databases. You can open the Toolbox by clicking View→Toolbox on the menu bar or by pressing the **CTRL+ALT+X** key combination on the keyboard. Fig.VB-1.22 shows the various tabs of the Toolbox:

Fig.VB-1.22

Properties Window

Properties window enables you to set properties and events of a form and its controls at the design time. You can open the Properties window using any of the following ways:

- By clicking **View→Properties** Window on the menu bar
- By pressing the F4 key on the keyboard
- By right-clicking an item added to a project and selecting the Properties option from the context menu that appears
- By right-clicking the opened form or any control added to the form in the Design window and selecting the Properties option from the context menu

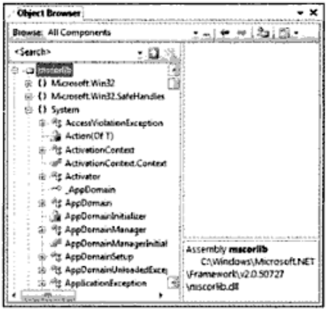
Fig.VB-1.23 shows the properties of `Form1` in the Properties window:

Fig.VB-1.23

You can notice in Fig.VB-1.23 that the properties are grouped under different categories, such as Appearance, Behavior, Data, and Design. The Alphabetical button (second button from the left) in the toolbar of the Properties window can be used to arrange these properties in an alphabetical order. Similar to displaying properties of a form or a control, you can also display its events in the Properties window, by clicking the button with lightening sign in the toolbar of the Properties window. You can also select some other form or control from the drop-down list present above the toolbar of the Properties window to display its properties or events in the Properties window.

Object Browser

The Object Browser enables you to view or search objects, such as namespaces, classes, structures, interfaces, and enums, along with their members, such as variables, properties, methods, and events. You can also use the Object Browser to view the information related to an item, such as properties, methods, and classes, in your code. You can do so by right-clicking the item in the Code Editor and selecting the Go To Definition option from the context menu that appears. You can open the Object Browser by either clicking View→Object Browser on the menu bar or pressing **CTRL+ALT+J** key combination on the keyboard. Fig.VB-1.24 shows the Object Browser:



Browse: All Components

<Search>

Assembly **mscorlib**
C:\Windows\Microsoft.NET\Framework\v2.0.50727\mscorlib.dll

Fig.VB-1.24

Class View Window

The Class View window enables you to view the items, such as classes, methods, and properties, associated with a project, in a tree structure. You can also use this window to search an item associated with a project. You can open the Class View window by either clicking View→Class View on the menu bar or pressing **CTRL+SHIFT+C** key combination on the keyboard. Fig.VB-1.25 shows the Class View window:

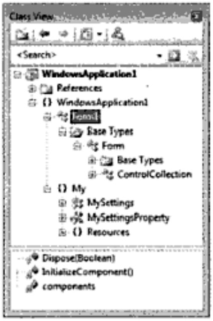


Fig.VB-1.25

Summary

In this chapter, we learned about:

- ❑ Versions of .NET Framework
- ❑ Benefits of .NET Framework
- ❑ Architecture of .NET Framework 3.5
- ❑ How to install Visual Studio 2008
- ❑ How to open Visual Studio 2008
- ❑ Visual Studio 2008 IDE

Introduction

With ever increasing need for more computer professionals, people now a days are opting for IT-enabled jobs. There are lots of programming languages available these days to learn, so one can get confused which programming language should be learned. People likely opt for languages that involve less code and at the same time, provide visually powerful programming environment for developing applications. Visual Basic is one of those languages that involve less code and is rich in graphical user interface. For a beginner, it is one of the easiest languages to learn. Before the introduction of .NET Framework, the last version of Visual Basic was Visual Basic 6.0. The latest version of Visual Basic is Visual Basic 2008 and it was released in November 2007.

In this chapter, we will learn how to create a Console application in Visual Basic 2008, keywords, operators, data types, variables, and constants. We will also learn about selection statements, iteration statements, and arrays in Visual Basic 2008.

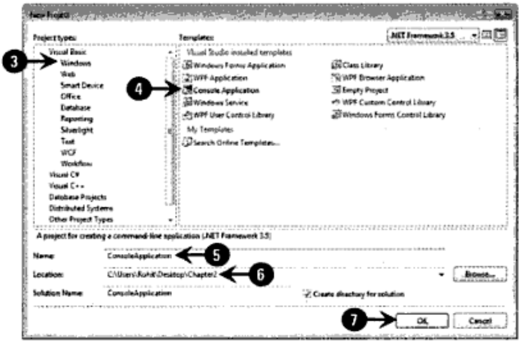
Let's first start by learning how to create a Console application in Visual Basic 2008.

Creating a Visual Basic 2008 Console Application

Visual Basic 2008 Console application is a command-line oriented application that allows the user to read characters from console, write characters to the console, and is executed in the Command Prompt. Console application does not have its user interface and it looks similar to MS-DOS application, which reminds you of the MS-DOS operating system. Console applications work in a Command Prompt and do not have any support for graphics and graphical devices, such as mouse, joystick, and so on.

To create a Console application, perform the following steps:

1. *Start* Visual Studio 2008 by clicking the **Start→All Programs→Microsoft Visual Studio 2008→Microsoft Visual Studio 2008** option.
2. *Click* **File→New→Project** on the menu bar or press the **CTRL+SHIFT+N** keys together. This opens the New Project dialog box, as shown in Fig.VB-2.1:



A project for creating a command-line application (.NET Framework 3.5)

Name: ConsoleApplication

Location: C:\Users\Rohit\Desktop\Chapter2

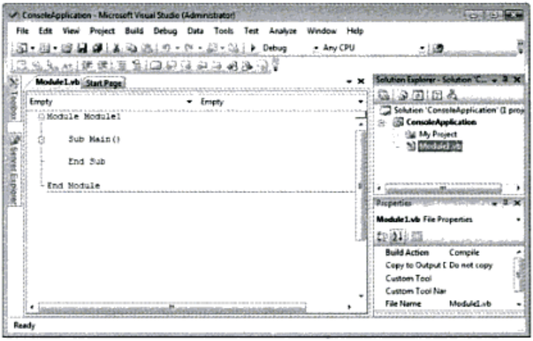
Solution Name: ConsoleApplication

☑ Create directory for solution

Fig.VB-2.1

3. *Select* **Visual Basic→Windows** in the **Project types** pane, as shown in Fig.VB-2.1.
4. *Select* **Console Application** in the **Templates** pane, as shown in Fig.VB-2.1.

5. *Type* a name for your application in the **Name** text box, as shown in Fig.VB-2.1. In this case, we have typed **ConsoleApplication.**
6. *Enter* the complete path of the folder where you want to save your application in the **Location** box, as shown in Fig.VB-2.1. In this case, we have entered **C:\Users\Rohit\Desktop\Chapter2.**
7. *Click* the **OK** button, as shown in Fig.VB-2.1. This closes the New Project dialog box and creates a new Console application, as shown in Fig.VB-2.2:


ConsoleApplication - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Data Tools Test Analyze Window Help
Debug
Any CPU
Module1.vb
Start Page
Toolbox
Server Explorer
Empty
Empty
Solution Explorer - Solution 'C...
Solution 'ConsoleApplication' (1 proj
ConsoleApplication
My Project
Module1.vb
Properties
Module1.vb File Properties
Ready

```
Module Module1

    Sub Main()

    End Sub

End Module
```

Build Action Compile
Copy to Output [Do not copy
Custom Tool
Custom Tool Nar
File Name Module1.vb

Fig.VB-2.2

When you create a new Console application in Visual Basic 2008, by default, the application contains a module file. A module file is a file with **.vb** extension and it contains the code for your Visual Basic program. You can notice in Fig.VB-2.2 that a moɑ.ɩle file, **Module1.vb**, has been created in the **Solution Explorer**. The default code for the **Module1.vb** file is as follows:

The preceding code defines a module, named **Module1**. A module is a Visual Basic type similar to a class. The module, **Module1**, contain a Sub procedure, **Main**. A Sub procedure is a series of Visual Basic statements enclosed by the **Sub** and **End Sub** statements. When the Console application runs, the **Main** Sub procedure is called automatically.

In this section, we have looked at how to create a Console application in Visual Basic 2008. In the next section, we discuss new features of Visual Basic 2008.

New Features of Visual Basic 2008

Visual Basic 2008 includes many new features that increase the productivity of Visual Basic developers and help them to create applications in Visual Basic more easily and efficiently. The new features of Visual Basic 2008 are as follows:

- Query expressions
- Local type inference
- Object initializers
- Extension methods
- Lambda expressions

- Anonymous types
- Nullable types
- Partial methods
- Support for XML

Let's discuss these different features one by one.

Query Expressions

A query expression is an expression that is used to query and transform data from a LINQ enabled data source (you read about LINQ in detail in *Chapter 9, Introducing Language-Integrated Query*). Query expressions comprise of the list of different types of clauses that help the user in performing different types of tasks. These clauses are as follows:

- **Aggregate clause:** Applies one or more aggregate functions to a collection.
- **Distinct clause:** Restricts the value of the current range variables. As a result, the duplicate values are deleted from the query results.
- **From clause:** Provides the collection and range variables for the query.
- **Group By clause:** Helps in grouping the elements of the result of the query. This clause is also helpful in applying aggregate functions to each group.
- **Group Join clause:** Combines two collections into a single hierarchical collection.
- **Join clause:** Joins two collections into a single collection.
- **Let clause:** Computes the value of the query result and then assign that value to a new variable in the query.
- **Order By clause:** Determines the order of sorting for columns in a query.
- **Select clause:** Declares the set of range variables for the query.
- **Skip clause:** Segregates some specified elements from a group of elements. As a result, it returns only remaining elements.
- **Skip While clause:** Excludes the elements in a collection unless some specific condition is not satisfied for the first time.
- **Take clause:** Provides certain specific number of adjacent elements from the start of the collection.
- **Take While clause:** Contains the elements unless the condition remains true, and ignores the remaining elements.
- **Where clause:** Provides certain specific filtering conditions for the query.

Local Type Inference

In Visual Basic 2008, compiler uses type inference feature to determine the local variables that are declared without using the **As** keyword. Normally, when we declare a variable, we use the **As** keyword to specify its data type. Such type of declaration is called explicit type declaration. However, with the type inference feature, you can skip the **As** keyword from the initialization expression. In this way, you can declare variables without explicitly specifying their data type.

The following code snippet shows how to declare a variable using explicit type declaration:

```
Dim k As Integer=9
```

The following code snippet shows how to declare a variable using the local type inference feature:

```
Dim number=7
```

Object Initializers

Object initializer is an important feature in Visual Basic 2008. It specifies the properties of the objects. An object initializer lets you assign values to the accessible fields or properties of an object without explicitly creating and invoking a constructor. You can use object initializers with the help of a single expression.

However, you can also use the object initializer in other contexts, such as Language Integrated Query (LINQ) query expressions (you will learn more about LINQ in *Chapter 9, Introducing Language-Integrated Query*).

Extension Methods

Visual Basic 2008 introduces extension methods to specify that a set of methods available on an instance of a type, such as class, interface, is open for extension. This means that you can add new methods to existing *class* without rewriting it or deriving a new class from it. Therefore, extension methods increase the set of methods available in any type. To create a extension method, use the **<Extension()>** attribute from the **System.RunTime.CompilerServices** namespace.

Lambda Expressions

A lambda expression is a function without a name that evaluates a single expression and returns a value. Visual Basic 2008 introduces lambda expressions as anonymous methods that contain expressions and statements to replace the delegate functions. These expressions are used in Visual Basic 2008 for declaring method code inline. You can use lambda expressions to create delegates or expression tree types. The following code snippet shows how to declare a lambda expression:

```
Dim Addition = Function(num as Integer) num+1
```

Anonymous Types

Visual Basic 2008 offers anonymous types as a new feature. This feature allows you to create objects without letting you to define a new type. Instead, the compiler itself generates the type. This type is also not having any useable name, inherits directly from **Object** type and contains the properties that are specified during the declaration of the objects. Since the name of the data type is not specified, that is why it is referred to as anonymous type.

Following code snippet declares and creates variable entity as an instance of an anonymous type that has two properties, Quality and weight.

The syntax for declaring anonymous type is as follows:

```
Dim entity=New With{Key.Quality="A", .Weight=1.97}
```

Remember query expression uses anonymous types for combining columns of data selected by a query. Although using anonymous types, you can select any number of columns.

Nullable Types

In certain situations, you may be working with a value type that does not have a defined value. For instance, in a database, you have to make a distinction so that the field may either have a meaningful assigned value or have no assigned value. Therefore, value types can be extended in such a way that they can have either normal values or a null value. Such an extension refers to a nullable type.

Each nullable type is constructed from the generic **Nullable(T)** structure. The following code illustrates how to construct a nullable Boolean type and then declares a variable of that type. There are following three ways of writing the declaration:

Variable backtowork can hold any of the three values, `True`; `False`; or no value. It is easy to declare variables and properties with nullable types. You can also declare an array with elements of nullable types. You can even use a **Function** procedure to return a nullable type.

Note

> You cannot create a nullable type on a reference type such as an array, a string, or a class.

Partial Methods

While working on a large project, there are times when you need to split the definition of a class, a struct, an interface, or a method over two or more source files. Each source file should contain a section of the class or method definition, and when you compile the application, the compiler combines all the source files. Splitting and combining a large project helps a group of developers to work on a single class of the large project simultaneously. You can use the **Partial** keyword to split a class definition. The **Partial** keyword determines whether or not the other source files of the class, interface, struct, or method can be defined in the namespace. However, note that all the source files must use the **Partial** keyword and be available during compilation to form the final type. A partial method provides the means for incorporating certain custom-generated code into the built-in designer code. This method is primarily used for the purpose of data validation. You can create the partial method for data validation in two steps:

- Define the method signature
- Write the implementation

The designer of the code defines the method signature and one or more calls to the method, then the developers provide the implementations for the methods, resulting in customizing the behavior of the generated code. In case no implementation is provided, the calls made to the method are removed by the computer, resulting in no additional performance overhead.

Support for XML

In Visual Basic 2008, you can now use XML as data types, to make the process of XML creation, XML transformation, XML modification, and query XML fast and easy. Visual Basic 2008 provides support for XML in the form of LINQ to XML (You learn about LINQ in detail in *Chapter 9, Introducing Language-Integrated Query*) using XML literals and XML axis properties, as discussed here:

- **XML literals:** Enable you to include XML directly in your code.
- **XML axis properties:** Enable you to access child nodes, descendant nodes, and attributes of an XML literal.

In this section, we have learned about new features of Visual Basic 2008. In the next section, we discuss keywords of Visual Basic 2008.

Visual Basic 2008 Keywords

A keyword is a word that is used for a specific task. Visual Basic 2008 provides the two types of keywords—reserved and unreserved. Reserved keywords are the one, which you cannot use for your programming elements such as variables and procedures. On the other hand, unreserved keywords are those keywords, which you can use for your programming elements such as variables and procedures. However, it is suggested to avoid using these as keywords, as it leads to subtle errors and makes the code difficult to understand.

Table 2.1 lists the reserved keywords available in Visual Basic 2008:

Table 2.2 lists the unreserved keywords available in Visual Basic 2008:

In this section, we have learned about keywords of Visual Basic 2008. In the next section, we discuss operators of Visual Basic 2008.

Visual Basic 2008 Operators

Operators play a vital role in performing some computation or other operations, such as arithmetical and logical operations on the operands. Therefore, the operator refers to the operations to be performed in the expression. All the operators have their own specified precedence. Operator precedence determines which operation will be executed first in an expression that involves multiple operations. Different operators available in Visual Basic 2008 are as follows:

- Arithmetic operators
- Assignment operators
- Concatenation operators
- Comparison operators
- Logical operators

Now, we discuss all these operator types one by one, starting with arithmetic operators.

Arithmetic Operators

The operators that are used for performing arithmetic operations such as subtraction, multiplication, and division, are called Arithmetic operators. Different arithmetic operators that are available in Visual Basic 2008 are given here in Table 2.3:

Here, we have discussed the arithmetic operators found in Visual Basic 2008. Now, we discuss the assignment operators found in Visual Basic 2008.

Assignment Operators

The operators that are used for assigning the values of one variable to another variable after performing different operations such as XOR operation, multiplication, division, integer division, addition, and subtraction. The following are the different assignment operators that are available in Visual Basic 2008, as listed in Table 2.4:

Here, we have discussed the assignment operators found in Visual Basic 2008. Now, we discuss the concatenation operators found in Visual Basic 2008.

Concatenation Operators

Many times, you may need to combine two text strings to display a message. The process of combining two text strings into one string is called string concatenation and the operators that are used to perform string concatenation are called concatenation operators. The following are the different concatenation operators that are available in Visual Basic 2008, as listed in Table 2.5:

The & (ampersand) and + (addition) operators are used for concatenating two text strings. The + operator is used for concatenating numeric operands with string operands whereas the & operator is used only for concatenating strings, as shown in Table 2.5.

Here, we have discussed the concatenation operators found in Visual Basic 2008. Now, we discuss the comparison operators supported by Visual Basic 2008.

Comparison Operators

Comparison operators are the operators that are used to compare two expressions. We can use this operator to compare numeric values, strings, and objects. A comparison operation is an operation that returns a Boolean value as a result. A Boolean value can be either **True** or **False.** Table 2.6 presents a list of all the comparison operators supported by Visual Basic 2008 and explains how they can be used:

logical operators supported by Visual Basic 2008.

Logical Operators

Logical operators are used to compare the expressions that involve Boolean operators and the result obtained from these operators is a Boolean value. Logical operators can be classified into the following three types:

- ❑ Unary logical operators
- ❑ Binary logical operators
- ❑ Short-Circuiting Logical Operations

Out of the above three types of logical operators, we will describe first two most common Logical Operators: Unary logical operators and Binary logical operators one by one.

Unary Logical Operators

A logical operator that involves only one operator is called a unary logical operator. **Not** operator is the unary logical operator in Visual Basic 2008. It performs the logical negation operation of an expression that evaluates a Boolean value. The **Not** operator returns exactly opposite of the operand on which it is applied. For example, if the expression evaluates to be **True**, then the result after applying the **Not** operator will be **False**, and vice versa. The following code snippet explains how we can use the **Not** operator:

After execution of the preceding lines of code, the Boolean variables X and Y will store the Boolean values **False** and **True**, respectively. As we know, the expression **(5>3)** evaluates to **True**; however, when this expression is preceded by the **Not** operator, that is, **Not(5>3)**, it returns **False**. In the same way, when the expression **(2>7)** is preceded by the **Not** operator, that is, **Not(2>7)**, it returns **True**.

Binary Logical Operators

Binary operators are those operators that take two operands (expressions). Three most commonly used binary logical operators available in Visual Basic 2008 are as follows:

- And
- Or
- Xor

And

The **And** operator is used to perform logical conjunction of two Boolean expressions. If both Boolean expressions evaluate to be **True**, then the final result after applying the **And** operator will also be **True**. However, if one of the two Boolean expressions evaluates to be **False**, then the final result after applying the **And** operator will also be **False.**

The following code snippet explains how we can use the **And** operator in our code:

True and **False**, respectively.

Or

The **Or** operator is used to perform logical disjunction of two Boolean expressions. If both Boolean expressions evaluate to be **False**, then the result after applying the **Or** operator will also be **False**. For all other cases, the result of applying the **Or** operator will be **True**. The following lines of code explain how we can use the **Or** operator in our code:

False and **True**, respectively.

Xor

The **Xor** operator is also used to perform logical disjunction of two Boolean expressions. If only one Boolean expression evaluates to be **True**, then the final result after applying the **Xor** operator will be **True**. However, if both the Boolean operators evaluate to be **True** or **False**, the result of applying the **Xor** operator will be **False**. The following code explains the use of the **Xor** operator:

True and **False**, respectively.

In addition to all the operators discussed so far, some other operators are also used in Visual Basic 2008. These are **AddressOf**, **GetType**, and **TypeOf** operators. The **AddressOf** operator gets the address of the procedure, the **GetType** operator gets the information about a type, and the **TypeOf** operator compares an object reference variable to a data type.

Here, we have discussed the logical operators found in Visual Basic 2008. Now, we learn about the precedence in which operators are executed in an application, developed by using Visual Basic 2008.

Operator Precedence

In Visual Basic 2008, you can use a large set of operators simultaneously in the expression to perform required calculations. However, using two or more operators may conflict the operator precedence; that is, which operation we will perform first. For this purpose, Visual Basic 2008 makes use of the precedence of different operators available in Visual Basic 2008. Therefore, operator precedence is a set of rules that specifies the order in which the compiler evaluates expressions. The operators associate with either the expression on its left or the expression on its right, and this is known as the associativity of the operator. To understand operator precedence more clearly, consider the following expression that contains two arithmetic operators:

```
5 + 3 * 2
```

In preceding example, if we first add 5 and 3 and then multiply the result by 2, the result will be 16. However, if we first multiply 3 to 2 and then add 5 to the result, the result will be 11. You can notice that the result is not the same in both cases. To avoid such contradictory results, Visual Basic 2008 has its own rules of precedence for all the operators supported by the language.

The arithmetic and concatenation operators have higher precedence than the comparison and logical operators. Comparison operators have higher precedence than the logical operators. However, all comparison operators have equal precedence; that is, they are evaluated in the order, left to right, in which they are arranged in the expression.

The arithmetic operators have the highest precedence, therefore, they are arranged in the order of highest precedence to lowest precedence, as follows:

- Exponentiation (^)
- Unary identity and negation (+, -)
- Multiplication and division (*, /)
- Integer division (\)
- Modulus arithmetic (Mod)
- Addition and subtraction (+, -)

Now, the concatenation operators and their order of precedence are as follows:

- String concatenation (+)
- String concatenation (&)

The Comparison operators have the same precedence and are evaluated from, left to right in the expression. Order of precedence of comparison operators is as follows:

- Equality (=)
- Inequality (<>)
- Less than, greater than (<, >)
- Greater than or equal to (>=)
- Less than or equal to (<=)
- Like
- Is

The Logical/Bitwise operators have the precedence order, from highest to lowest:

- Negation—(Not)
- Conjunction—(And, AndAlso)
- Disjunction—(Or, OrElse, Xor)

In this section, we have discussed Visual Basic 2008 operators and their precedence. In the next section, we discuss data types supported by Visual Basic 2008.

Data Types

A data type determines the type of data that is stored in a variable. It can be Integer, String, Boolean, and so on. For creating a variable of a particular data type, we should first know the range of possible values that the data type allows. The various data types supported by Visual Basic 2008, their storage size, and the range of values they allow are given in Table 2.7:

In this section, we have discussed data types in Visual Basic 2008. In the next section, we discuss how to make a conversion between different data types in Visual Basic 2008.

Data Type Conversion

Sometimes, we may be required to assign value stored in the variable of one data type to the variable of some other data type. When we do this, the value of the data type changes or modifies according to the target data type. But before assigning the value, first we should know whether conversion between the two data types is possible or not.

Conversion is always dependent on compatibility. If the values of data types can be assigned to each other then data types are considered as compatible data types, otherwise not. For example, we can assign value stored in a Byte variable to an Integer variable (compatible data types) whereas we cannot assign value stored in an Integer variable to a Char variable (incompatible data types).

Moreover, we can assign value stored in variable with data type having smaller storage size to variable with data type having larger storage size without any problem. However, we must ensure that the value we are going to assign does not fall outside the range of values supported by the target data type.

Visual Basic 2008 provides some conversion functions, which we can use while assigning the value. A list of such functions and their purpose are given in Table 2.8:

CUShort	**Converting to UShort type**

In this section, we have discussed data type conversion in Visual Basic 2008. In the next section, we discuss about variables in Visual Basic 2008.

Variables

A variable is an identifier that denotes a storage location in the memory. By using a variable's name in your program, you are referring to the information stored at that location. Every variable has a type that determines what values can be stored in that variable. A variable can store different values during the execution of a program. Each variable has a data type and it can store only those values that fall in the range of values supported by its data type. You can give any name to a variable but it should be meaningful because it makes the code more readable. Some examples of meaningful variable names are salary, height, name, age and total marks.

While assigning name to a variable in Visual Basic 2008, we should follow the following rules:

- A variable name can only contain alphabets, digits, and underscores
- A variable name should not begin with a digit or any numeric value
- A variable name cannot contain a blank space
- Keywords cannot be used as a variable name

Besides using the preceding rules, we should also try to use meaningful names for naming the variables. For example, instead of using variable names such as a, b, c, and d, we can use more specific variable names such as age, height, weight, and grade.

A variable can be declared in Visual Basic 2008 at class, module, procedure, or block level using the **Dim**

A variable can be assigned a value at the time of its declaration by using the = sign. Variables can be assigned a value at the time of their declaration as follows:

The default data type for any variable is **Object**. If any variable is not assigned a value, the default value is assigned to it according to its data type. The rules to determine, which default values are assigned to the uninitialized variables, are as follows:

- **0**, for all numeric types (including Byte)
- **Binary 0** for Char
- **Nothing** for all reference types such as Object, String, and all arrays. It indicates that no object is associated with the reference.
- **False** for Boolean
- **12:00 AM of January 1 of the year 1** for Date (01/01/0001 12:00:00 AM)

We can also declare multiple variables of the same data type without repeating the type, as follows:

```
    Dim count1, count2 As Integer
```

Variable names can be prefixed to indicate their data type, which helps when someone else is reading your code. Use of variable prefixes is optional. Table 2.9 lists some of the prefixes that have become conventional for the Visual Basic data types:

Table 2.9: Variable Prefixes

Data type	Prefix
`Integer`	int
`Long`	lng
`Object`	obj
`Single`	sng
`String`	str
User-defined type	udt

In this section, we have discussed variables in Visual Basic 2008. In the next section, we discuss about constants in Visual Basic 2008.

Constants

Constants are the names given to the values that do not change during the execution of the program. Declaring a constant is useful when we have to use a value at many places in a program. If we have declared a value as a constant at one point then, we can use the name of the constant instead of the value for further references, and all the instances of that value can be modified by changing only the value of the constant at the point of declaration. In Visual Basic 2008, constants are declared with the keyword Const. The syntax of declaring constants in Visual Basic 2008 is as follows:

```
[ <attrlist> ] [{ Public | Protected | Friend | Protected Friend | Private }] [
Shadows ] Const constantlist
```

Some parts of the preceding syntax have already defined in Table 2.1. The remaining term is a constantlist, which is declared in the statement. Each constant in the constantlist must use the following syntax:

```
name [ As type ] = initexpr
```

An explanation of the terms used in the preceding syntax is given in Table 2.10:

Table 2.10: Explanation of Syntax of constantlist

Term	Explanation
name	
type	
initexpr	

A constant can be declared as follows:

```
Const Pi = 3.14159
```

In this section, we have discussed about constants in Visual Basic 2008. In the next section, we learn about selection statements in Visual Basic 2008.

Selection Statements

Selection statements are the statements, which changes the program flow based upon whether a certain condition is fulfilled or not. This condition is Boolean expression, which is checked before the execution of a block of code inside the selection statement. Visual Basic supports two types of selection statement, as follows:

- ❑ If Else statement

- Select Case statement

Let's now discuss these statements, one by one.

If Else Statement

This statement allows you to test whether a certain condition is fulfilled or not. If the condition is fulfilled, the program control is transferred to the blocks of code that are inside the **If** statement; otherwise, the program control is transferred to another block of code. The syntax for an **If Else** statement is as follows:

be executed and ultimately the If statement will terminate. If the condition is **False**, then the condition in the **ElseIf** statement is tested and corresponding block of code is executed. If none of the conditions, **If** and **ElseIf**, are **True** and the **Else** statement is present, the code inside the **Else** statement is executed.

Let's perform these steps to see how an **If Else** statement can be implemented in a Console application:

1. *Create* a new Console application with the name **IFELSE**.
2. *Add* the following code to **Module1.vb** file:

3. *Run* the application by pressing the **F5** key on the keyboard.

Now, if you enter the value, 2008, you will get the output, as shown in Fig.VB-2.3:

Fig.VB-2.3

On the other hand, if you enter the value, 2007, you will get the output, as shown in Fig.VB-2.4:

Fig.VB-2.4

Select Case Statement

This statement compares the value of an expression with different values of other expressions in some given **Case** statements. Now, if the **Case** matches with the specified test expression, the programs control transfers to that **Case** statement and the statements that are inside that **Case** statement get executed. The syntax for a

statement. You can use multiple **Case** statements in a **Select** statement. Each **Case** statement is having a different value that is tested against **testexpression**. Finally, the code that matches **testexpression** is executed.

Let's perform these steps to see how a **Select Case** statement can be implemented in a Console application:

1. *Create* a new **Console** application with the name **SELECTCASE.**

3. *Run* the application by *pressing* the **F5** key on the keyboard. The output of the application is displayed, as shown in Fig.VB-2.5:

Fig.VB-2.5

In this section, we have discussed about selection statements in Visual Basic 2008. In the next section, we learn about iteration statements in Visual Basic 2008.

Iteration Statements

Suppose you want to execute a set of statements 50 times in your program. Instead of writing the code 50 times, you can put the code inside a loop and specify a condition that the loop has to execute 50 times; thus, this condition saves the complexity and time involved in coding. An iteration statement executes a statement or a set of statements in a repeated manner. In Visual Basic, there are four types of iteration statements, which are as follows:

- While statement
- Do While statement
- For statement
- For Each statement

Let's now discuss these statements, one by one.

While Statement

It executes a set of statements as long as a given condition is **True**. The syntax for a **While... End While** statement is given as follows:

In the preceding syntax, the statements enclosed in the **While** statement are repeatedly executed till the condition is **True**. You can also terminate a **While** statement at any time with an **Exit While** statement.

Let's perform these steps to see how a While statement can be implemented in a Console application:

1. *Create* a new **Console** application with the name **WHILE.**
2. *Add* the following code to **Module1.vb** file:

as shown in Fig.VB-2.6:

Fig.VB-2.6

Do While Statement

The Do While statement is helpful in the execution of different set of statements for variable number of times. The syntax for a **Do While** statement is given here:

There are two types of **Do While** statements depending upon the execution of the conditions. In the first type of **Do While** syntax, **condition** is evaluated at the beginning and in the second type of **Do While**, **condition** is evaluated at the end of the loop.

Let's perform these steps to see how a **Do While** statement can be implemented in a Console application:

1. *Create* a new **Console** application with the name DOWHILE.
2. *Add* the following code to Module1.vb file:

3. *Run* the application by pressing the **F5** key on the keyboard. The output of the application is displayed, as shown in Fig.VB-2.7:

Fig.VB-2.7

For Statement

The **For** statement is one of the most popular statement among all Visual Basic statements. A statement is used when we have to execute a group of statements repeatedly for a specified number of times. The syntax for a **For** statement is given here:

The **For** statement needs a loop index for its execution, as it counts the number of loop iterations. In the preceding syntax, when the statement starts, the **counter** is automatically set to **start**. Each time during the looping cycle, **counter** is incremented by *one*. For every step, you can specify a positive or negative value in the **For** statement. In case, if you don't specify a value, it is set to a default value, 1. When the value of **counter** equals to *end*, the loop ends. **datatype** is the data type of **counter**, which is required when *counter* is not already declared. You can also terminate a **For** statement at any time with an **Exit For** statement.

Let's perform these steps to see how a **For** statement can be implemented in a Console application:

1. *Create* a new Console application with the name **FOR**.

3. *Run* the application by *pressing* the **F5** key on the keyboard. The output of the application is displayed, as shown in Fig.VB-2.8:

For Each Statement

The **For Each** statement iterates through all the items in a list, which may be an array or a collection of objects. The **For Each** statement works in the same way as the **For** loop works. The syntax of the **For Each** statement is as follows:

1. *Create* a new Console application with the name **FOREACH**.
2. *Add* the following code to **Module1.vb** file:

3. *Run* the application by *pressing* the **F5** key on the keyboard. The output of the application is displayed, as shown in Fig.VB-2.9:

Fig.VB-2.9

In this section, we have discussed about iteration statements in Visual Basic 2008. In the next section, we learn how about arrays in Visual Basic 2008.

Arrays

An array is a set of values that are logically related to each other, such as the highest marks per subject in a class of students. An array allows you to refer to these values by the same name and use a number called **Index**, for identifying the individual values. The individual values of an array are called the elements of the array. These elements are stored in the array with the index values starting from 0 to one less than the size of the array.

The syntax for declaring an array is as follows:

```
Dim ArrayName(ArraySize)  As datatype
```

The syntax for declaring an array that can hold ten integer elements is as follows:

```
Dim myArray(10) As Integer
```

The array myArray in the preceding example contains 10 elements, which are stored in the indexes starting from 0 to 9. Using an array is much easier than declaring 10 different variables, as it involves only single variable, myArray as compared to 10 different variables. To store and retrieve values to and from arrays in Visual Basic 2008, perform the following steps:

1. *Create* a new Console application with the name **Arrays**.
2. *Add* the following code to **Module1.vb** file:

3. *Run* the application by pressing the **F5** key on the keyboard. The output of the application is displayed, as shown in Fig.VB-2.10:

Fig.VB-2.10

Summary

In this chapter, we learned:

- ❑ How to create a Console application in Visual Basic 2008
- ❑ New features of Visual Basic 2008
- ❑ Keywords, operators, data types, variables, and constants
- ❑ Selection statements and iteration statements in Visual Basic 2008
- ❑ How to create arrays in Visual Basic 2008

Introduction

Programming, in simple words, means giving instructions to a computer to process the data and provide the required output. There are mainly two programming approaches, Procedure-Oriented Programming (POP) and Object-Oriented Programming (OOP).

In procedure-oriented programming, you first break the problem into smaller sections of code and then solve each section separately. Later on, all the solved sections of the program are integrated to solve the entire problem. Each small section of the code is written within a block of code, called a method. You can call one method from another. Therefore, these methods are dependent on each other. As a result, reusability of the application becomes difficult. Some programming languages that use the POP approach are COBOL, FORTRAN, and C.

The concept of OOP has been introduced to overcome the difficulty of limited or no reusability of code. OOP uses the concept of object for reusing the existing code. The concept of OOP revolves entirely around an object. An object in a programming language does not mean a tangible or visible thing rather it is an organization of code. An object may contain certain behaviors and properties. In programming languages, we represent a behavior as a method and properties of an object as attributes. Out of these two approaches, OOP is considered better to follow since it follows an approach, which is related to real world objects. Examples of languages that follow OOP concepts are C++, JAVA, and VB.

In this chapter, you learn about the four main principles of OOP: encapsulation, inheritance, abstraction, and polymorphism. In addition, you learn about classes and objects, structures, properties, interfaces, and namespaces used in Visual Basic.

Working with Classes and Objects

A class is a primary building block for the programs created in a programming language that follows the OOP approach, such as Visual Basic and C#. You can use classes to encapsulate variables and methods into a single unit. Let's look at an example to understand the concept of classes better. Suppose you need to create an object of a class, named **Bird**, in your program. To do so, you first need to create a class called **Bird**, which contains all the functionalities or behaviors and properties of any bird. You can then use the **Bird** class to create objects of the **Bird** class, as needed. For example, you can use the **Bird** class as a template to create an object named **Owl**. The **Owl** object of the **Bird** class would contain a property nocturnal, which would imply that owl is a nocturnal bird.

Classes allow you to define a self-contained environment wherein, you control all the functions that can be applied to a given set of data and also control access to the data. The declaration of a class starts with the **Class** keyword followed by the class name. A class is similar to a container that may have data members (variables, constants, or fields) and member functions (methods, properties, events, indexers, operators, constructors, and destructors), and other classes. A class also supports inheritance, which is a mechanism in which a derived class extends a base class. The syntax of a class in Visual Basic is as follows:

Note

You will learn more about inheritance later in the chapter.

To access the data members and member functions of a class, you need to create an object of that class. The syntax to create an object of a class in Visual Basic is as follows:

```
<access-modifier> <ObjectName> As <ClassName>   'Declaration
<ObjectName> = New <ClassName()>  'Instantiation
```

These are some important concepts that will help you while working with classes and objects:

- □ Access modifiers

- Methods
- Constructors and destructors
- Partial classes
- Shared methods
- Extension methods

Let's learn about each of these in detail next.

Access Modifiers

Access modifiers in Visual Basic are keywords used to specify the accessibility of a member or a type. Access modifiers help you to avoid jumbling of data and methods with the existing code, as well as protect an object of a class from outside interference. The access modifier protects an object by defining a certain scope to access its data and methods in a restricted manner. You can declare a class and its methods with an access modifier. However, one method can contain only one modifier. The different types of access modifiers used in Visual Basic are listed in Table 3.1:

Note

Methods in Visual Basic

A method in Visual Basic is a block of code that contains a series of statements to perform an action. Every action in Visual Basic is performed in the context of a method. Methods are declared in a class by specifying the access level, the return value, the name of the method, and the method parameters. All these are collectively referred to as the signature of the method.

You can find an example of declaring a method in the following code snippet:

Let's now learn how you can work with constructors and destructors in a Visual Basic program.

Constructors and Destructors

Constructors and destructors are special types of methods. A constructor is a method that is called when an object is created. A destructor is a method that is called when the object is finally destroyed. The constructor initializes all class members whenever you create an object of the class, and the destructor destroys the values assigned to the class members, when the object is not required anymore. In Visual Basic, the `New` keyword is used to create constructors and the `Finalize` or `Dispose` methods are used to call a destructor.

The main features of constructors are:

- A constructor is a Sub procedure declared with the **New** keyword.
- Constructors have the same name as the class itself.
- Constructors do not have any return type.
- It is not mandatory to declare a constructor; it is invoked automatically.

A destructor (or finalizer) is called when an object is finally destroyed. Destructors are used to clean the instances of classes when the instances are not required. You cannot call a destructor in your application; they are invoked automatically. Visual Basic provides a garbage collection mechanism that is executed when the runtime environment finds it necessary or when an object is not destroyed until its reference count drops to 0. You have no way of telling when an object will be destroyed; and when the destructor will be called. You can, however, implement a custom method that allows you to control object destruction by calling the destructor.

Let's create an application named **ConstructorApp** to learn how to use constructors, by performing the following steps:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2008→Microsoft Visual Studio 2008** to open the Visual Studio 2008 IDE.
2. In the Visual Studio 2008 IDE, *click* **File→New→Project** from the menu bar to open the **New Project** dialog box.
3. In the **New Project** dialog box, *select* **Visual Basic→Windows** in the **Project types** pane and the **Console Application** option in the **Templates** pane.
4. *Enter* **ConstructorApp** in the **Name** text box to specify the name of the application and *specify* an appropriate location for the application in the **Location** box.
5. *Click* the **OK** button. The **New Project** dialog box closes and the **ConstructorApp** application is created.
6. In the **Module1.vb** file, *add* the code given in Listing 3.1:

Listing 3.1: Defining Constructor

In Listing 3.1, we have created three constructors: the first without any parameters, the second with one parameter, and the third with two parameters.

7. *Press* the **F5** key to run the application. The output of the code given in the **Listing 3.1** appears as shown in Fig.VB-3.1:

Fig.VB-3.1

Partial Classes

A partial class in Visual Basic is the class that enables you to specify the definition of a class in two or more source files. All the source files contain a section of the class definition. The definitions in the different source files are combined when the application is executed. You can divide a class into two or more partial classes, each stored in a separate file, so that you can work on each partial class separately.

You can declare a partial class by using the **Partial** keyword. The **Partial** keyword indicates that all the parts of the class must be available at compile time to generate the final class.

Let's create an application named **PartialClass** to learn how to use a partial class, by performing the following steps:

1. *Repeat* the steps 1-3 discussed while creating the **ConstructorApp** application.
2. *Enter* **PartialClass** in the **Name** text box to specify the name of the application and *specify* an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **New Project** dialog box closes and the **PartialClass** application is created.
4. In the **Module1.vb** file, *add* the code given in Listing 3.2:

Listing 3.2: Using a Partial Class

In Listing 3.2, the fields and constructors of the **Orders** class are declared in one partial class definition, and the **PrintOrders** method is defined in another partial class definition.

5. *Press* the **F5** key to run the application. The output of the code given in the Listing 3.2 appears, as shown in Fig.VB-3.2:

Fig.VB-3.2

Shared Methods

You can call a shared method without creating an instance of the class in which the shared method is declared. You can use the class name and the dot operator (.) to access a shared method outside the class.

Let's create an application named **SharedMethod** to learn how to use a shared method in a class, by performing the following steps:

1. *Repeat* the steps 1-3 discussed while creating the **ConstructorApp** application.
2. *Enter* **SharedMethod** in the **Name** text box to specify the name of the application, specify an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **New Project** dialog box closes and the **SharedMethod** application is created.
4. In the **Module1.vb** file, *add* the code given in Listing 3.3:

Listing 3.3: Using Shared Methods

In Listing 3.3, two shared functions, **reciprocal** and **fraction**, are created inside a class, **MathFunction**, and are accessed using the name of the class.

5. *Press* the **F5** key to run the application. The output of the code given in the Listing 3.3 appears, as shown in Fig.VB-3.3:

Fig.VB-3.3

Extension Methods

Extension method is one of the new features in Visual Basic 2008. An extension method is a technique used to extend a class without deriving a new class from that class. The behavior of extension methods is similar to that of shared methods. An extension method can either be a Sub procedure or a function. You cannot define an extension property, field, or event. All extension methods must be marked with the **<Extension()>** extension attribute from the **System.Runtime.CompilerServices** namespace.

Let's create an application named **ExtensionMethod** to learn how to use an extension method, by performing the following steps:

1. *Repeat* the steps 1-3 discussed while creating **ConstructorApp** application.
2. *Enter* **ExtensionMethod** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **New Project** dialog box closes and the **ExtensionMethod** application is created.
4. In the **Module1.vb** file, add the code given in Listing 3.4:

Listing 3.4: Implementing an Extension Method

Now, to call the extension method you need to add another module, named `Module2.vb`, to the `ExtensionMethod` application.

5. *Right-click* the application **ExtensionMethod** in the **Solution Explorer**, as shown in Fig.VB-3.4:

Fig.VB-3.4

6. *Click* **Add→New Item** (Fig.VB-3.4). This opens the **Add New Item** dialog box, shown in Fig.VB-3.5:

Fig.VB-3.5

7. *Select* **Module** from the **Templates** pane and *click* the **Add** button (Fig.VB-3.5). This adds Module2.vb file to the **ExtensionMethod** application.
8. *Add* the code in Listing 3.5 to **Module2.vb** file:

Listing 3.5: Code in Module2.vb File

After adding the code to **Module2.vb** file, you need to set the Startup object of **ExtensionMethod** application.

9. *Right-click* **ExtensionMethod** project and *select* the **Properties** option from the context menu, as shown in Fig.VB-3.6:

Fig.VB-3.6

This opens the Project designer, where you can set project properties, as shown in Fig.VB-3.7:

Fig.VB-3.7

10. *Select* the **Application** tab (Fig.VB-3.7).
11. *Select* **Module2** from the **Startup object** drop-down list (Fig.VB-3.7).
12. *Press* the **F5** key to run the application. The output of the application appears, as shown in Fig.VB-3.8:

Fig.VB-3.8

Encapsulation

Encapsulation is the process of hiding the irrelevant information and showing only the relevant information to the user. It is a way to organize data and methods into a single unit; therefore, preventing the data from being modified by unauthorized users. Encapsulation is implemented through access modifiers. Access modifiers help to implement this feature by defining a scope to access data and methods in a restricted manner. Consequently, you can describe encapsulation as the ability of an object to hide its internal data and methods, and make only the intended parts of the object programmatically accessible.

In OOP terms, encapsulation is the process of wrapping data and members in a class. Only specific methods or properties can access the private members of a class. In other words, encapsulation is an approach to hide the internal state and behavior of a class or an object from unauthorized access. It restricts the external user from sharing and manipulating data, therefore minimizing the chances of data corruption. The advantages of encapsulation are as follows:

- **Data hiding through the use of the Private access modifier:** Encapsulation provides a way to protect our data from unauthorized access. Therefore, instead of defining our data as **Public**, we declare specific fields, such as, data members, member functions, properties, or indexers, as **Private**. The private data can be indirectly operated in two ways, first, through the accessor and mutator methods, and secondly, through a named property.
- **Increasing the maintainability of the code:** Encapsulation increases the maintainability of the code by showing only the relevant information to the user.
- **Preventing data corruption:** Encapsulation prevents data corruption by specifying member variables of a class as private, so that they can only be accessed by specific methods or properties.
- **Wrapping up of data members and member functions in a class:** Encapsulation binds the data members and the member functions of a class into a single unit. This is the most important feature of encapsulation.

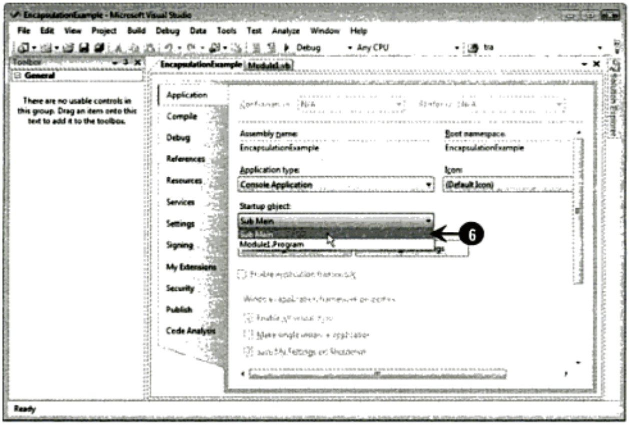
Let's create a Console application named **EncapsulationExample** to learn how to implement encapsulation, by performing the following steps:

1. *Repeat* the steps 1-3 discussed while creating the **ConstructorApp** application.
2. *Enter* **EncapsulationExample** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **New Project** dialog box closes and the **EncapsulationExample** application is created.
4. In the **Module1.vb** file, *add* the code given in Listing 3.6:

Listing 3.6: Implementing Encapsulation

accessible only in the class where they are decalred. To implement encapsultaion, the **Private** keyword is used.

5. Now, *open* the Project Designer as done earlier in the **ExtensionMethod** application.
6. *Set* the **Startup object** to **Sub Main**, as shown in Fig.VB-3.9:



EncapsulationExample - Microsoft Visual Studio

Assembly name: EncapsulationExample

Root namespace: EncapsulationExample

Application type: Console Application

Icon: (Default Icon)

Startup object: Sub Main

Fig.VB-3.9

7. *Press* the **F5** key to run the application. The output of the code given in the Listing 3.6 appears as shown in Fig.VB-3.10:

Fig.VB-3.10

Inheritance

The most important reason for using OOP is to promote the reusability of code and eliminate redundancy of code. To ensure reusability, the object oriented languages promote the use of inheritance. Inheritance is defined as the property through which a child class obtains all the features defined in its parent class. A parent class is at a higher level in the class hierarchy as compared to the child class. For example, if we consider the **Parrot** class as a child class, it obtains its features from the parent class, the **Bird** class.

When a class inherits the properties of another class, the class inheriting the properties is called a derived class and the class that allows inheritance of its properties is called a base class. Inheritance in OOP is of four types:

- **Single inheritance:** Contains one base class and one derived class
- **Hierarchical inheritance:** Contains one base class and multiple derived classes of the same base class
- **Multilevel inheritance:** Contains a class derived from a derived class
- **Multiple inheritance:** Contains several base classes and a derived class

Visual Basic supports single, hierarchical, and multilevel inheritance. It does not support multiple inheritance directly because multiple inheritance supports multiple base classes and in Visual Basic, a derived class cannot have more than one base class. You can implement multiple inheritance in Visual Basic through interfaces.

Note

An interface is a collection of data members and member functions. You learn about interface in detail later in the chapter.

Inheritance represents a kind of relationship between two classes. Let's understand it through an example. Suppose there are two classes named **A** and **B** and the **B** class is derived from the **A** class, as shown in Fig.VB-3.11:

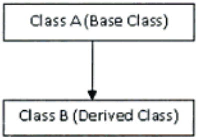


Fig.VB-3.11

In Fig.VB-3.11, **A** class is referred as the base class or the parent class and **B** class is referred as the derived class or child class. The derived class, **B**, is a completely new class and contains all the data and methods of its base class, and also includes its own data and methods.

In this section, you learn how to:

- Define a derived class
- Access the members of a base class
- Work with abstract classes
- Work with sealed classes

Defining a Derived Class

A derived class gains all the non-private data of its base class. It also gains the behavior of the base class, in addition to any other data or behavior defined for itself. It means the derived class, **B**, has two effective types: the type of the new class and the type of the class that it inherits.

Let's understand how we can define derived classes with the help of the following code snippet:

Accessing Members of the Base Class

When a class is derived from a base class, the members of the base class become the members of the derived class. The access modifier is used while declaring members of the base class to specify the access scope of the base class members in the derived class.

Let's create an application named **AccessingMembers** to learn how to access members of a base class, by performing the following steps:

1. *Repeat* the steps 1-3 discussed while creating **ConstructorApp** application.
2. *Enter* **AccessingMembers** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **New Project** dialog box closes and the **AccessingMembers** application is created.
4. In the **Module1.vb** file, *add* the code given in Listing 3.7:

Listing 3.7: Accessing Base Class Members

In Listing 3.7, **BaseClass** is the parent class and **DerivedClass** is the derived class.

5. *Press* the **F5** key to run the application. The output of the code given in the Listing 3.7 appears, as shown in Fig.VB-3.12:

Fig.VB-3.12

Working with Abstract Classes

If the objects of a class cannot be instantiated, it is called an abstract class. In Visual Basic, we have a single base class and can have multiple derived classes. If you have created a base class and want to ensure that no object of the base class is created later, you can make the base class as abstract. The **MustInherit** keyword in a class indicates that the class cannot be instantiated and is an abstract class. The basic purpose of an abstract class is to provide a common definition of the base class that can be shared by multiple derived classes.

Some characteristics of an abstract class are as follows:

- ❑ You cannot instantiate an abstract class directly. This implies that you cannot create an object of the abstract class. To use the members of the abstract class, you need to define a non-abstract class that inherits the abstract class. After you have defined the non-abstract class, you can access the members of the abstract class using the objects of the non-abstract class.
- ❑ An abstract class can contain abstract as well as non-abstract members.
- ❑ You must declare at least one abstract method in an abstract class.
- ❑ An abstract class is always public.

Let's create an application named **AbstractClass** to learn how to use an abstract class, by performing the following steps:

1. *Repeat* the steps 1-3 discussed while creating the **ConstructorApp** application.
2. *Enter* **AbstractClass** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **New Project** dialog box closes and the **AbstractClass** application is created.
4. In the **Module1.vb** file, *add* the code given in Listing 3.8:

Listing 3.8: Working with Abstract Classes

In Listing 3.8, we have created an abstract class, `Shape`. Inside the `Shape` class, we have created an abstract function, `Area`.

5. *Press* the **F5** key to run the application. The output of Listing 3.8 appears, as shown in Fig.VB-3.13:

Fig.VB-3.13

Working with Sealed Classes

A sealed class implies that the class cannot be used as a base class. The main purpose of using a sealed class is to take away the inheritance feature from the users so that they cannot derive a class from a sealed class. Once you have declared a class as sealed, no other class can inherit that class. The **NotInheritable** keyword is used to indicate that a class cannot be inherited. When you apply the **NotInheritable** keyword as a modifier to a class, the class becomes final.

Let's create an application named **SealedClass** to learn how to use a sealed class, by performing the following steps:

1. *Repeat* the steps 1-3 discussed while creating the **ConstructorApp** application.
2. *Enter* **SealedClass** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **New Project** dialog box closes and the **SealedClass** application is created.
4. In the **Module1.vb** file, add the code given in Listing 3.9:

Listing 3.9: Using Sealed Class

In Listing 3.9, we have created a sealed class, `SealedClass`, and use it in the **Main** Sub procedure.

5. *Press* the **F5** key to run the application. The output of Listing 3.9 appears, as shown in Fig.VB-3.14:

Fig.VB-3.14

Implementing Polymorphism

Polymorphism, in general, can be explained as one entity having multiple forms. In Visual Basic, you can use one procedure in multiple ways, with the help of polymorphism. For example, suppose you have to write a program for calculating the area of some geometrical shape. You can use the same procedure name for calculating the area of a circle, a triangle, or a rectangle, using different parameters.

The important features of polymorphism are as follows:

- ❑ Allows you to invoke methods of a derived class through the base class reference during runtime.
- ❑ Helps implement different implementations of multiple methods that are called through the same name.
- ❑ Helps call a method of a class irrespective of the specific implementation it provides.

In Visual Basic, there are two ways to implement polymorphism:

- ❑ Compile time polymorphism
- ❑ Run time polymorphism

Let's learn about them in detail.

Implementing Compile Time Polymorphism

When the compiler compiles a program, the compiler has the information about the method arguments. Accordingly, the compiler binds the appropriate method to the respective object at the compile time itself. This process is called compile time polymorphism or early binding. You can implement compile time polymorphism through overloaded methods and operators. The arguments passed are matched in terms of number, type, and order; and then the overloaded methods are invoked.

Compile time polymorphism is categorized as follows:

- ❑ Method Overloading
- ❑ Operator Overloading

Method Overloading

Method overloading is a concept in which a method behaves according to the number and types of parameters passed to it. Method overloading allows you to define multiple methods with the same name but with different signatures. When you call overloaded methods, the compiler automatically determines which method should be used according to the signature specified in the method call.

Note

> As described earlier, a method signature is the combination of the method's name along with the number and types of parameters.

Let's create an application named **MethodOverloading** to learn how to overload a method, by performing the following steps:

1. *Repeat* the steps 1-3 discussed while creating the **ConstructorApp** application.

2. *Enter* **MethodOverloading** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **New Project** dialog box closes and the **MethodOverloading** application is created.
4. In the **Module1.vb** file, *add* the code given in Listing 3.10:

Listing 3.10: Overloading Method

rectangle. When the **Area** Sub procedure is called, the compiler tries to find a method (Sub procedure) whose signature exactly matches with the method call. The retrieved Sub procedure is then executed. If the compiler finds multiple matches, it generates an error message.

5. *Press* the **F5** key to run the application. The output of the code given in the Listing 3.10 appears as shown in Fig.VB-3.15:

Fig.VB-3.15

Operator Overloading

All the operators have their specified meaning and functionality; for example, the **+** operator adds two numerals, and the **-** operator subtracts two numerals. However, at times, you might need to change the default functionality of an operator. You can do so by operator overloading; for example, the **+** operator can be overloaded to concatenate two strings, instead of numerals.

The mechanism of assigning a special meaning to an operator, according to user defined data types such as classes, is known as operator overloading. It is not possible to overload all the operators. Table 3.2 shows the overloading behavior of different operators:

Let's create an application named **OperatorOverloading** to learn how to overload a method, by performing the following steps:

1. *Repeat* the steps 1-3 discussed while creating the **ConstructorApp** application.
2. *Enter* **OperatorOverloading** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **New Project** dialog box closes and the **OperatorOverloading** application is created.
4. In the **Module1.vb** file, *add* the code given in Listing 3.11:

Listing 3.11: Overloading an Operator

In Listing 3.11, the **Operator –** method takes one argument of type **UnaryOperator** and changes the sign of data member **opr**.

5. *Press* the **F5** key to run the application. The output of the code given in the Listing 3.11 appears, as shown in Fig.VB-3.16:

Fig.VB-3.16

Runtime Polymorphism

Runtime polymorphism in Visual Basic is better known as overriding. Overriding allows a derived class to define a specific implementation for a method, other than the implementation defined by its base class. This implementation of the method in the derived class overrides or replaces the implementation of the method in its base class. This feature is known as runtime polymorphism because the compiler binds the method to an object while the program is being executed (runtime), and not when the program is being compiled.

When you call a method, the method defined in the derived class is invoked and executed instead of the one in the base class, but with the following conditions:

- You must declare the base class method as **Overridable**.
- You must implement the derived class method using the **Overrides** keyword.

Let's now learn how to override a method in Visual Basic.

Overriding a Method

A basic concept behind OOP is that you can create virtual methods, which can be overridden in a derived class. OOP allows the derived class to provide implementation of a method that is defined in the parent class. You can do this in Visual Basic with the **Overridable** and **Overrides** keywords. For this, you must explicitly define the Sub procedures in the base class as **Overridable**. You use the **Overridable** keyword in a Sub procedure to indicate that you want to have a base method overridden in a derived class. Using the **Overrides** keyword, you must specifically tell the compiler that you are intending to override an existing overridable Sub procedure.

Let's create an application named **MethodOverriding** to learn how you can overload a method by performing the following steps:

1. *Repeat* the steps 1-3 discussed while creating the **ConstructorApp** application.
2. *Enter* the name **MethodOverriding** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **New Project** dialog box closes and the **MethodOverriding** application is created
4. In the **Module1.vb** file, *add* the code given in Listing 3.12:

Listing 3.12: Overriding a Method

5. *Press* the **F5** key to run the application. The output of the code given in the Listing 3.12 appears, as shown in Fig.VB-3.17:

Fig.VB-3.17

Structures

A structure in Visual Basic is a user-defined value type. A structure can contain constructors, fields, methods, properties, and nested types, similar to a class. The **Structure** statement creates a structure of elements in Visual Basic. All elements of a structure are private by default, and a structure initializes its elements to the default value for each data type if nothing is specified. In Visual Basic, one big difference between structures and classes is that a structure does not support inheritance. If you do not use the **New** operator to call a constructor when you are declaring a structure variable, the structure object is created but the values of the structure variable are unassigned.

The syntax of a **Structure** statement is similar to that of a class, with the main difference being that a structure is a value type and a class is a reference type. The syntax of a **Structure** statement is:

In the preceding syntax:

- **attributelist**: Specifies the attributes to be applied to a declared programming element. This is optional.
- **accessmodifier:** specifies an access modifier, such as **Public**, **Protected**, **Friend**, and **Private**. This is optional.
- **Structure:** Is a keyword used to create a structure.
- **name:** Specifies the name of the structure.
- **End Structure:** Terminates the structure definition.

Let's now learn how you can use a structure in our Visual Basic code.

Using a Structure

Structure in Visual Basic allows you to create a new value-type objects that are similar to the built-in type objects such as **Integer**, **Decimal**, **Boolean**, and so on.

Let's create an application named **Structure** to learn how you use a structure, by performing the following steps:

1. *Repeat* the steps 1-3 discussed while creating the **ConstructorApp** application.
2. *Enter* the name **Structure** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **New Project** dialog box closes and the **Structure** application is created.
4. In the **Module1.vb** file, *add* the code given in Listing 3.13:

Listing 3.13: Using a Structure

In Listing 3.13, you can find that a structure, **X**, is created using the **Structure** keyword. The **X** is then called in the **Main** Sub procedure of the code.

5. *Press* the **F5** key to run the application. The output of the code given in the Listing 3.13 appears, as shown in Fig.VB-3.18:

Fig.VB-3.18

Working with Properties

In Visual Basic, properties are a standard part of the language itself. A property provides you a way to expose an internal data element of a class in a simple and intuitive manner. Visual Basic is one of the first languages to offer direct support for properties.

You can implement properties in Visual Basic with the **Get** property procedure and **Set** property procedure. You can create a property by defining an externally available name and then writing the **Set** and **Get** property procedures to implement the property. The **Get** property procedure is used to return the property value, and the **Set** property procedure is used to assign a new value to the property.

In this section, you will learn about using a property and using an anonymous type for read-only properties.

Using a Property

To read and write the data, Visual Basic introduced the concept of properties. It also prevents the data from external usage and modifications. You can declare a property in which you can use the **Get** and **Set** property procedure to retrieve the required value and to assign a value to the specified data.

Let's create an application named **Properties** to learn to use properties, by performing the following steps:

1. *Repeat* the steps 1-3 discussed while creating the **ConstructorApp** application.
2. *Enter* the name **Properties** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **New Project** dialog box closes and the **Properties** application is created.
4. In the **Module1.vb** file, *add* the code given in Listing 3.14:

Listing 3.14: Using a Property

As shown in Listing 3.14, a property **Name** is defined which takes **Get** and **Set** property procedures to get and assign value to the variable **empName.** In the **Main** sub procedure of the **Module1**, you can see that we have created an object of the **EmployeeDetail** class, named **detail,** using which we call the property **Name** to assign and retrive the values through the property procedures.

5. *Press* the **F5** key to run the application. The output of the code given in the Listing 3.14 ,appears as shown in Fig.VB-3.19:

Fig.VB-3.19

Using an Anonymous Type for Read-Only Properties

While creating properties for your application, you might need to create a few read-only properties as well. You can encapsulate these read-only properties into a single unit through anonymous types. Anonymous types provide a way to encapsulate the read-only properties of an object without having to first explicitly define a type. The compiler generates the type name as required but this type name is not available at the source code level. The compiler derives the properties type to generate the type name. In other words, anonymous types create unnamed structure types. In Visual Basic 2008, the declaration of an instance of an anonymous type starts with the **New** keyword, followed by the **With** keyword. The declaration uses an initializer list to specify the properties of the type. The statement, **New With {.Name = "Key Board"}**, creates an anonymous type instance with a member named **Name.** Its syntax is similar to object initializers except that object initializers specify a type between the **New** and **With** keywords. An anonymous type provides you an easy way to encapsulate a set of read-only properties into a single object without defining a new type.

Let's create an application named **AnonymousType** to learn to use an anonymous type for read-only properties by performing the following steps:

1. *Repeat* the steps 1-3 discussed while creating the **ConstructorApp** application.
2. *Enter* the name **AnonymousType** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **AnonymousType** application is created.
4. In the **Module1.vb** file, add the code given in Listing 3.15:

Listing 3.15: Using Anonymous Type for Read-Only Properties

In Listing 3.15, we define a new anonymous type and create an object named **product**.

5. *Press* the **F5** key to run the application. The output of the code given in the Listing 3.15 appears, as shown in Fig.VB-3.20:

Fig.VB-3.20

Interfaces

Interface is a collection of abstract data members and member functions. Interfaces in Visual Basic are introduced to provide the feature of multiple inheritance to classes. The methods defined in an interface do not have their implementation and only specify the number and types of parameters they will take and the type of values they will return. An interface is always implemented in a class.

Interface in Visual Basic is equivalent to an abstract base class. You cannot instantiate an object through an interface, but you can offer a set of functionalities that is common to several different classes.

Let's learn how we can define, implement, and inherit an interface in Visual Basic.

Defining an Interface

An interface is defined in the same way as defining a class. The difference is that a class is declared with the **Class** keyword and an interface is declared with an **Interface** keyword. The syntax of defining an interface in Visual Basic is as follows:

In the preceding syntax, the **Interface** keyword is used to define an interface.

Implementing an Interface

You can implement an interface using a class. An interface is implemented using the **Implements** keyword. The code-snippet for implementing an interface is given in Listing 3.16:

Listing 3.16: Implementing an Interface

In Listing 3.16, the **ImplementInterface** class implements two interfaces, named **MyFirstInterface** and **MySecondInterface**. The interface, named **MyFirstInterface**, contains a single method, **MyFirstMethod**, and the interface, named **MySecondInterface**, contains a method, **MySecondMethod.** Both these methods do not return any value. The interface methods, **MyFirstMethod** and **MySecondMethod**, are invoked with the help of an object of the **ImplementInterface** class.

Inheriting an Interface

You can derive a new interface from an existing interface in the same way as deriving a new class from a base class. The derived interface inherits all the members of the base interface in the same way as a derived class does. Suppose, you have a class that implements an interface derived from another interface. Now, when you call a method of the base interface using an object of the class, then the entire inheritance hierarchy is searched until the actual type of the method is found.

Let's create an application named **InterfaceInheritance** to learn to inherit an interface, by performing the following steps:

1. *Repeat* the steps 1-3 discussed while creating the **ConstructorApp** application.
2. *Enter* the name **InterfaceInheritance** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **New Project** dialog box closes and the **InterfaceInheritance** application is created.
4. In the **Module1.vb** file, *add* the code given in Listing 3.17:

Listing 3.17: Interface Inheritance

5. *Press* the **F5** key to run the application. The output of the code given in the Listing 3.17 appears, as shown in Fig.VB-3.21:

Fig.VB-3.21

In Fig.VB-3.21, you can see that **DerivedInterface** interface implements **BaseInterface** interface and both of them are implemented in the **InterfaceImplementer** class.

Namespaces

The concept of namespace is not new to Visual Basic. A namespace is a kind of wrapper around one or more structural elements that make the elements unique. Whether or not you explicitly declare a namespace in the Visual Basic source file, the compiler adds a default namespace. Namespaces have public access and is not modifiable.

A namespace in Visual Basic has the following properties:

- It organizes large code projects
- The operator delimits it

Namespaces in Visual Basic is of two categories: user-defined and system-defined. The user-defined namespaces are the namespaces you create in the code, and the system-defined namespaces are the one which are already added in your code when you create a new application. All the code you write exists within an implied namespace that exists for the current context of the code. In Visual Basic, the **Imports** statement is used to tell the compiler which namespaces you want to use in the program.

Let's learn to create a user-defined namespace and also how you can pass the reference of the namespace in your program.

Note

> Namespaces are always public; therefore, the declaration of a namespace cannot include any access modifier.

Creating Namespaces

When you create a large number of classes, it is helpful to divide them into their own namespaces to help organize things. You can use namespaces to group the type so that you can use it multiple times and also to avoid the conflict with the names that are already declared. When you create a namespace, you must use the **Namespace** keyword followed by its name.

Let's create an application named **MyNamespace** to learn to create namespace, by performing the following steps:

1. *Repeat* the steps 1 and 2 discussed while creating the **ConstructorApp** application.
2. In the **New Project** dialog box, *select* **Visual Basic** in the **Project types** pane and the **Class Library** option in the **Templates** pane.
3. *Enter* the name **MyNamespace** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** box.
4. *Click* the **OK** button. The **New Project** dialog box closes and the **MyNamespace** application is created.
5. In the **Class1.vb** file, *add* the code given in Listing 3.18:

UserClass.

6. *Click* **Build→Build Solution** on the menu bar to build the application.

Referencing Namespaces

You can also use a user-defined namespace in your application. To use a user-defined namespace in your application, you must add reference of that namespace to your application.

Let's create an application named **MyApplication** to learn to add reference of **MyNamespace** namespace, by performing the following steps:

1. *Repeat* the steps 1-3 discussed while creating the **ConstructorApp** application.
2. *Enter* the name **MyApplication** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **New Project** dialog box closes and the **MyApplication** application is created.
4. *Right-click* the project name in the **Solution Explorer**, as shown in Fig.VB-3.22:

5. *Select* **Add Reference** from the context menu (Fig.VB-3.22). The **Add Reference** dialog box opens, as shown in Fig.VB-3.23:

Fig.VB-3.23

6. In the **Add Reference** dialog box, *click* the **Browse** tab to locate the **MyNamespace.dll** file (Fig.VB-3.23).
7. *Locate* the **MyNamespace.dll** file and select it, (Fig.VB-3.23).
8. *Click* the **OK** button to add the reference to the **MyApplication** project (Fig.VB-3.23). The **MyNamespace.dll** file is added to the **MyApplication** project.
9. In the **Module1.vb** file of the **MyApplication** project, *add* the code given in Listing 3.19:

Listing 3.19: Adding Reference

In Listing 3.19, a user-defined namespace, **MyNamespace.MyNamespace**, is added to the **MyApplication** project.

10. *Press* the **F5** key to run the application. The output of the code given in the Listing 3.19 appears, as shown in Fig.VB-3.24:

Fig.VB-3.24

Summary

In this chapter, you learned about:

- Defining classes and objects in a Visual Basic application
- Hiding irrelevant information in a class using encapsulation
- Implementing reusability of code through inheritance
- Implementing the same procedure in multiple ways through polymorphism
- Creating data types that store small amount of data with the help of structures
- Working with properties
- Specifying the members that must be supplied by classes with the help of interfaces
- Organizing Visual Basic code with the help of namespaces

Introduction

Windows forms are building blocks of an application. It is a graphical user interface for building Windows client applications that use Common Language Runtime (CLR). It is the name given to the Graphical User Interface (GUI) that constitutes a part of Microsoft .NET Framework.

Apart from Console applications, which run directly from the Windows Command Prompt, other applications designed on .NET Framework are built using forms.

There are forms for Windows applications as well as Web applications. Windows forms possess advanced graphical and visual representations and are therefore, highly customizable. These forms are also highly programmable, that is, their behavior can be customized using any .NET compliant language. Windows forms acts as a container for .NET controls. They offer various smart client features, such as tabbed navigation (wherein a user can navigate to all the controls in a cycle), ordering of tabs, handling mouse events, and so on.

In this chapter, you learn to create a Windows Forms application. We also learn to perform various operations on Windows forms, such as adding controls on the form, setting the title of the form, setting tab order of controls, enabling and disabling controls, and so on. Further, you also learn to create multiple forms, message boxes, input boxes, and dialog boxes.

Let us start by creating the Windows Forms application in Visual Basic 2008.

Creating a Visual Basic 2008 Windows Forms Application

You can create a variety of applications using Windows forms. Various controls can be added to make the applications more functional and user-friendly. You can add controls in the Windows forms from the **Toolbox** by dragging and dropping the controls or by double-clicking controls.

Let's perform the following steps to create a new Windows Forms application:

1. *Click* **File→New→Project**. The **New Project** dialog box appears, as shown in Fig.VB-4.1:

Fig.VB-4.1

2. *Select* the **Visual Basic→Windows** option in the **Project types** pane (Fig.VB-4.1).
3. *Select* the **Windows Forms Application** template in the **Templates** pane (Fig.VB-4.1).
4. *Enter* a name for your application in the **Name** text box. In our case, we have entered **FirstProject** (Fig.VB-4.1) .

5. *Enter* the complete path of the folder where you want to save your application in the **Location** box (Fig.VB-4.1).
6. *Click* the **OK** button (Fig.VB-4.1). The **FirstProject** application opens, as shown in Fig.VB-4.2:

Fig.VB-4.2

In this section, we learned to create a Windows Forms application in Visual Basic 2008. In the next section, we discuss some basic operations that you can perform on Windows forms in Visual Basic 2008.

Performing Some Basic Operations on Windows Forms

You can perform basic operations on Windows forms. The different basic operations you can perform on the Windows forms are as follows:

- Setting the title of a form
- Adding controls to a form
- Handling the Click event of a button
- Docking and anchoring controls
- Setting tab order of controls
- Enabling and disabling controls

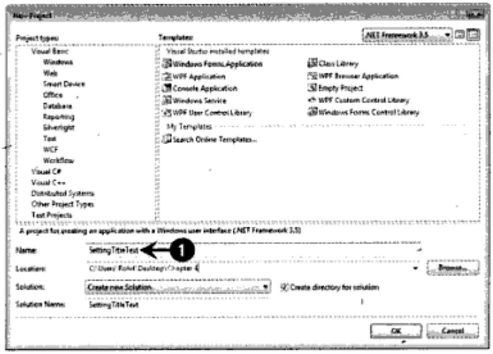
Let's start with learning how to set the title of a form.

Setting the Title of a Form

The text in the title bar of a form can be set at either design time or run time. At design time, it can be set by changing the **Text** property of the form from the **Properties** window. In the following steps, we set the title at run time.

Let's perform the following steps to set the title of the form:

1. *Create* a new Windows Forms application by entering **SettingTitleText** as the name of the application, as shown in Fig.VB-4.3:



A project for creating an application with a Windows user interface (.NET Framework 3.5)

Name: SettingTitleText

Location: C:\Users\Rohit\Desktop\Chapter 4

Solution: Create new Solution

☑ Create directory for solution

Solution Name: SettingTitleText

Fig.VB-4.3

2. *Open* the Code Editor by double-clicking the form and then *add* the highlighted code given in Listing 4.1 in the **Form1_Load** event handler:

Listing 4.1: Code for Setting the Title Text of the Form

In Listing 4.1, you have set the title text for the **Form1** as **Hello**, using the **Text** property of **Form1**.

3. *Run* the application by *pressing* **F5** key on the keyboard. As a result, the title text of the form changes to Hello, as shown in Fig.VB-4.4:

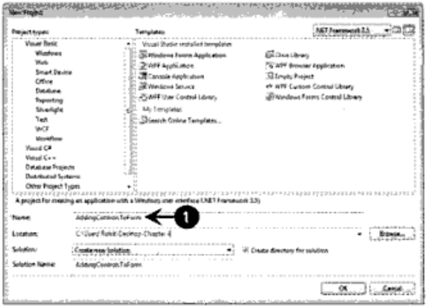
Fig.VB-4.4

Adding Controls to a Form

In Visual Basic 2008, we can add controls such as Label, Button, TextBox, and so on to our application from the **Toolbox** to facilitate user interaction.

Let's perform the following steps to add **Button** control to a form:

1. *Create* a new Windows Forms application by entering **AddingControlsToForm** as the name of the application, as shown in Fig.VB-4.5:



A project for creating an application with a Windows user interface (.NET Framework 3.5)

Name: AddingControlsToForm

Location: C:\Users\Rohit\Desktop\Chapter 4

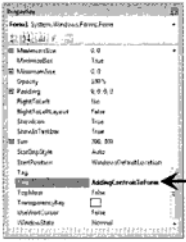
Solution: Create new solution

☒ Create directory for solution

Solution Name: AddingControlsToForm

Fig.VB-4.5

2. *Set* the **Text** property of **Form1** to **AddingControlsToForm**, as shown in Fig.VB-4.6:

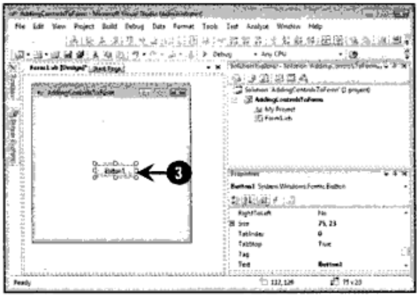


Form1 System.Windows.Forms.Form

MaximumSize	0, 0
MinimizeBox	True
MinimumSize	0, 0
Opacity	100%
Padding	0, 0, 0, 0
RightToLeft	No
RightToLeftLayout	False
ShowIcon	True
ShowInTaskbar	True
Size	300, 300
SizeGripStyle	Auto
StartPosition	WindowsDefaultLocation
Tag	
Text	AddingControlsToForm
TopMost	False
TransparencyKey	□
UseWaitCursor	False
WindowState	Normal

Fig.VB-4.6

3. *Drag* and *drop* the **Button** control on **Form1** from the **Toolbox**, as shown in Fig.VB-4.7:



Button1 System.Windows.Forms.Button

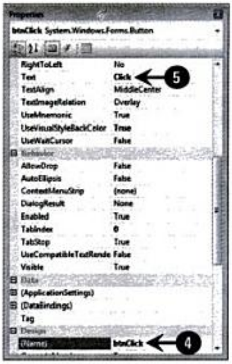
RightToLeft	No
Size	75, 23
TabIndex	0
TabStop	True
Tag	
Text	Button1

Fig.VB-4.7

Note

You can also add controls on the form by double-clicking the controls in the Toolbox.

4. *Select* the **Button** control and change its **Name** property to **btnClick** in the **Properties** window, as shown in Fig.VB-4.8:



btnClick System.Windows.Forms.Button

RightToLeft	No
Text	Click
TextAlign	MiddleCenter
TextImageRelation	Overlay
UseMnemonic	True
UseVisualStyleBackColor	True
UseWaitCursor	False
⊟ Behavior	
AllowDrop	False
AutoEllipsis	False
ContextMenuStrip	(none)
DialogResult	None
Enabled	True
TabIndex	0
TabStop	True
UseCompatibleTextRende	False
Visible	True
⊟ Data	
⊞ (ApplicationSettings)	
⊞ (DataBindings)	
Tag	
⊟ Design	
(Name)	btnClick

Fig.VB-4.8

5. *Set* the **Text** property of the button to **Click**, as shown in Fig.VB-4.8. You observe that the text written on the button is changed to the value you gave for the **Text** property, as shown in Fig.VB-4.9:

Fig.VB-4.9

Note

Similarly, you can add other controls such as Label, TextBox, RichTextBox, ListBox and so on to your applications.

Handling the Click Event of a Button

Visual Basic 2008 is an event-driven language, a language in which the flow of the program is controlled by user actions, such as by pressing a key from the keyboard or by clicking some control with a mouse. Clicking a button and entering some text into a text box are basic examples of events. We can write code to handle an event in the code designer.

Let's first now learn how we can handle the **Click** event of a button by performing the following steps:

1. *Create* a new Windows Forms application with the name **HandlingEvents.**
2. *Change* the **Text** property of **Form1** to **HandlingEvents**.
3. *Drag* a **Button** control and a **TextBox** control from the **Toolbox** and *drop* on the **HandlingEvents** form.
4. *Set* the **Text** property of the button to **ClickMe** from the Properties windows.
5. To handle the **Click** event, *double-click* the button and *add* the code in the Code Editor, as given in Listing 4.2:

Listing 4.2: Code for Handling the Click Event of the Button Control

6. *Run* the application by *pressing* the **F5** key and *click* the **ClickMe** button (Fig.VB-4.10). As a result, the text, **Welcome to Visual Basic 2008**, appears in the text box, as shown in Fig.VB-4.10:

Fig.VB-4.10

Docking and Anchoring Controls

Docking refers to attaching a control to either an edge (top, right, bottom, or left) or the client area of the parent control. On the other hand, in anchoring you specify the distance that each edge of your control maintains from the edges of the parent control. You can use docking and anchoring to align and arrange the controls present on a form.

Let's perform the following steps to dock and anchor a **Button** control in the application:

1. *Open* the **AddingControlsToForm** application.
2. For docking the **Button** control along the top edge of the form, *select* the button in the design view and *set* its **Dock** property from the **Properties** window, as shown in Fig.VB-4.11:

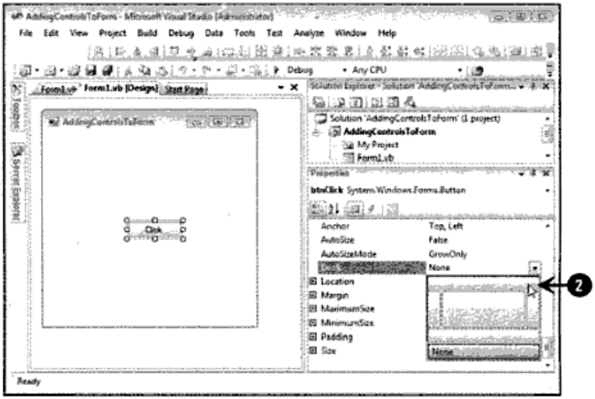
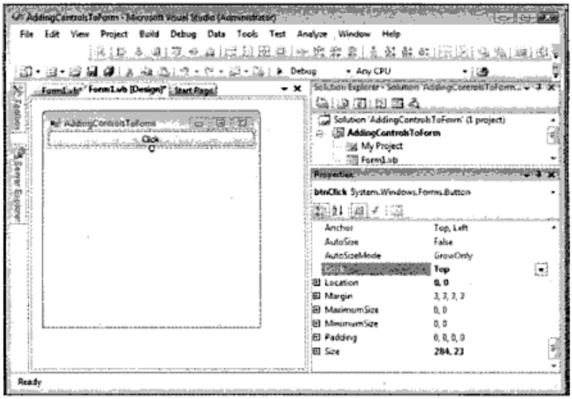

AddingControlsToForm - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Data Tools Test Analyze Window Help
Debug
Any CPU
Form1.vb
Form1.vb [Design]
Start Page
Toolbox
Server Explorer
AddingControlsToForm
Click
Solution Explorer - Solution 'AddingControlsToForm...
Solution 'AddingControlsToForm' (1 project)
AddingControlsToForm
My Project
Form1.vb
Properties
btnClick System.Windows.Forms.Button
Anchor
AutoSize
AutoSizeMode
Location
Margin
MaximumSize
MinimumSize
Padding
Size
Top, Left
False
GrowOnly
None
None
2
Ready

Fig.VB-4.11

The **Button** control is docked to the top edge of the form, as shown in Fig.VB-4.12:


AddingControlsToForm - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Data Tools Test Analyze Window Help
Debug
Any CPU
Form1.vb*
Form1.vb [Design]*
Start Page
Toolbox
Server Explorer
AddingControlsToForm
Click
Solution Explorer - Solution 'AddingControlsToForm...
Solution 'AddingControlsToForm' (1 project)
AddingControlsToForm
My Project
Form1.vb
Properties
btnClick System.Windows.Forms.Button
Ready

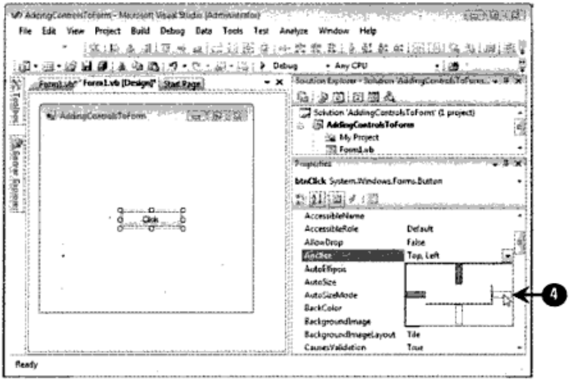
Anchor	Top, Left
AutoSize	False
AutoSizeMode	GrowOnly
	Top
Location	**0, 0**
Margin	3, 3, 3, 3
MaximumSize	0, 0
MinimumSize	0, 0
Padding	0, 0, 0, 0
Size	**284, 23**

Fig.VB-4.12

We have seen how a control is docked. Now, we learn to anchor a control. To do so, first we must change the **Dock** property of the **Button** control to **None**.

3. *Select* the **Button** control in the design view and *set* its **Dock** property to **None** from the Properties window.

4. To anchor the **Button** control along all the edges of the form, *select* the **Anchor** property of the **Button** control in the **Properties** window and *click* all the edges which are seen as blank, as shown in Fig.VB-4.13:


AddingControlsToForm - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Data Tools Test Analyze Window Help
Debug
Any CPU
Form1.vb* Form1.vb [Design]* Start Page
Toolbox
Solution Explorer
AddingControlsToForm
Click
Solution Explorer - Solution 'AddingControlsToForm'...
Solution 'AddingControlsToForm' (1 project)
AddingControlsToForm
My Project
Form1.vb
Properties
Ready
4

btnClick System.Windows.Forms.Button

AccessibleName	
AccessibleRole	Default
AllowDrop	False
Anchor	Top, Left
AutoEllipsis	
AutoSize	
AutoSizeMode	
BackColor	
BackgroundImage	
BackgroundImageLayout	Tile
CausesValidation	True

Fig.VB-4.13

As a result, the button is anchored to all the edges of the form.

5. To see how anchoring works, *resize* the form from the bottom right corner. You see the **Button** control also enlarging in the same proportion, as shown in Fig.VB-4.14:

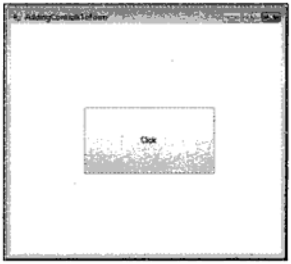


Fig.VB-4.14

Setting the Tab Order of Controls

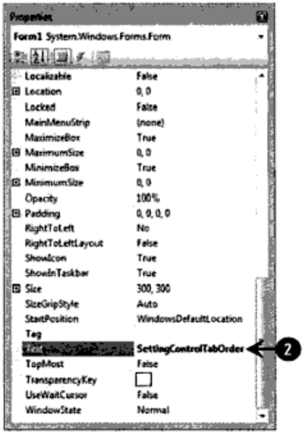
Controls on a Windows Forms can be made accessible in an appropriate sequence by setting their **TabIndex** property.

Let's perform the following steps to set the tab order of controls:

1. *Create* a new Windows Forms application by entering **SettingControlTabOrder** as the name of the application, as shown in the Fig.VB-4.15:

Fig.VB-4.15

2. *Set* the **Text** property of **Form1** as **SettingControlTabOrder** from the **Properties** window, as shown in the Fig.VB-4.16:



Form1 System.Windows.Forms.Form

Localizable	False
⊞ Location	0, 0
Locked	False
MainMenuStrip	(none)
MaximizeBox	True
⊞ MaximumSize	0, 0
MinimizeBox	True
⊞ MinimumSize	0, 0
Opacity	100%
⊞ Padding	0, 0, 0, 0
RightToLeft	No
RightToLeftLayout	False
ShowIcon	True
ShowInTaskbar	True
⊞ Size	300, 300
SizeGripStyle	Auto
StartPosition	WindowsDefaultLocation
Tag	
Text	SettingControlTabOrder
TopMost	False
TransparencyKey	□
UseWaitCursor	False
WindowState	Normal

Fig.VB-4.16

3. *Drag* two **Button** controls and two **TextBox** controls from the **Toolbox** and *drop* them on the form, as shown in Fig.VB-4.17:

Fig.VB-4.17

4. *Set* the **Name** properties of the two buttons as **btnName** and **btnAge** respectively. *Set* their **Text** properties as **Add Name** and **Add Age**, respectively, as shown in the Fig.VB-4.18.
5. *Set* the **Name** properties of the two text boxes as **txtName** and **txtAge,** respectively.
6. *Select* the first **Button** control and *set* its **TabIndex** property value to **0** (zero), as shown in Fig.VB-4.18:

Fig.VB-4.18

Note

7. *Apply* Step 6 for all other controls by making each côntrol's **TabIndex** property value one more than its previous control's **TabIndex** property value.

Note

We can navigate through controls using the TAB key (on the keyboard) after setting the TabIndex property of the controls present on a form.

8. *Run* the application by *pressing* the **F5** key. The focus is on the first **Button** control, as shown in Fig.VB-4.19:

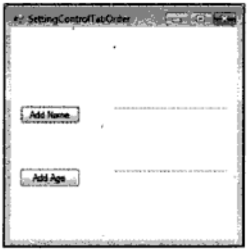


Fig.VB-4.19

9. *Press* the **TAB** key. The cursor moves to the second control, containing the **TabIndex** value as **1**. In this case, the first **TextBox** is and you see it focused, as shown in Fig.VB-4.20:

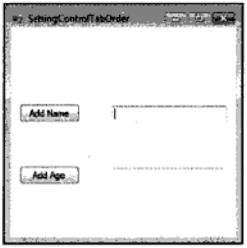


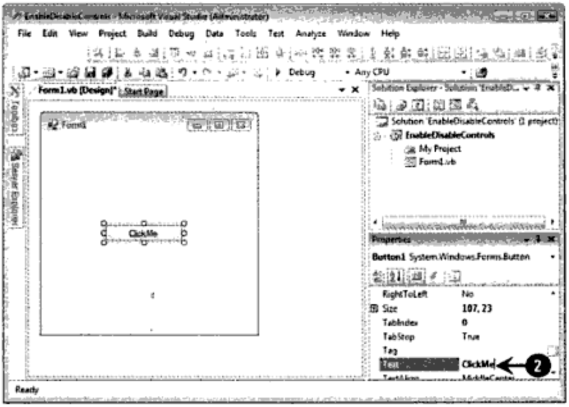
Fig.VB-4.20

Enabling and Disabling Controls

You can set the various properties of the controls in Windows forms to perform manipulations in your applications. For example, you can set the properties to enable or disable the controls using the **Properties** window. To disable a control, set the **Enabled** property to **False** in the **Properties** window of the control. Similarly, to enable a disabled control, set the **Enabled** property to **True**. By default, the **Enabled** property of a control is set as **True**.

Let's perform the following steps to learn how to enable and disable controls:

1. *Create* a new Windows Forms application with the name **EnableDisableControls.**
2. *Drag* a button control from the **Toolbox** and *drop* on the **Form**. *Change* the caption of the **Button** control to **ClickMe**, as shown in the Fig.VB-4.21:


EnableDisableControls - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Data Tools Test Analyze Window Help
Debug
Any CPU
Form1.vb [Design]*
Start Page
Toolbox
Server Explorer
Form1
ClickMe
Solution Explorer - Solution 'EnableD...
Solution 'EnableDisableControls' (1 project)
EnableDisableControls
My Project
Form1.vb
Properties
Button1 System.Windows.Forms.Button
2
Ready

RightToLeft No
Size 107, 23
TabIndex 0
TabStop True
Tag
Text ClickMe
TextAlign MiddleCenter

Fig.VB-4.21

3. *Double-click* the **Button** control and *add* the following highlighted code to the **Code Editor**, as given in Listing 4.3:

Listing 4.3: Code for Disabling the Button Control

4. *Run* the application by pressing the **F5** key on the keyboard. The output of the application is shown in Fig.VB-4.22, which displays the **ClickMe** button enabled, by default:

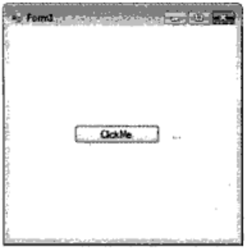


Fig.VB-4.22

5. Now, *click* the **ClickMe** button, as shown in the Fig.VB-4.23:

Fig.VB-4.23

In this section, we learned to perform some basic operations on Windows forms in Visual Basic 2008. In the next section, you learn to work with multiple forms.

Working with Multiple Forms

Suppose you have designed your application, with an introductory form to welcome the user, a data entry form to get data from the user, and a summary form to display the data analysis results. Instantly, it occurs to you that not all VB 2008 Windows projects are organized into modules, classes, and forms. How then does the code in one form reach the code in another form? In other words, how can the codes in the analysis module read what the user has entered in the data entry form?

This problem is easily solved if you have some idea about working with multiple forms. It means that how the data entered by the user at one form is read at another form.

Let's perform the following steps to see how to work with multiple forms:

1. *Create* a new Windows Forms application, named **WorkingWithMultipleForms**. The application contains just one Windows form, **Form1**.
2. To *add* a second form to this project, *select* **Add Windows form** from **Project** menu to open the **Add New Item** dialog box.
3. *Select* **Windows Form** option in the **Templates** pane, as shown in Fig.VB-4.24:

Fig.VB-4.24

4. Now, **n2** is added to the p

Fig.VB-4.25

5. Now, *add* a text box **TextBox1**, to **Form2**. When the user clicks on a button in **Form1**, we read the text entered in the text box on **Form2** and display it in a text box in **Form1**.

If you want to display the second form as soon as **Form1** is loaded, then you must put the code for displaying the **second** form inside the **Load** event handler of **Form1**.

6. *Add* the following code to show the second form using the **Show** method, in the Code Editor:

7. Now *add* a **Button** control and a **TextBox** control to **Form1**.
8. *Set* the **Text** property of the button to **Read Text**. When the user clicks this button, we want the text in the text box of **Form2** to be read and to display it in the text box of **Form1**.
9. Now, *double-click* the **Read Text** button in **Form1** and *add* the following highlighted code inside the **Button1 Click** event handler in the Code Editor:

10. *Run* the application by *pressing* **F5** key. You can see **Form1** and **Form2** (Fig.VB-4.26).
11. *Type* some text in the text box in **Form2,** as shown in Fig.VB-4.26.
12. *Click* the **Read Text** button on **Form1** (Fig.VB-4.26).

As a result, **Form2** disappears and the text appears in the text box in **Form1**, as shown in Fig.VB-4.27:

Setting the Startup Form

A startup form is the first form displayed when an application having two or more forms loads. By default, Form1 is the startup form. We make a form as a startup form when we want to display it at the beginning of our application. To learn how to set the startup form of an application, we open the project we created in the previous section of this chapter and make the second form of this project as the startup form.

Let's perform the following steps to set the startup form:

1. *Open* the **WorkingWithMultipleForms** project.
2. *Select* **Project→WorkingWithMultipleForms Properties** on the menu bar, as shown in Fig.VB-4.28:

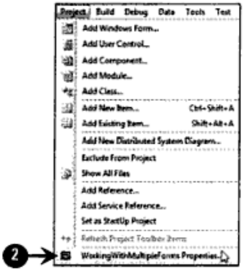
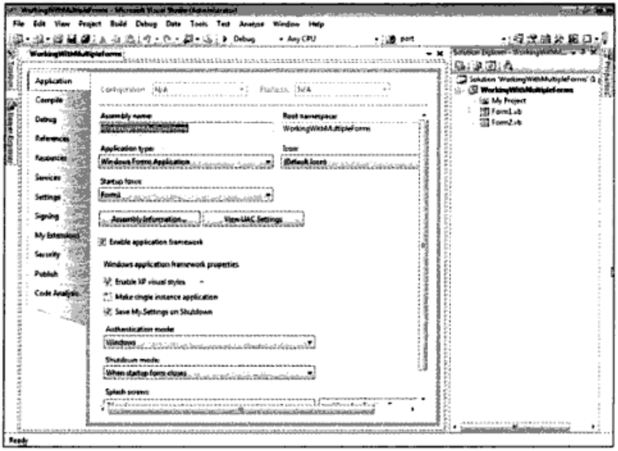


Fig.VB-4.28

Note

The **Project Designer** appears, as shown in Fig.VB-4.29:



Configuration: N/A

Platform: N/A

Assembly name:

Root namespace: WorkingWithMultipleForms

Application type: Windows Forms Application

Icon: (Default Icon)

Startup form: Form1

Assembly Information... View UAC Settings

☑ Enable application framework

Windows application framework properties

☑ Enable XP visual styles

☐ Make single instance application

☑ Save My.Settings on Shutdown

Authentication mode: Windows

Shutdown mode: When startup form closes

Splash screen:

Fig.VB-4.29

In the **Project Designer**, **Form1** is selected in the **Startup form** option. Now, if you run this project, **Form1** appears.

3. In the **Project Designer**, *click* the combo box under the **Startup form** option and *select* **Form2,** as shown in Fig.VB-4.30:

Fig.VB-4.30

Note

> Similarly, to make a form appear as the first form in the application, select any existing form in the **Project Designer.**

4. Now, *press* **F5** to run this project. **Form2** appears, as shown in Fig.VB-4.31:

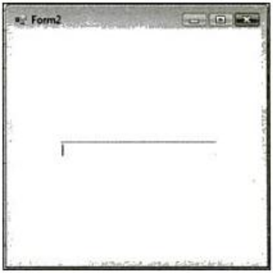


Fig.VB-4.31

As a result, **Form2** is now set as the startup form. Now, whenever the application is run, **Form2** is displayed in place of **Form1**, which is by default the startup form.

In this section, we learned to work with multiple forms in Visual Basic 2008. In the next section, you learn to create message boxes in Visual Basic 2008.

Creating Message Boxes

To display a message to the user at run time, we can create a message box. A message box can be created through the code with the help of the **MsgBox** function. The syntax of the **MsgBox** function is below:

A of the arguments passed to the **MsgBox** function is as follows:

- **Prompt:** A string expression that is displayed as the message in the message box. The maximum length for this expression is 1,024 characters.
- **Buttons:** A set of values specifying the number and type of buttons to display, the icon style to use, the identity of the default button, and the modality of the message box. If you do not specify the Buttons argument in the function, the function takes the default value zero as the value for the Buttons argument.
- **Title:** The string expression displayed in the title bar of the dialog box. If you do not specify the Title, the name of the application is placed in the title bar.

Table 4.1 shows the possible constants that the **Buttons** argument of the **MsgBox** function can take:

Table 4.1: Possible Constants for the Buttons Argument of the MsgBox Function

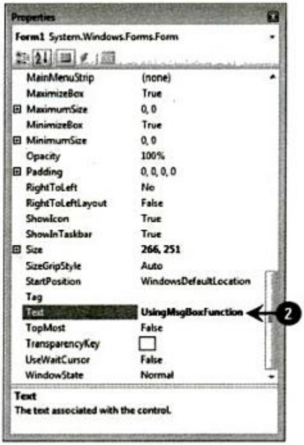
Value	Description
524288	Ensures that the text is right-aligned

The **MsgBox** function returns a value from the **MsgBoxResult** enumeration. The values in the **MsgBoxResult** enumeration indicate which button in the message box the user has clicked. These values may include one of the following:

- OK
- Cancel
- Abort
- Retry
- Ignore
- Yes
- No

Let's perform the following steps to learn to create a message box:

1. *Create* a new Windows Forms application with the name **UsingMsgBoxFunctionApp**.
2. *Set* the **Text** property of **Form1** as **UsingMsgBoxFunction**, as shown in the Fig.VB-4.32:



Properties

Form1 System.Windows.Forms.Form

MainMenuStrip	(none)
MaximizeBox	True
⊞ MaximumSize	0, 0
MinimizeBox	True
⊞ MinimumSize	0, 0
Opacity	100%
⊞ Padding	0, 0, 0, 0
RightToLeft	No
RightToLeftLayout	False
ShowIcon	True
ShowInTaskbar	True
⊞ Size	266, 251
SizeGripStyle	Auto
StartPosition	WindowsDefaultLocation
Tag	
Text	UsingMsgBoxFunction
TopMost	False
TransparencyKey	□
UseWaitCursor	False
WindowState	Normal

Text
The text associated with the control.

Fig.VB-4.32

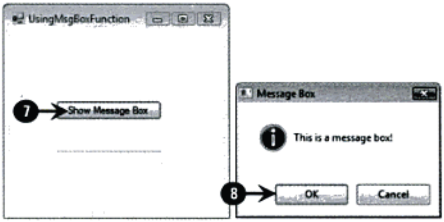
3. *Drag* a **Button** control and a **TextBox** control from the **Toolbox** and *drop* on your form.
4. *Set* the **Name** and **Text** properties of **Button** control to **btnShowMessageBox** and **Show Message Box,** respectively.

5. *Set* the **Name** property of the **TextBox** to **txtMessage.**
6. *Double-click* the **Button** on **Form1** in the design view and *add* the following Code to the Code Editor, as given in Listing 4.4:

Listing 4.4: Code for Using the MsgBox Function

In the above code, we create a message box and store the result of the selection made by the user on the message box in an Integer variable named `Result`. If the user clicks the OK button at run time, the text 'You have clicked OK', is added to the text box present on the Form.

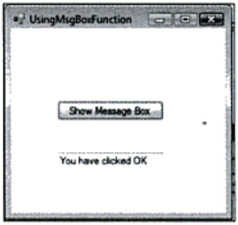
7. *Run* the application by pressing the **F5** key and *click* the **Show Message Box** button (Fig.VB-4.33). As a result, a message box appears displaying a message, as shown in Fig.VB-4.33:



This is a message box!

Fig.VB-4.33

8. *Click* the **OK** button in the message box as shown in Fig.VB-4.33. The message box closes and the text, 'You have clicked OK', is added to the text box present on the form as shown in Fig.VB-4.34:



You have clicked OK

Fig.VB-4.34

In this section, we learned to create a message box, which provides a form-like user interface to display a message to the user at run time. Besides displaying messages, however, sometimes we also need to accept some input from the user at run time. This can be done using an input box. We discuss input boxes in the next section.

Creating Input Boxes

An Input box is also a form-like user interface similar to a message box. However, unlike a message box, an input box accepts the input from the user. An input box can be created with the help of the **InputBox** function. The syntax of the **InputBox** function is given below:

A description of the arguments passed to the **InputBox** function is given below:

- **Prompt:** A string expression that is displayed as the message in the input box. The maximum length for this expression is 1,024 characters (depending on the width of the characters used).
- **Title:** A string expression displayed in the title bar of the input box. Note that if you omit the title, the application name is placed in the title bar.
- **DefaultResponse:** A string expression displayed in the text box as the default response, in case no other input is provided. Note that if you omit the **DefaultResponse**, the displayed text box is empty.
- **XPos:** The distance in pixels of the left edge of the dialog box from the left edge of the screen. Note that if you omit **XPos**, the dialog box is centered horizontally.
- **YPos:** The distance in pixels of the upper edge of the dialog box from the top of the screen. Note that if you omit **YPos**, the dialog box is positioned vertically about one-third of the way down the screen.

Let's perform the following steps to learn how we can create an input box:

1. *Create* a new Windows Forms application with the name **UsingInputBoxFunctionApp**.
2. *Set* the **Text** property of **Form1** as **UsingInputBoxFunction,** as shown in Fig.VB-4.35:

Fig.VB-4.35

3. *Drag* a **Button** control and a **TextBox** control from the Toolbox and *drop* on your Form.
4. *Set* the **Name** and **Text** properties of the **Button** control as **btnShowInputBox** and **Show Input Box,** respectively.

5. *Set* the **Name** property of the **TextBox** control to **txtMessage**.
6. *Double-click* the **Button** control on **Form1** in the desigr view and *add* the following code in the Code Editor, as given in Listing 4.5:

Fig.VB-4.36:

Fig.VB-4.36

As a result, an input box appears, as shown in Fig.VB-4.36.

8. *Enter* some text in the text box provided on the input box, as shown in Fig.VB-4.37:

Fig.VB-4.37

9. *Click* the **OK** button in the input box, as shown in Fig.VB-4.37. The input box closes and the text box present on the form shows the text that you typed, as shown in Fig.VB-4.38:

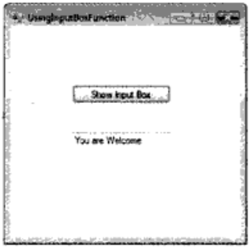


Fig.VB-4.38

In this section, we learned to create input boxes in Visual Basic 2008. In the next section, you learn to create dialog box in Visual Basic 2008.

Creating Dialog Boxes

Dialog box is a movable window that is displayed on the screen when you select a specific menu option. They are called as dialog box because they facilitate a dialog between the user and the computer by either informing the user of something or requesting the user for some sort of input, or both. However, pre-defined dialog boxes do not always fulfill our requirements. So, there are certain ways to create customized dialog boxes. In this section, we create an application, in which the user enters some text in a dialog box, and the entered text is read when the dialog box closes.

Let's perform the following steps to learn how we can create a dialog box:

1. *Create* a new Windows Forms application with the name **CreatingDialogBoxesApp.**
2. *Change* the **Text** property of **Form1** to **CreatingDialogBoxes.**
3. *Add* a **Button** control and a **TextBox** control on **Form1** from the Toolbox.
4. *Change* the **Name** property of the **Button** and the **TextBox** controls as **btnShowDialogBox** and **txtEnteredText,** respectively. Also *change* the **Text** property of the button to **Show Dialog Box.**
5. Now *add* one more form to your application by *clicking* **Project→Add Windows Form.** Further, *change* the **Text** property of **Form2** to **Enter Your Text.**
6. *Add* one **Label** control, one **TextBox** control, and two **Button** controls on **Form2** from the Toolbox.
7. *Change* the **Name** property of the **Label**, the **TextBox** and the two **Button** controls as **lblEnterText, txtEnterText**, and **btnOK** and **btnCancel,** respectively. Also *change* the **Text** properties of the **Label** and the two **Button** controls as **Enter Your Text** and **OK** and **Cancel,** respectively. After adding controls to **Form2**, it appears as shown in Fig.VB-4.39:

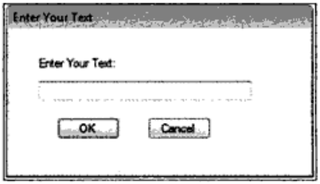
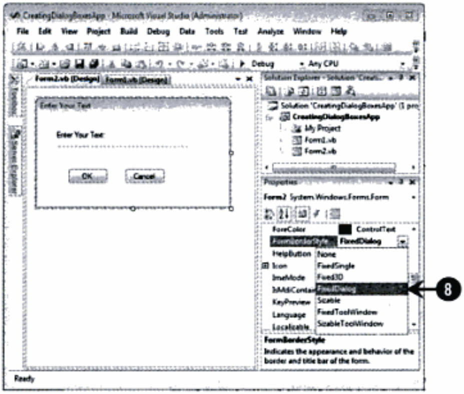


Fig.VB-4.39

8. *Set* the **FormBorderStyle** property of **Form2** to **FixedDialog,** as shown in Fig.VB-4.40:



CreatingDialogBoxesApp - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Data Tools Test Analyze Window Help
Debug
Any CPU
Form2.vb [Design]
Form1.vb [Design]
Toolbox
Server Explorer
Enter Your Text
Enter Your Text
OK
Cancel
Solution Explorer - Solution 'Creati...
Solution 'CreatingDialogBoxesApp' (1 pro
CreatingDialogBoxesApp
My Project
Form1.vb
Form2.vb
Properties
8
Ready

Form2 System.Windows.Forms.Form

ForeColor ControlText
FormBorderStyle FixedDialog
HelpButton
Icon
ImeMode
IsMdiContainer
KeyPreview
Language
Localizable

None
FixedSingle
Fixed3D
FixedDialog
Sizable
FixedToolWindow
SizableToolWindow

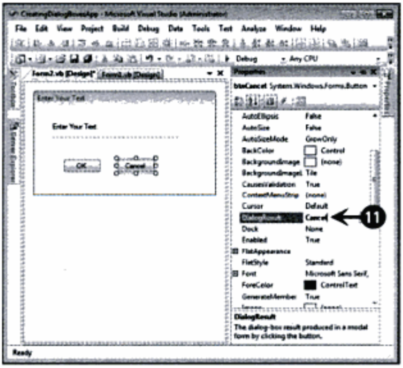
FormBorderStyle
Indicates the appearance and behavior of the border and title bar of the form.

Fig.VB-4.40

9. *Set* the **ControlBox** property of **Form2** to **False** to remove the control box (the minimize, maximize, and close buttons appear at the upper right corner of the form).

10. *Set* the **ShowInTaskbar** property of **Form2** to **False** meaning that when this dialog box appears, it does not display an icon in the Windows taskbar.

11. *Set* the **DialogResult** property of the **OK** button to **OK** and also the same property of the **Cancel** button to **Cancel,** as shown in Fig.VB-4.41:



CreatingDialogBoxesApp - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Data Tools Test Analyze Window Help
Debug
Any CPU
Form2.vb [Design]*
Form1.vb [Design]
Toolbox
Server Explorer
Enter Your Text
Enter Your Text
OK
Cancel
Properties
11
Ready

btnCancel System.Windows.Forms.Button

AutoEllipsis False
AutoSize False
AutoSizeMode GrowOnly
BackColor Control
BackgroundImage (none)
BackgroundImageLayout Tile
CausesValidation True
ContextMenuStrip (none)
Cursor Default
DialogResult Cancel
Dock None
Enabled True
FlatAppearance
FlatStyle Standard
Font Microsoft Sans Serif,
ForeColor ControlText
GenerateMember True

DialogResult
The dialog-box result produced in a modal form by clicking the button.

Fig.VB-4.41

The **DialogResult** property returns a value from the **DialogResult** enumeration, when the dialog box is closed. So, you can determine which Button the user has clicked. This property can have one of the following values:

- OK
- Cancel
- Abort
- Retry
- Ignore
- Yes
- No
- None

Displaying and Reading Data from Dialog Boxes

For displaying the dialog box, the user clicks on the **Show Dialog Box** button. Here, we use the **ShowDialog** method, instead of the **Show** method. This is due to the reason that the **ShowDialog** method returns a **DialogResult** value which indicates what button the user has clicked. Now, if the user clicks the **OK** Button, the text entered by the user in the text box of the dialog box, is displayed in the text box of **Form1**.

Let's perform the following steps to display and read the data from dialog boxes:

12. *Double-click* the **Show Dialog Box** button on **Form1** in the design view and *add* the highlighted code snippet given in Listing 4.6 in the Code Editor:

Listing 4.6: Using the ShowDialog Method

13. *Double-click* the **OK** button on **Form2** in the design view and *add* the highlighted code snippet given in Listing 4.7 in the Code Editor:

Listing 4.7: Code for Closing the Form

14. Now, *run* the application by pressing **F5** key on the keyboard. As a result, **Form1** loads.
15. *Click* the **Show Dialog Box** button on **Form1**, the dialog box, which is **Form2**, opens, as shown in Fig.VB-4.42:

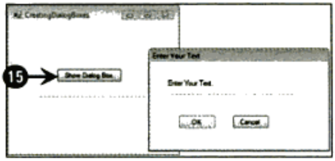
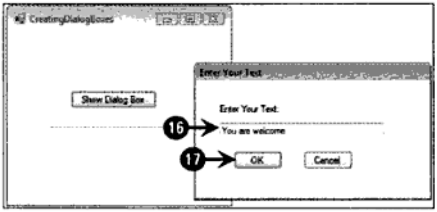


Fig.VB-4.42

16. *Enter* any text in the text box of the dialog box, as shown in Fig.VB-4.43.
17. *Click* the **OK** button, as shown in Fig.VB-4.43:

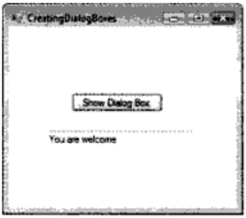


Enter Your Text:

You are welcome

Fig.VB-4.43

The dialog box closes and the text you entered in the text box of the dialog box is shown in the text box of **Form1**, as shown in Fig VB-4.44:



You are welcome

Fig.VB-4.44

Summary

In this chapter, you learned about:

- Creating a Windows Forms application
- Performing basic operations on Windows Forms, such as setting the title of a form, adding controls to a form, and handling events
- Working with multiple forms and setting the startup form
- Creating of message boxes, input boxes, and dialog boxes

Introduction

A control is an object that can be placed on the form to facilitate the user interaction with the applications. Windows controls are the controls used for creating Windows Forms applications. These controls are available in the Toolbox of Visual Studio 2008 Integrated Development Environment (IDE). In this chapter, we are going to cover some popular Windows controls, such as Label, TextBox, Button, RadioButton, CheckBox, ComboBox, ListBox, PictureBox, Timers, ProgressBar, and two grouping controls— GroupBox and Panel. These two are called Grouping controls because they allow the developer to group other controls while developing a Window Forms application. Let's now discuss different Windows controls one by one with the help of applications.

The Label Control

The **Label** control is used for displaying static text that you do not want to be edited by the user or a banner containing some message for the user. It can also be used to display dynamic text. Dynamic text is the text that changes after the occurrence of an event in an application. For setting a label's appearance, you need to set properties of the **Label** control.

Here, we learn how to perform the following tasks with the help of an application:

- Formatting the text in labels
- Handling the **Click** event of labels

Formatting the Text in Labels

You can format the text in a label by setting the `Font` property of the label using the **Properties** window. To format the text in the labels, perform the following steps:

1. *Create* a new Windows Forms application and name it **LabelsExample**.
2. *Drag* and *drop* the **Label** control on the Windows form from the **Toolbox**, as shown in the Fig.VB-5.1:

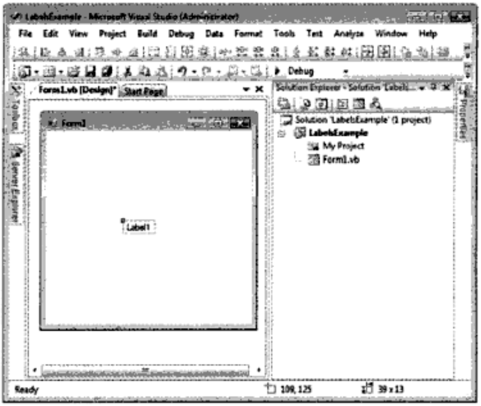

LabelsExample - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Data Format Tools Test Analyze Window Help
Debug
Form1.vb [Design]* Start Page
Toolbox
Server Explorer
Form1
Label1
Solution Explorer - Solution 'Labels...
Solution 'LabelsExample' (1 project)
LabelsExample
My Project
Form1.vb
Properties
Ready
109, 125
39 x 13

Fig.VB-5.1

3. *Set* the **BackColor** property to **LightPink** from the **Web** tab, as shown in Fig.VB-5.2:

Fig.VB-5.2

4. *Set* the **Text** property to **Hello World** and **TextAlign** property to **MiddleCenter**, as shown in Fig.VB-5.2.
5. *Select* the **Label** control on the form and *click* the ellipsis button (...) in front of the **Font** property in the **Properties** window. This opens the **Font** dialog box, as shown in Fig.VB-5.3:

Fig.VB-5.3

6. In the **Font** dialog box, *select* a font, font style, and font size, and then *click* the **OK** button (Fig.VB-5.3). In our case, we have selected **Arial Black** as the font, **Bold** as the font style, and **8** as the font size. You can see the design view of this application in Fig.VB-5.4:

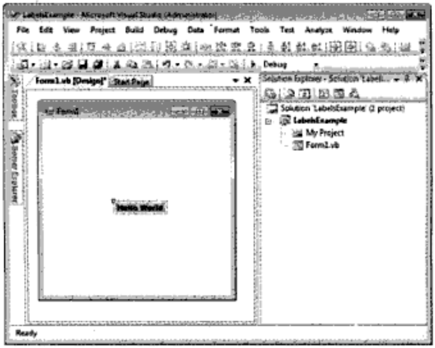

LabelsExample - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Data Format Tools Test Analyze Window Help
Debug
Toolbox
Server Explorer
Form1.vb [Design]*
Start Page
Form1
Hello World
Solution Explorer - Solution 'Label...
Solution 'LabelsExample' (1 project)
LabelsExample
My Project
Form1.vb
Ready

Fig.VB-5.4

Fig.VB-5.4 displays the appearance of the Windows form along with the **Label** control after setting the properties.

Handling the Click Event of Label

You can perform an action at runtime with the **Label** control by handling its **Click** event in the code. To handle the **Click** event of **Label** control, perform the following steps:

1. *Select* the **Label** control and *double-click* it in the form.

3. *Run* the application by pressing the **F5** key on the keyboard and then *click* the label.

A message box displaying the message. You have clicked the label control, appears, as shown in Fig.VB-5.5:

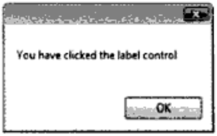

You have clicked the label control

Fig.VB-5.5

Here, we have discussed about Label control. Let's now learn about TextBox control in Visual Basic 2008.

The TextBox Control

The **TextBox** control is a Windows Forms control that lets you enter text on a Windows form at runtime. **TextBox** controls are mostly used when the user requires simple text area where one or few lines of text can be displayed. By default, a **TextBox** control accepts only a single line of text. However, you can make a **TextBox** control to accept multiple lines of text, and disable text editing, by setting different properties of the **TextBox** control.

Let's perform these steps to create a Windows Forms application showing how to use **TextBox** control:

1. *Create* a new Windows Forms application and name it **TextBoxExample.**
2. *Drag* and *drop* a **TextBox** control from the **Toolbox** on **Form1**, as shown in Fig.VB-5.6:

Fig.VB-5.6

3. *Double-click* the **TextBox** control for opening the Code Editor and *add* the following code, as given in Listing 5.1:

Listing 5.1: Code for Using TextBox

The explanation of the lines of code of the preceding code snippet is as follows:

```
    TextBox1.ForeColor = Color.Blue
```

In the above code snippet, the **ForeColor** property of a control is used to define the fore color of the text. In our example, we have set it to blue color.

In the above code snippet, we are checking the length of the text entered in the textbox through the conditional **If** statement. As soon as the length of the text exceeds 6 characters, it displays a message box showing the warning message and sets the **ReadOnly** property of the textbox to **True**, which means that now you are not able to write in the textbox.

4. *Run* the application by pressing the **F5** key on the keyboard and as a result a form gets displayed, as shown in Fig.VB-5.7:

Fig.VB-5.7

5. *Enter* some text in the textbox in the form. You will notice that the color of the text in the text box becomes blue, as shown in Fig.VB-5.8:

6. *Enter* some more text in the text box. As soon as the character count exceeds 6, a message box appears, as shown in Fig.VB-5.9:

Fig.VB-5.9

7. *Click* the **OK** button of the message box, as shown in Fig.VB-5.9.

The textbox now becomes read-only, as shown in Fig.VB-5.10:

Fig.VB-5.10

Note

> Whitespaces are also treated as characters.

Here, we have discussed about TextBox control. Let's now move on to learn about Button control in Visual Basic 2008.

The Button Control

The **Button** control is one of the most basic Windows Forms controls. Almost every Windows Forms application has at least one **Button** control associated with it. The **Button** control lets you generate a **Click** event. Here, we learn how to perform the following tasks using the **Button** control:

- ❑ Formatting the text in buttons
- ❑ Setting the background and foreground colors of buttons

Formatting the Text in Buttons

You can format the text displaying on a **Button** control by setting its **Font** property. This property can be set either using the **Properties** window or using the Code Editor.

Let's perform these steps to create a Windows Forms application showing how to format the text in buttons using the **Properties** window:

1. *Create* a new Windows Forms application and name it **ButtonExample**.
2. *Drag* and *drop* the **Button** control on the Windows form from the **Toolbox,** as shown in Fig.VB-5.11:

Fig.VB-5.11

3. *Select* the **Button** control on the form and then *set* its **Text** property to **.NET Programming** in the **Properties** window.
4. *Set* the **Font** property of the **Button** control using the **Font** dialog box. Here, we choose the **Font Style** as **Bold Italic** and the **Font** as **Arial Black** and then *click* the **OK** button, as shown in Fig.VB-5.12:

You can see the design view of this application in Fig.VB-5.13:

Fig.VB-5.13

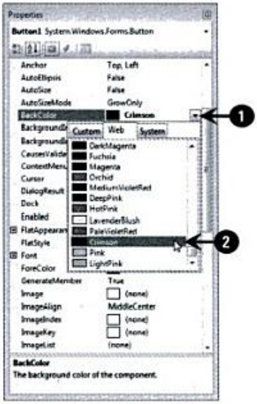
Fig.VB-5.13 shows the **Button** control with the name, **.NET Programming.**

Setting the Background and Foreground Colors of Buttons

You can add background and foreground color of a button by setting its **BackColor** and **Forecolor** properties, respectively. Let's perform the following steps to set the background and foreground color of buttons:

1. *Select* the **Button** control and then in the **Properties** window, *click* the down arrow in front of the **BackColor** property.

A drop-down list appears with three tabs, named **Custom**, **Web**, and **System**, as shown in Fig.VB-5.14:



Properties

Button1 System.Windows.Forms.Button

Anchor Top, Left
AutoEllipsis False
AutoSize False
AutoSizeMode GrowOnly
BackColor Crimson
BackgroundI
BackgroundI
CausesValida
ContextMenu
Cursor
DialogResult
Dock
Enabled
FlatAppearan
FlatStyle
Font
ForeColor
GenerateMember True
Image (none)
ImageAlign MiddleCenter
ImageIndex (none)
ImageKey (none)
ImageList (none)

BackColor
The background color of the component.

Fig.VB- 5.14

2. *Select* the **Web** tab and then *select* a color from the color palette displayed under this tab. Here, we set the color to **Crimson**, as shown in Fig.VB-5.14.

The background color of the button changes to crimson, as shown in Fig.VB-5.15:

Fig.VB-5.15

3. *Select* a color from the color palette of the **ForeColor** property. Here, we *set* the color to **DarkBlue**, as shown in Fig.VB-5.16:

Fig-VB- 5.16

The fore color of the **Button** control changes, as shown in Fig.VB-5.17

Fig.VB-5.17

4. *Run* the application by *pressing* the **F5** key on the keyboard. As a result, the output is displayed as shown in Fig.VB-5.18:

Fig.VB-5.18

Here, we have discussed about the **Button** control. Let's now move on to learn about **RadioButton** control in Visual Basic 2008.

The RadioButton Control

A **RadioButton** control, also known as an option button, is used to select one option from a set of options. Radio buttons always work in groups. This means whenever you select one radio button from a group of radio buttons, the other radio buttons in the group automatically get deselected. A radio button can display a text, or an image, or both.

Let's perform these steps to create a Windows Forms application showing how to use **RadioButton** control:

1. *Create* a new Windows Forms application and name it **RadioButtonExample.**
2. *Drag* and *drop* two **Label** controls and three **RadioButton** controls from the **Toolbox** on **Form1**, as shown in Fig.VB-5.19:

Fig.VB-5.19

3. *Set* the **Name** and **Text** properties of the controls, as shown in Table 5.1:

4. *Set* the **Font Style** of **Label** with text, **Select the Font Color**, as **Bold**, as shown in Fig.VB-5.20.
5. *Set* the size of the **Label** control with text **WELCOME** as **14** and make its **Font Style** as **Bold**, as shown in Fig.5.20:

Fig.VB-5.20

6. In design view, *double-click* the **Red** radio button and *add* the following code snippet in the Code Editor:

7. In the design view, *double-click* the **Blue** radio button and *add* the following code snippet in the Code Editor:

8. In the design view, *double-click* the **Green** radio button and *add* the following code snippet in the Code Editor:

9. *Press* the **F5** key on the keyboard to run the application. You will notice that the word **WELCOME** appears in red color because the radio button beside the **Red** option is selected, by default, as shown in Fig.VB-5.21:

Fig.VB-5.21

10. Now, *select* the radio button beside the **Blue** option, the word **WELCOME** is displayed in blue color, as shown in Fig.VB-5.22:

Fig.VB-5.22

11. Similarly, *select* the radio button beside the **Green** option, the word **WELCOME** is displayed in green color, as shown in the Fig.VB-5.23:

Fig.VB-5.23

Here, we have discussed abo⸤t **RadioButton** control. Now, let's learn about CheckBox control in Visual Basic 2008.

The CheckBox Control

A **CheckBox** control accepts a value of either **True** or **False**. To select a **CheckBox** control, you need to just click it. To clear, again click it. When you select the **CheckBox**, it holds the **True** value and when you clear the **CheckBox**, it holds the **False** value. A **CheckBox** control can display image or corresponding text associated with it. It can also display both at the same time.

Let's perform these steps to create a Windows Forms application showing how to use **CheckBox** control:

1. *Create* a new Windows Forms application and name it **CheckBoxesExample**.
2. *Drag* and *drop* two **Label** controls and two **CheckBox** controls from the **Toolbox** on **Form1**, as shown in Fig.VB-5.24:

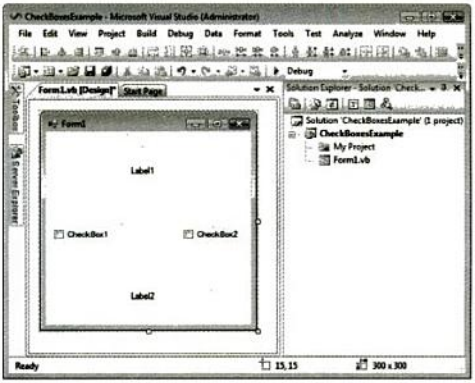

CheckBoxesExample - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Data Format Tools Test Analyze Window Help
Debug
Form1.vb [Design]* Start Page
Toolbox
Server Explorer
Form1
Label1
CheckBox1
CheckBox2
Label2
Solution Explorer - Solution 'Check...
Solution 'CheckBoxesExample' (1 project)
CheckBoxesExample
My Project
Form1.vb
Ready
15,15
300 x 300

Fig.VB-5.24

3. *Set* the **Name** and **Text** properties of these controls, as shown in Table 5.2:

Table 5.2: Name and Text Properties of Controls on Form1

Control Name	Name	Text
Label1	lblTitle	Select the Font Style
Label2		
CheckBox1		
CheckBox2		

4. *Set* the **Font** property of **lblTitle** by setting its **Font style** to **Bold** and **Size** to **10**, as shown in Fig.VB-5.25.
5. Also, *set* the **Font** property of **lblWelcome** by setting its **Size** to **12,** as shown in the Fig.VB-5.25:

Fig.VB-5.25

6. In the design view, *double-click* the **Bold** check box and *add* the following code snippet in the Code Editor:

8. *Run* the application by pressing the **F5** key on the keyboard. The output window appears, as shown in Fig.VB-5.26:

Fig.VB-5.26

9. *Select* the **CheckBox** with text **Bold.** As a result, the label with text **WELCOME** changes its **Font Style** to **Bold**, as shown in Fig.VB-5.27:

10. *Select* the **CheckBox** with text **Italic**. As a result, the label changes its **Font Style** to **Italic**, as shown in Fig.VB-5.28:

Here, we have discussed about CheckBox control in Visual Basic 2008. Now, let's learn about **ComboBox** control in Visual Basic 2008.

The ComboBox Control

ComboBox is a Windows control that is widely used for selecting an option from a list just as the **ListBox** control; however, unlike the **ListBox** control, you can also enter your own text in the **ComboBox** control. The **ComboBox** control is used to display data in a drop-down list. When the user has selected an option, the drop-down list contained in the **ComboBox** automatically collapses. A user can choose only a single option from the list of items. You can also add or remove any item from this drop-down list. Each item in a **ComboBox** control is recognized by its position in the drop-down list, which is known as an index.

Let's perform these steps to create a Windows Forms application showing how to use **ComboBox** control:

1. *Create* a new Windows Forms application and name it **ComboBoxExample**.
2. Now, *drag* and *drop* two **ComboBox** controls on the Windows form from the **Toolbox**. Also, *drag* and *drop* three **Label** controls for displaying texts as banner.

Table 5.3 lists the description of the different controls that are used in the **ComboBoxExample**:

Label3	**For displaying the caption text for the ComboBox2**

3. *Select* **Label2** and then from the **Properties** window, *set* the **Text** property as **Select Date.** Similarly, *set* the **Text** property for **Label3** as **Select Month** and for the **Label1** as **Check Your SunSign**.
4. *Click* the arrow at the top right corner of the **ComboBox1** (Fig.VB-5.29). As a result, the Smart Tag of the combo box appears, as shown in Fig.VB-5.29.
5. *Click* the **Edit Items** option on the Smart Tag of **ComboBox1**, as shown in Fig.VB-5.29:

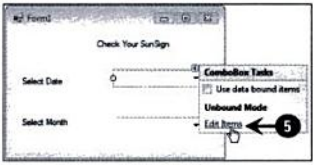
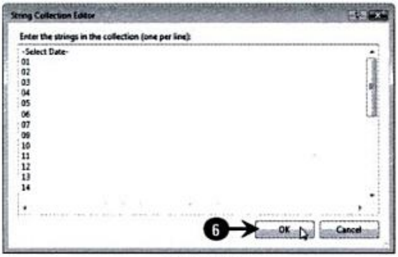


Fig.VB-5.29

As a result, the **String Collection Editor** dialog box appears, as shown in Fig.VB-5.30:



```
-Select Date-
01
02
03
04
05
06
07
09
10
11
12
13
14
```

Fig.VB-5.30

6. In the **String Collection Editor** dialog box, *add* date values in the **Enter the strings in the collection (one per line)** list box and then *click* the **OK** button (Fig.VB-5.30).
7. Similarly, *add* month values in the **String Collection Editor** dialog box for **ComboBox2** and *click* the **OK** button, as shown in Fig.VB-5.31:

Fig.VB-5.31

We assume that the user first select the date in the **ComboBox1** and then select the month option from the **ComboBox2**. To display a message when the user selects the month, use the **SelectedIndexChanged** event of the **ComboBox2**.

8. *Double-Click* **Combobox2** and *add* the code under the **SelectedIndexChanged** event handler in the Code Editor, as shown in Listing 5.2:

Listing 5.2: Code for Using ComboBox

9. *Press* the **F5** key on the keyboard to *run* the application. As a result, the output gets displayed, as shown in Fig.VB-5.32:

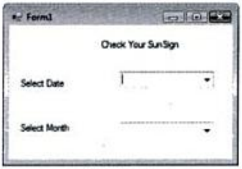
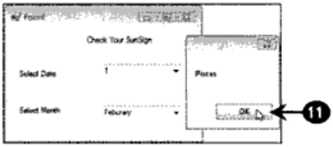


Fig.VB-5.32

10. *Choose* a date from the combo box beside the **Select Date** option and the month from the combo box beside the **Select Month** option. As a result, the sun sign according to the selected date and month appears in a message box, as shown in Fig.VB-5.33:



Check Your SunSign

Select Date 1

Select Month Feburary

Pisces

Fig.VB-5.33

11. *Click* the **OK** button to close the message box (Fig.VB-5.33).

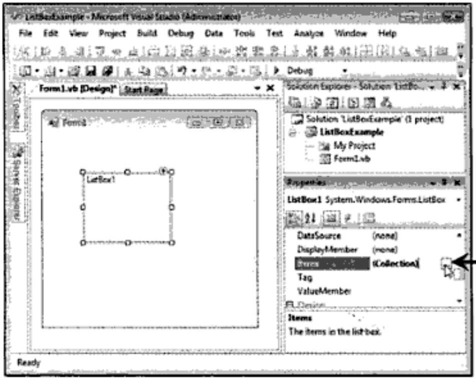
Here, we have discussed about **ComboBox** control. Now, let's learn about **ListBox** control in Visual Basic 2008.

The ListBox Control

ListBox is a standard Windows control that is used to display the text as a list. The text can be displayed as a sorted or an unsorted list. You can add the text as an item into this collection for displaying it on a **ListBox** control. Similarly, you can remove an item for not displaying it on the **ListBox** control. Each item in a **ListBox** control is recognized by its position in the list, which is known as its index.

Let's perform these steps to create a Windows Forms application showing how to use **ListBox** control:

1. *Create* a new Windows Forms application and name it **ListBoxExample.**
2. *Drag* and *drop* a **ListBox** control from the **Toolbox** on **Form1,** as shown in Fig.VB-5.34:



ListBox1 System.Windows.Forms.ListBox

DataSource (none)

DisplayMember (none)

Items (Collection)

Tag

ValueMember

Items

The items in the list box.

ListBoxExample - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Data Tools Test Analyze Window Help
Debug
Form1.vb [Design]* Start Page
Toolbox
Server Explorer
Form1
ListBox1
Solution Explorer - Solution 'ListBo...
Solution 'ListBoxExample' (1 project)
ListBoxExample
My Project
Form1.vb
Properties
Ready
3

Fig.VB-5.34

3. *Click* the ellipsis button (...) in front of the **Items** property in the **Properties** window to set the **Items** property of the **ListBox** control (Fig.VB-5.34). The **String Collection Editor** dialog box appears, as shown in Fig.VB-5.35:

Fig.VB-5.35

4. *Enter* the items in the **String Collection Editor** dialog box and *click* the **OK** button, as shown in Fig.VB-5.35.

Now, you can see all the items in **ListBox** control, as shown in Fig.VB-5.36:

Fig.VB-5.36

5. *Double-click* the **ListBox** control and *add* the following code to the **SelectedIndexChanged** event in the Code Editor:

displaying a message in the message box.

6. *Press* the **F5** key on the keyboard to run the application, as shown in Fig.VB-5.37:

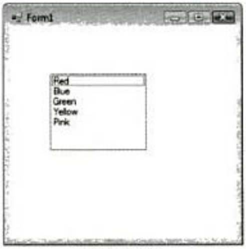
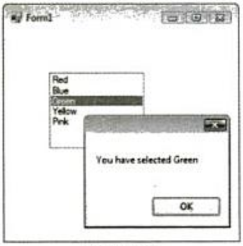


Fig.VB-5.37

7. *Select* any item from the listbox. A message box appears showing the corresponding message, as shown in Fig.VB-5.38:



You have selected Green

Fig.VB-5.38

Here, we have discussed about **ListBox** control in Visual Basic 2008. Now, let's learn about **GroupBox** control in Visual Basic 2008.

The GroupBox Control

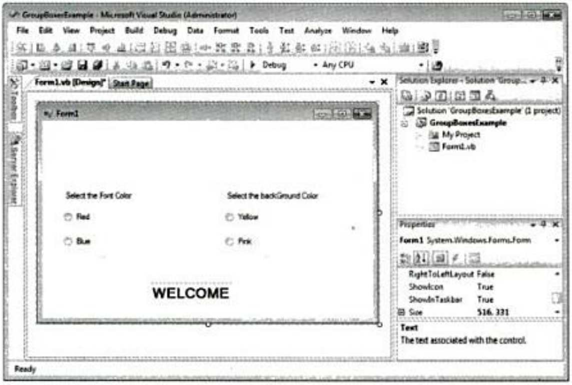
A **GroupBox** control is used to group together all the controls related to a task. Depending upon the various tasks to be performed, the form can be divided into various groups. Group boxes can or cannot have a caption.

Let's perform these steps to create a Windows Forms application showing how to use a **GroupBox** control:

1. *Create* a new Windows Forms application and name it **GroupBoxesExample.**
2. *Drag* and *drop* two **GroupBox** controls and one **TextBox** control on **Form1**. Also, add two **RadioButton** controls inside each **GroupBox.**
3. *Set* the **Name** and **Text** properties of these controls, as shown in Table 5.4:

4. *Set* the size of the **TextBox** control to **16** and the Font Style to **Bold**. Also, **set** the **TextAlign** property to **Center.**

After inserting controls in the form and setting their respective properties, the form looks, as shown in Fig.VB-5.39:


GroupBoxesExample - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Data Format Tools Test Analyze Window Help
Debug
Any CPU
Form1.vb [Design]*
Start Page
Form1
Select the Font Color
Red
Blue
Select the backGround Color
Yellow
Pink
WELCOME
Solution Explorer - Solution 'Group...
Solution 'GroupBoxesExample' (1 project)
GroupBoxesExample
My Project
Form1.vb
Toolbox
Server Explorer
Ready

Properties

Form1 System.Windows.Forms.Form

RightToLeftLayout	False
ShowIcon	True
ShowInTaskbar	True
Size	516, 331

Text
The text associated with the control.

Fig.VB-5.39

5. In the design view, *double-click* the **Red** radio button and *add* the following code snippet in the Code Editor:

8. *Double–click* the **Pink** radio bu the Code Editor:

9. *Press* the **F5** key on the to The output of this application is shown in Fig.VB-5.40:

Fig.VB-5.40

Here, you can see, we have used two **GroupBox** controls to group two sets of **RadioButton** controls. If you don't use the **GroupBox** controls then you can only select one radio button out of the four. However, as we need to set the font color as well as background color at the same time, so we have to group the **RadioButton** controls into two separate groups using two **GroupBox** controls.

Here, we have discussed about **GroupBox** control in Visual Basic 2008. Now, let's learn about **Panel** control in Visual Basic 2008.

The Panel Control

The **Panel** control is also used for putting different controls together into an identifiable group, same as the **GroupBox** control. The differences between the two controls are that the **Panel** control has scroll bars, but the **GroupBox** control does not have, and the **GroupBox** control displays a caption, but the **Panel** control does not.

Let's perform these steps to create a Windows Forms application showing how to use **Panel** control:

1. *Create* a new Windows Forms application and name it **PanelsExample**.
2. *Drag* and *drop* two **Panels** on **Form1** from the **Toolbox**.
3. Add three **RadioButton** controls to each of the two **Panel** controls. Also, add two **TextBox** controls in **Form1** from the **Toolbox**.
4. *Set* the **Text** properties of the **RadioButton** controls, as shown in Table 5.5:

After adding controls and setting their respective properties, the form looks as shown in Fig.VB-5.41:

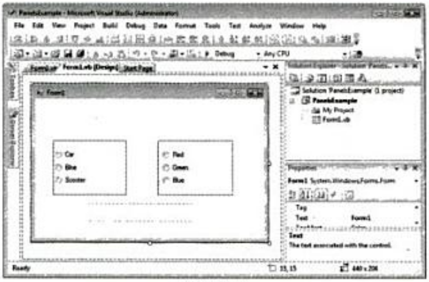
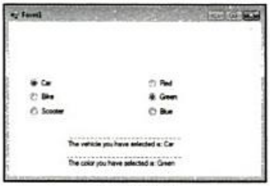

PanelExample - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Data Format Tools Test Analyze Window Help
Form1.vb [Design]
Start Page
Form1
Car
Bike
Scooter
Red
Green
Blue
Solution Explorer - Solution PanelExample
Solution 'PanelExample' (1 project)
PanelExample
My Project
Form1.vb
Properties
Form1 System.Windows.Forms.Form
Tag
Text
Form1
Text
The text associated with the control.
Ready
15, 15
440 x 206

Fig.VB-5.41

5. *Add* the code to the Code Editor for handling the **CheckChanged** events of the **RadioButton** controls, as shown in Listing 5.3:

6. *Run* the application by pressing the F5 key on the keyboard and then *select* a radio button in each panel. As a result, two messages are displayed in the two text boxes, as shown in Fig.VB-5.42:



The vehicle you have selected is: Car

The color you have selected is: Green

Fig.VB-5.42

Here, we have discussed about **Panel** control in Visual Basic 2008. Now, let's learn about **PictureBox** control in Visual Basic 2008.

The PictureBox Control

PictureBox is a Windows control that is used for displaying images in the Windows Forms applications. The image or picture can also be edited in this picturebox. For example, you can stretch the image you have added in the Windows form.

Let's perform these steps to create a Windows Forms application showing how to use **PictureBox** control:

1. *Create* a new Windows Forms application and name it **PictureBoxesExample.**
2. *Drag* and *drop* one **PictureBox** control on **Form1** from the **Toolbox.** Fig.VB-5.43 displays the form after adding the **PictureBox** control:

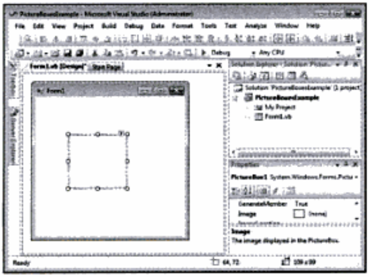
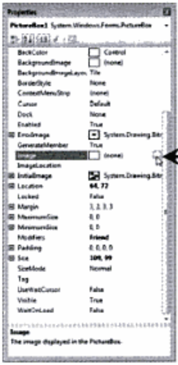

PictureBoxesExample - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Data Format Tools Test Analyze Window Help
Debug
Any CPU
Form1.vb [Design]* Start Page
Form1
Solution Explorer - Solution PictureBoxesExample
Solution 'PictureBoxesExample' (1 project)
PictureBoxesExample
My Project
Form1.vb
Properties
PictureBox1 System.Windows.Forms.PictureBox
GenerateMember True
Image (none)
Image
The image displayed in the PictureBox.
Ready
64, 72
109 x 99

Fig.VB-5.43

3. *Click* the ellipsis button (...) that is displayed in front of the **Image** property in the **Properties** window to set the **Image** property of the **PictureBox** control, as shown in the Fig.VB-5.44:



BackColor	□ Control
BackgroundImage	□ (none)
BackgroundImageLayout	Tile
BorderStyle	None
ContextMenuStrip	(none)
Cursor	Default
Dock	None
Enabled	True
ErrorImage	System.Drawing.Bitr
GenerateMember	True
Image	□ (none)
ImageLocation	
InitialImage	System.Drawing.Bitr
Location	64, 72
Locked	False
Margin	3, 3, 3, 3
MaximumSize	0, 0
MinimumSize	0, 0
Modifiers	Friend
Padding	0, 0, 0, 0
Size	109, 99
SizeMode	Normal
Tag	
UseWaitCursor	False
Visible	True
WaitOnLoad	False

Image
The image displayed in the PictureBox.

Fig.VB-5.44

As a result, the **Select Resource** dialog box is displayed, as shown in Fig.VB-5.45:

Fig.VB-5.45

4. *Click* the **Import** button to add the image in the **Select Resource** dialog box (Fig.VB-5.46).

Fig.VB-5.46

5. *Click* the **OK** button, as shown in Fig.VB-5.46.

As a result, the picture adds in the **PictureBox** control placed on the form, as shown in Fig.VB-5.47:

6. *Press* the **F5** key on the keyboard to run the application. The output form is shown in Fig.VB-5.48:

Here, we have discussed about **PictureBox** control in Visual Basic 2008. Now, let's learn about **ProgressBar** and **Timer** controls in Visual Basic 2008.

The ProgressBar and Timer Controls

The **ProgressBar** control is used to indicate the progress of any operation. It shows a bar that fills itself from left to right as the operation progresses. The **Minimum** and **Maximum** properties indicate the range of values representing the progress of a task. Usually, the **Minimum** property is set to value zero and **Maximum** property is set to a value that indicates the completion of a task. **Timers** are controls that are used to generate periodic events. These controls are called components and they do not appear in a window at run time. At design time, they appear in the component tray below the form in which they are added. You can add a **Timer** control to your form from the **Toolbox.**

Let's perform these steps to create a Windows Forms application showing how to use **Progressbar** and **Timer** controls:

1. *Create* a new Windows Forms application and name it **ProgressBarsTimersExample.**
2. *Drag* and *drop* one **PictureBox**, one **Button**, one **ProgressBar**, and one **Timer** control on **Form1** from the **Toolbox.**
3. *Set* the **Name** property of the Button to **btnAddImage** and its **Text** property to **Add Image.** Now, the form appears as shown in Fig.VB-5.49:

Fig.VB-5.49

4. In the design view, *double-click* the button having caption **Add Image** (Fig.VB-5.49) and *add* the following code snippet in the Code Editor:

5. Now, *add* the following code for the **Tick** event of **Timer1** in the Code Editor:

Note

6. *Press* the **F5** key to run the application to see the output as shown in Fig.VB-5.50:

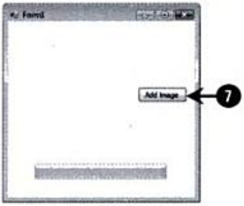


Fig.VB-5.50

7. Now, *click* the **Add Image** button (Fig.VB-5.50). This starts increasing the value of the **Value** property of the **ProgressBar** control, as shown in the Fig.VB-5.51:

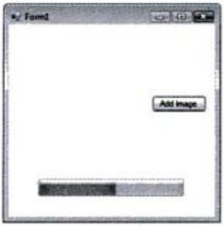


Fig.VB-5.51

After the value of the **Value** property of the **ProgressBar** control reaches 100, the image is loaded in the **PictureBox** control on the form, as shown in Fig.VB-5.52:

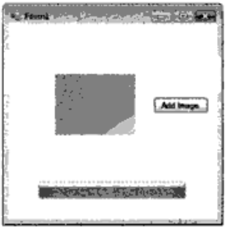


Fig.VB-5.52

Summary

In this chapter, we learned how to work with the following controls:

- □ The Label control
- □ The TextBox control
- □ The Button control
- □ The RadioButton control
- □ The CheckBox control
- □ The ComboBox control
- □ The ListBox control
- □ The GroupBox control
- □ The Panel control
- □ The PictureBox control
- □ The ProgressBar control
- □ The Timer control

Introduction

Apart from including Windows Forms to develop desktop applications, Visual Studio 2008 encompasses a new technology known as Windows Presentation Foundation (WPF), previously known as Avalon, which offers several features and functionalities to develop high-end desktop applications. WPF supports various media, such as text, images, audio, and video, and allows you to work with two-dimensional (2-D) as well as three-dimensional (3-D) graphics. Unlike Windows Forms, which require numerous technologies, such as Graphics Device Interface (GDI+), Windows Media Player, and DirectX application programming interfaces (APIs) to work with 2-D, 3-D, and multimedia, WPF offers a cohesive medium that inherently provides the functionality of these technologies. This implies that you do not need separate APIs in WPF to work with graphics, animations, and multimedia and hence it is easier and simpler to develop graphic-rich desktop applications by using WPF.

As stated, WPF allows you to develop desktop applications as its predecessor Windows Forms does; however, WPF has several improvements and enhancements over Windows Forms. Some of the enhancements for WPF include:

- **Improved application model:** The WPF application model is made up of several namespaces and classes that assist you in developing desktop applications.
- **Improved controls:** Several controls are common to both Windows Forms and WPF, for example, buttons, text boxes, and labels. However, some WPF controls, such as buttons, now support varied content types (for example, text, image, and list of items).
- **Support for data-validation and data-binding models and Language Integrated Query (LINQ):** WPF introduces certain classes, properties, and interfaces that allow you to bind data through the traditional data model and LINQ.
- **Enhanced support for 2-D and 3-D graphics, animations, and multimedia:** You can use a variety of 2-D shapes and 3-D classes that allow you to create and manipulate the 3-D content to develop attractive WPF applications. In WPF, you can animate controls, 2-D shapes, and text to develop graphic-rich applications. WPF also supports multimedia to incorporate audio, video, and images in the applications.

WPF was initially incorporated with .NET Framework 3.0 and is now included in .NET Framework 3.5. The version of WPF included in .NET Framework 3.5 is WPF 3.5.

In this chapter, you learn about the architecture of WPF 3.5 and the different types of WPF 3.5 applications. In addition, you learn about the WPF 3.5 Designer, the use of eXtensible Application Markup Language (XAML) in WPF, and some of the common controls of WPF 3.5. You also learn how to use resources and styles in WPF.

Now, let's start our discussion with the architecture of WPF 3.5.

The Architecture of WPF 3.5

Although WPF 3.5 is a part of .NET Framework 3.5, it has both managed and unmanaged components. Managed and unmanaged components of WPF 3.5 are shown in Fig.VB-6.1:

Fig.VB-6.1

As shown in Fig.VB-6.1, WPF 3.5 consists of the PresentationFramework, PresentationCore, WindowsBase, CLR, milcore, User32, DirectX, and Kernel components. The components that are shaded, that is, PresentationFramework, PresentationCore, WindowsBase, and milcore, are the essential components to work with WPF 3.5 applications. Now, let's briefly discuss these four components in detail.

The PresentationFramework Component

The PresentationFramework component refers to the **PresentationFramework.dll** in .NET Framework 3.5. This component offers classes to control the appearance and presentation of WPF 3.5 applications. For instance, controls, layout, and data binding in WPF applications are handled by the PresentationFramework component.

The PresentationCore Component

The PresentationCore component is implemented as the **PresentationCore.dll** assembly in .NET Framework 3.5. This component provides some of the most commonly used types and features of WPF 3.5. The classes and types offered by the PresentationCore component provide certain essential functionalities, such as properties and events in WPF 3.5. Note that the PresentationCore component does not offer types for the user interface (UI) of WPF 3.5 applications such as those offered by the PresentationFramework component.

The WindowsBase Component

The WindowsBase component is implemented as the **WindowsBase.dll** assembly in .NET Framework 3.5. This component provides with the fundamental WPF features, such as threading. Some of these features can be accessed and used outside the WPF domain.

The MIL or Milcore Component

WPF 3.5 also contains an unmanaged component called the Media Integration Layer (MIL or milcore). Milcore is an unmanaged component in WPF 3.5. Milcore interacts with DirectX and acts as a medium or interface between DirectX and CLR (and managed WPF components). Due to this interaction with DirectX, which is the component that processes all the design-related elements in WPF, milcore allows the display of 2-D as well as 3-D content in WPF 3.5 applications.

Now, let's know about the types of applications in WPF 3.5.

Types of WPF Applications

You can develop WPF 3.5 applications using Visual Studio 2008. There are broadly two types of WPF applications supported by Visual Studio 2008, which are standalone WPF applications and XAML browser applications (XBAPs). Let's learn about these two types of applications in detail beginning with standalone WPF applications.

Standalone WPF Applications

Standalone WPF applications are similar to Windows Forms applications, that is, you can install standalone applications on the users' computers and view them in the **Start** menu. Note that standalone WPF applications run with the privileges of the currently logged-in user. Let's now learn how to create a standalone WPF application.

Creating a Standalone WPF Application

In Visual Studio 2008, you can easily and quickly create a standalone WPF application. Visual Studio 2008 offers a project template for creating standalone WPF applications. Using the project template of Visual

Studio 2008 to create a standalone WPF application automatically adds the essential files in the application. Perform the following steps to create a standalone WPF application in Visual Studio 2008:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2008→Microsoft Visual Studio 2008** on your computer to open the Visual Studio 2008 IDE.
2. In the Visual Studio 2008 IDE, *click* **File→New→Project** from the menu bar to open the **New Project** dialog box. In the **New Project** dialog box, *select* **Visual Basic→Windows** in the **Project types** pane, as shown in Fig.VB-6.2:

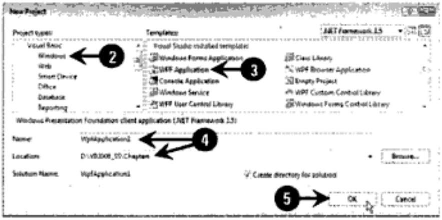


Fig.VB-6.2

As shown in Fig.VB-6.2, there is a **WPF Application** project template in the **Templates** pane of the **New Project** dialog box. This template creates the initial files and folders for creating standalone WPF applications.

3. In the **Templates** pane of the **New Project** dialog box, *select* **WPF Application**, as shown in Fig.VB-6.2.
4. *Specify* an appropriate name and location for the standalone WPF application in the **Name** text box and **Location** box, respectively, as shown in Fig.VB-6.2. In this case, we have specified **WpfApplication1** as the name and **VB2008_SS\Chapter6** folder in the **D** drive as the location.
5. *Click* the **OK** button in the **New Project** dialog box (Fig.VB-6.2). A new standalone WPF application named **WpfApplication1** is created, as shown in Fig.VB-6.3:

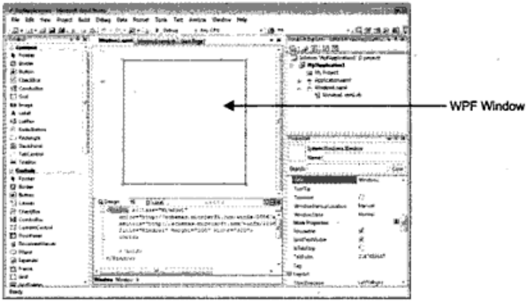


Fig.VB-6.3

As shown in Fig.VB-6.3, the designer interface of the standalone WPF applications is similar to that of Windows Forms applications. For instance, there are certain tools available in the Toolbox and a form-like structure called window at the center.

Visual Basic 2008

Visual Basic 2008

IN

Authored by:

Kogent Solutions Inc.

Published by:

dreamtech
PRESS

Limits of Liability/disclaimer of Warranty: The author and publisher have used their best efforts in preparing this book. The author make no representation or warranties with respect to the accuracy or completeness of the contents of this book, and specifically disclaim any implied warranties of merchantability or fitness of any particular purpose. There are no warranties which extend beyond the descriptions contained in this paragraph. No warranty may be created or extended by sales representatives or written sales materials. The accuracy and completeness of the information provided herein and the opinions stated herein are not guaranteed or warranted to produce any particulars results, and the advice and strategies contained herein may not be suitable for every individual. Neither Dreamtech Press nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

Trademarks: All brand names and product names used in this book are trademarks, registered trademarks, or trade names of their respective holders. Dreamtech Press is not associated with any product or vendor mentioned in this book.

ISBN: 978-81-7722-918-9

Edition: 2009

Printed at: Printman India, Patparganj, Delhi.

CONTENTS

Chapter 1 ■ Introducing .NET Framework 3.5 1

Introduction 2

Versions of .NET Framework 2

Benefits of .NET Framework 3

Architecture of .NET Framework 3.5 3

- Common Language Runtime 4
- .NET Framework Class Library 5
- Common Type System 5
- Common Language Specification 5
- Windows Forms 5
- ASP.NET and ASP.NET AJAX 6
- ADO.NET 6
- Windows Presentation Foundation 6
- Windows Communication Foundation 6
- Windows Workflow Foundation 6
- Windows CardSpace 6
- LINQ 7

Installing Visual Studio 2008 7

Opening Visual Studio 2008 11

Exploring Visual Studio 2008 IDE 13

- Menu Bar 13
- Toolbar 13
- Design Window 14
- Code Editor 14
- Server Explorer 15
- Solution Explorer 15
- Toolbox 16
- Properties Window 16
- Object Browser 17
- Class View Window 18

Summary 18

Chapter 2 ■ Getting Started with Visual Basic 2008 19

Introduction 20

Creating a Visual Basic 2008 Console Application 20

New Features of Visual Basic 2008 21

- Query Expressions 22
- Local Type Inference 22
- Object Initializers 22
- Extension Methods 23
- Lambda Expressions 23
- Anonymous Types 23
- Nullable Types 23
- Partial Methods 24
- Support for XML 24

Chapter 6 ■ Introducing Windows Presentation Foundation 131

Introduction 132
The Architecture of WPF 3.5 132
The PresentationFramework Component 133
The PresentationCore Component 133
The WindowsBase Component 133
The MIL or Milcore Component 133
Types of WPF Applications 133
Standalone WPF Applications 133
XAML Browser Applications 135
The WPF 3.5 Designer 137
The Design View 137
The XAML View 140
The Split View Bar 141
The Tag Navigator 142
XAML and WPF 143
XAML Elements and Attributes 143
Namespaces and XAML 145
Markup Extensions 145
Common Controls in WPF 3.5 146
Using the Grid Control 147
Using the Button Control 148
Using the TextBox Control 149
Using the PasswordBox Control 151
Using the TextBlock Control 153
Using the Border Control 154
Using the GridSplitter Control 155
Using the Canvas Control 157
Using the StackPanel Control 158
Resources and Styles 159
Using a Static Resource 160
Using a Dynamic Resource 161
Setting Style Through a Resource 163
Summary 164

Chapter 7 ■ Windows Workflow Foundation 165

Introduction 166
Workflow Principles 166
Workflows Coordinate Work Performed by People and Software 166
Workflows are Long Running and Stateful 166
Workflows are based on Extensible Models 166
Workflows are Transparent and Dynamic throughout their Lifecycle 166
Components of Windows Workflow Foundation 167
Workflow 167
Base Activity Library 168
Custom Activities 169
Host Process 170

Introduction

The .NET Framework is one of the most widely used software development environment in today's programming world. Before its introduction, programmers had to face a lot of difficulties to integrate the code written using different programming languages. This was due to the reason that each language used a different execution environment to execute the code written using that language. For example, code written using Visual Basic 6.0 requires a different execution environment for execution than that is required by code written using Visual C++. With the .NET Framework, Microsoft has provided programmers a single platform for developing applications using different programming languages, such as Visual Basic, Visual C#, and Visual C++.

The .NET Framework 3.5 is shipped with the Microsoft Visual Studio 2008. Microsoft Visual Studio is a set of development tools designed to help software developers to develop complex applications more quickly and easily. It provides the necessary environment in which developers can create and execute various types of applications, including Console applications, Windows Forms applications, WPF applications, Web applications, and Web services. It has improved the process of development and made it easier.

In this chapter, we learn about versions of .NET Framework, benefits of .NET Framework, and architecture of .NET Framework. We also learn how we can install Visual Studio 2008 and how we can open it. Finally, we take a look at Visual Studio 2008 IDE.

Let's first start by taking an overview of the different versions of .NET Framework.

Versions of .NET Framework

The .NET Framework has seen many upgrades since the release of its first version in 2002. All the versions of the .NET Framework that have been released till now are described as follows:

- **.NET Framework 1.0:** The .NET Framework 1.0 is the first version of the .NET Framework and was released by Microsoft on February 13, 2002. It is available for download in the form of a redistributable package as well as a Software Development Kit (SDK). It is also a part of Visual Studio .NET 2002, which is the first version of Visual Studio .NET.
- **.NET Framework 1.1:** The first major upgrade of the .NET Framework, the .NET Framework 1.1, was released on April 3, 2003. It is available for download in the form of a redistributable package as well as a Software Development Kit (SDK). It is also a part of Visual Studio .NET 2003, which is the second version of Visual Studio .NET. In contrast to the .NET Framework 1.0, the .NET Framework 1.1 has in-built support for mobile ASP.NET controls and Open Database Connectivity (ODBC) and Oracle databases. It also has support for Internet Protocol version 6 (IPv6).
- **.NET Framework 2.0:** The second major upgrade of the .NET Framework, the .NET Framework 2.0, was released on January 22, 2006. It is available for download in the form of a redistributable package as well as a Software Development Kit (SDK). It is also a part of Visual Studio 2005 and Microsoft SQL Server 2005. The .NET Framework 2.0 is the last version of the .NET Framework that has support of Windows 2000. The .NET Framework 2.0 has many changes and enhancements as compared to the .NET Framework 1.1. It has a number of Application Programming Interface (API) changes. It contains many new ASP.NET Web controls and data controls. It also contains new personalization features for ASP.NET, for example support for themes, skins, and WebParts.
- **.NET Framework 3.0:** The third major upgrade of the .NET Framework, the .NET Framework 3.0, was released on November 21, 2006. It contains a set of managed code APIs, which are an integral part of Windows Vista and Windows Server 2008. Managed code is the code that runs under Common Language Runtime (CLR). We discuss CLR in detail later in this chapter. The .NET Framework 3.0 uses the same version of CLR that was incorporated with .NET Framework 2.0. The .NET Framework 3.0 includes the following four new components:
  - Windows Presentation Foundation (WPF)
  - Windows Communication Foundation (WCF)

  - Windows Workflow Foundation (WF)
  - Windows CardSpace (WCS)

- **.NET Framework 3.5:** The fourth major upgrade of the .NET Framework, the .NET Framework 3.5, was released on November 19, 2007. Similar to the .NET Framework 3.0, the .NET Framework 3.5 also uses the same version of CLR. The .NET Framework 3.5 also installs the .NET Framework 2.0 SP1 and the .NET Framework 3.0 SP1, which includes methods and properties that are required for the .NET Framework 3.5 features, such as Language Integrated Query (LINQ). In addition to LINQ, the .NET Framework 3.5 includes many other new features, such as extension methods, lambda expressions, anonymous types, and built-in support for ASP.NET AJAX.

After having a quick overview of the versions of the .NET Framework, let's move on to discuss the benefits of the .NET Framework.

Benefits of .NET Framework

The .NET Framework offers many benefits to the programmers in developing applications. Some of these benefits are as follows:

- **Consistent programming model:** The .NET Framework provides a consistent object-oriented programming model across different languages. You can use this model to create programs for performing different tasks, such as connecting to and retrieving data from databases, and reading from and writing to files.
- **Language interoperability:** Language interoperability is a feature that enables code written in different languages to interact with each other. This allows reusability of code and improves the efficiency of the development process. For example, you can inherit a class created in C# in Visual Basic and vice-versa. The CLR has built-in support for language interoperability. However, there is no assurance that the code written using one programming language will work properly in programs developed using another programming language. Therefore, to ensure multi-language code interoperability, a set of language features and rules, called Common Language Specification (CLS), is defined. The components that follow these rules and expose only CLS features are said to be CLS-compliant.
- **Automatic management of resources:** When you create a .NET application, you do not need to manually free application resources, such as files, memory, network and database connections. The CLR automatically tracks the resource usage and saves you from the task of manual resource management.
- **Ease of deployment:** The .NET Framework makes the task of deployment easier. In most cases, to install an application, you need to copy the application along with its components, on the target computer. The .NET Framework provides easy deployment of applications by installing new applications or components that do not have an adverse effect on the existing applications. In .NET, applications are deployed in the form of assemblies; therefore, registry entries are not required to store information about components and applications. In addition, problems that used to arise due to different versions of an assembly are also overcome or eliminated in .NET Framework since assemblies also store information about different versions of the components used by an application.

Architecture of .NET Framework 3.5

The .NET Framework 2.0 and the .NET Framework 3.0, along with their service packs, form the foundation of the .NET Framework 3.5. In other words, the architecture of the .NET Framework 3.5, besides its new features and enhancements, includes components of the .NET Framework 2.0 and the .NET Framework 3.0. Architecture of the .NET Framework 3.5 is shown in Fig.VB-1.1:

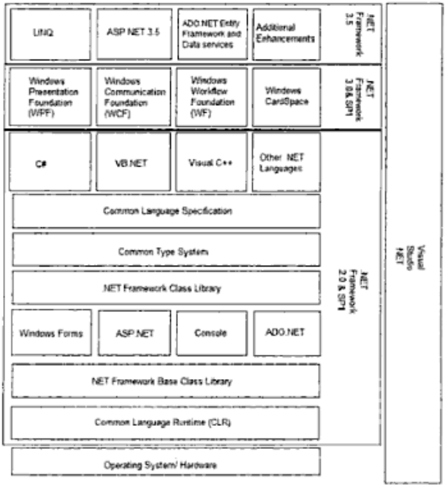

LINQ
ASP.NET 3.5
ADO.NET Entity Framework and Data services
Additional Enhancements
.NET Framework 3.5
Windows Presentation Foundation (WPF)
Windows Communication Foundation (WCF)
Windows Workflow Foundation (WF)
Windows CardSpace
.NET Framework 3.0 & SP1
C#
VB.NET
Visual C++
Other .NET Languages
Common Language Specification
Common Type System
.NET Framework Class Library
Windows Forms
ASP.NET
Console
ADO.NET
.NET Framework Base Class Library
Common Language Runtime (CLR)
.NET Framework 2.0 & SP1
Operating System/ Hardware
Visual Studio .NET

Fig.VB-1.1

As shown in Fig.VB-1.1, the main components of the .NET Framework 2.0 are CLR, .NET Framework Base Class Library, Windows Forms, ASP.NET, Common Type System (CTS), CLS, and .NET languages, such as C# and Visual Basic. The .NET Framework 3.0 adds four major components — Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF), Windows Workflow Foundation (WF), and Windows CardSpace—to the .NET Framework 2.0. Similarly, the .NET Framework 3.5 adds few more components and features, including LINQ, ASP.NET 3.5, and ActiveX Data Objects .NET (ADO.NET) Entity Framework and Data services, to the .NET Framework 3.0.

Let's now discuss the major components of the .NET Framework 3.5, one by one.

Common Language Runtime

One of the most important components of the .NET Framework is CLR, better known as the runtime. It provides functionalities, such as memory management, exception handling, debugging, security, thread execution, code execution, code safety, code verification, compilation. The CLR can host a variety of languages and provides common tools to these languages; thereby, ensuring interoperability between code written in different languages. The managed environment of the runtime eliminates many common software issues. For example, the runtime automatically releases the objects when they are no longer in use. This automatic memory management resolves the issue of memory leaks and invalid memory references.

CLR is the module that actually runs your .NET applications. When you run a .NET application, the language compiler compiles the source code into an intermediate code, called Microsoft Intermediate Language (MSIL) code. The MSIL code is similar to Java's bytecode. The MSIL code is later converted by the Just-In-Time (JIT) compiler into native machine code, which is the final executable code. Fig.VB-1.2 explains the functioning of CLR:

ASP.NET and ASP.NET AJAX

ASP.NET is a Web development model, which is used to develop interactive, data-driven Web applications over the Internet. ASP.NET Web applications can be created using any CLR-compliant language, such as Visual Basic, Visual C#, and Visual C++.

AJAX, formerly code-named as Atlas, is an extension of ASP.NET for developing and implementing AJAX functionality. ASP.NET AJAX includes both client-side and server-side components that allows developers to create Web applications, which does not require complete reload of the page while making any modifications to the page. It enables you to send only parts of a Web page to the Web server by allowing you to make asynchronous calls to the Web server. This decreases network traffic as well as processing on the Web server.

ADO.NET

ActiveX Data Objects .NET (ADO.NET) is a technology for working with data and databases of all types. It provides access to various data sources, such as Microsoft SQL Server, and data sources exposed through OLE DB and eXtensible Markup Language (XML). You can use ADO.NET to connect to data sources for retrieving, manipulating, and updating data. The most important feature of ADO.NET is disconnected data architecture. In this architecture, applications are connected to the databases only till data is retrieved or modified.

Windows Presentation Foundation

Apart from Windows Forms, Windows client applications can also be developed through WPF (formerly codenamed as Avalon). WPF also facilitates building various kinds of interfaces, such as documents, media, two and three-dimensional graphics, animations. It helps in creating Windows client applications of superior quality. You can use WPF for creating both standalone and browser-hosted applications. WPF introduces a new language called eXtensible Application Markup Language (XAML), which is a language based on XML.

Windows Communication Foundation

Windows Communication Foundation (WCF) (formerly codenamed as Indigo) is a service-oriented technology introduced by Microsoft for building and running connected systems. The service-oriented design results in a distributed system that runs between the services and clients. You can understand WCF more easily if you are familiar with concepts, such as Web services, remoting, distributed transactions, and message queuing.

WCF based applications are interoperable with any process as these communicate through Simple Object Access Protocol (SOAP) messages. When a WCF process connects with a non-WCF process, it uses XML-based encoding for SOAP messages, but when it connects with another WCF process, the SOAP messages are encoded into a binary format.

Windows Workflow Foundation

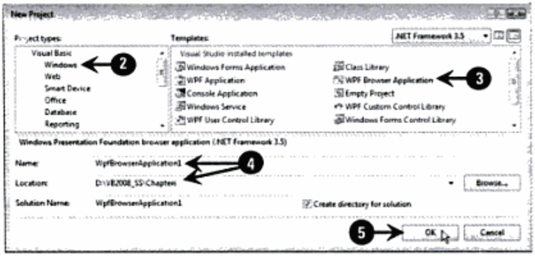
Windows Workflow Foundation (WF) is a technology introduced by Microsoft that provides a programming model for building workflow based applications on Windows. The components of WF include activities, workflow runtime, workflow designer, and a rules engine. WF is a part of .NET Framework 3.0 and 3.5.

The most important feature of WF is the separation between the business process code and the actual implementation code. Before WF was introduced, both the business logic and the actual implementation code were written together while developing applications.

Windows CardSpace

Windows CardSpace (WCS) is a client software provided by Microsoft that makes the process of securing resources easier and also makes sharing personal information on the Internet more secure. It helps programmers to develop Web sites and software that are less prone to identity related attacks such as phishing. WCS solves the problems of traditional online security mechanisms by reducing dependence on

2. In the Visual Studio 2008 IDE, *click* **File→New→Project** from the menu bar to open the **New Project** dialog box. In the **New Project** dialog box, *select* **Visual Basic→Windows** in the **Project types** pane. The project templates for Visual Basic Windows applications appear in the **Templates** pane, as shown in Fig.VB-6.5:



Name: WpfBrowserApplication1

Location: D:\VB2008_SS\Chapter6

Solution Name: WpfBrowserApplication1

☑ Create directory for solution

Fig.VB-6.5

3. *Select* **WPF Browser Application** from the **Templates** pane, as shown in Fig.VB-6.5, to create an XBAP.
4. *Specify* an appropriate name and location for the XBAP in the **Name** and **Location** boxes, respectively, as shown in Fig.VB-6.5. By default, the first XBAP that you create is named **WpfBrowserApplication1**; however, you can change it to make it more user-friendly. In this case, we accept the default name and specify the **D:\VB2008_SS\Chapter6** folder as the location.
5. *Click* the **OK** button in the **New Project** dialog box (Fig.VB-6.5). The dialog box is closed and **WpfBrowserApplication1** is created, as shown in Fig.VB-6.6:

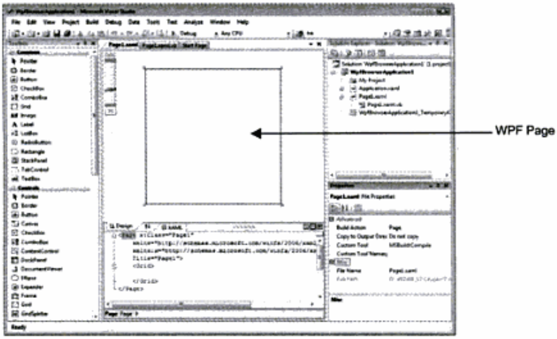


Fig.VB-6.6

As shown in Fig.VB-6.6, the UI of XBAPs is similar to the UI of standalone WPF applications. In addition, some of the essential files, such as the **Application.xaml** and **Application.xaml.vb** files, are also part of the XBAP. However, XBAPs have the **Page1.xaml** and **Page1.xaml.vb** files instead of the **Window1.xaml** and **Window1.xaml.vb** files in standalone WPF applications. The **Page1.xaml** and **Page1.xaml.vb** files can be viewed in the Solution Explorer and correspond to a page in the XBAP.

Note

The WPF 3.5 Designer

In Visual Studio 2008, the WPF 3.5 Designer is the designer interface that offers an easy, quick, and interactive way of working with the UI of WPF applications. The important WPF Designer elements for a standalone WPF application are shown in Fig.VB-6.7:

Fig.VB-6.7

As shown in Fig.VB-6.7, the WPF Designer resembles the Windows Forms Designer. For instance, the window, a form-like structure, appears at the center of the WPF Designer just as the Windows form is shown at the center of the Windows Forms Designer. In addition, the WPF Designer and Windows Forms Designer have the Solution Explorer, Properties window, and Toolbox. The difference lies in that the WPF Designer is divided into two primary views, the Design view and XAML view. Other designer elements, such as the split view bar and the tag navigator assist you in working with the Design and XAML views. Let's explore the Design and XAML views, split view bar, and the tag navigator.

The Design View

As the name suggests, the Design view is the area where you build the visual aspect or UI of a WPF application by placing, dragging and resizing, and manipulating the appearance of the controls. The Design view allows you to easily and quickly build the UI of the WPF application in a What You See Is What You Get (WYSIWYG) manner, as shown in Fig.VB-6.8:

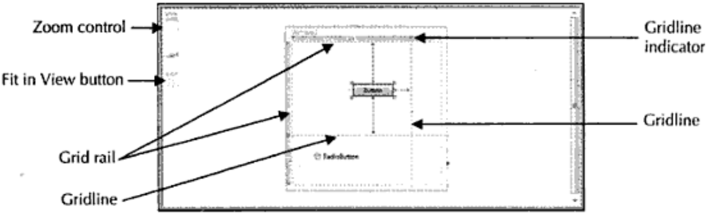


Fig.VB-6.8

As shown in Fig.VB-6.8, the Design view itself has several UI elements that assist you in quickly designing WPF applications. For instance, there is a Zoom control to zoom in or out of the Design view, move and resize handles to appropriately position and resize the controls, margin lines to set the margins of a control, and much more. Let's briefly go through the elements of the Design view.

Grid Rails, Gridlines, and Gridline Indicators

The grid rails, gridlines, and gridline indicators pertain to the **Grid** control of WPF. When you create a WPF 3.5 application, a **Grid** control is by default added to the application. The **Grid** control allows you to represent the WPF application as a grid or lattice. The entire grid is enclosed by grid rails that span horizontally and vertically on the top and left respectively, as shown in Fig.VB-6.8.

By default, the grid has one row and one column; however, there can be multiple rows and columns. You can create additional rows and columns by clicking the desired positions on the grid rails. When you click the grid rails, gridlines and gridline indicators appear at those positions (Fig.VB-6.8). If you click the grid rail on the top, then a vertical gridline appears dividing the grid into two columns. Similarly, if you click the grid rail on the left, then a horizontal gridline appears dividing the grid into two rows. You can control the height and width of the rows and columns by moving the gridline indicators, which appear as triangles on the grid rails.

The Zoom Control and Fit in View Button

You can find the Zoom control and the Fit in View button at the upper-left corner of the Design view. The Zoom control allows you to zoom in and zoom out of the Design view. To zoom in or out, you can drag the Zoom control slider up or down. The current zoom level, which in Fig.VB-6.9 is **100%**, can be seen on top of the Zoom control:

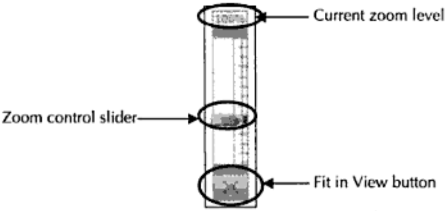


Fig.VB-6.9

In case, you want to zoom or resize the Design view according to the available space in the Design view, then you can quickly do so by clicking the Fit in View button, which appears just below the Zoom control.

control, then margin is measured from the edge of the control to the nearest gridline. Each control has four margins—top, bottom, left, and right. These margins are represented by four lines, called margin lines, emerging from the edges of the control, as shown in Fig.VB-6.12:

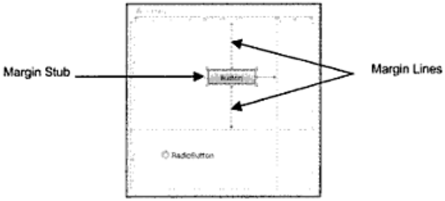


Fig.VB-6.12

Note that the margin lines appear only when the control is selected. You may notice in Fig.VB-6.12 that only the top, bottom, and right margin lines of the button appear; there is no left margin line. Instead, there is a small circle, called the *margin stub*, on the left edge of the button. The margin stub indicates that the respective margin is set to zero.

The margins for a control are collectively accessed through the **Margin** property of the control. You can set the **Margin** property of a control in the Design view, XAML view, or code-behind file. To set the **Margin** property in the Design view, select the control and drag the control. You can also use the **Properties** window to set the **Margin** property by typing values for margins. You need to provide the margin values in a clockwise manner starting from the left margin, that is, left, top, right, and bottom. Note that when you type the new margin values in the Properties window, the size of the control may change.

Snaplines

Snaplines are a feature in the WPF Design view that assists you in aligning the controls in the WPF 3.5 applications. A snapline appears only when there is more than one control and you drag or resize any control. It appears as a light brown line along the edges of the controls, as shown in Fig.VB-6.13:

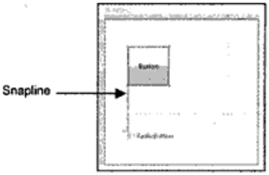


Fig.VB-6.13

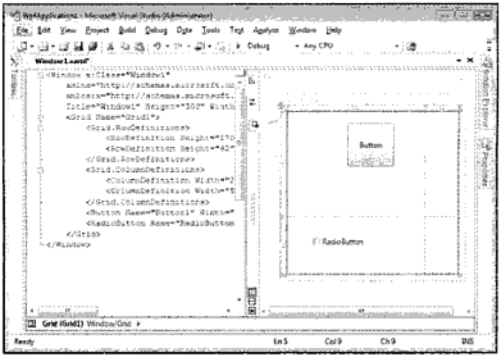
In Fig.VB-6.13, you can see that at one end of the snapline, the number **82** is displayed. This number represents the distance between the edges of the controls that you want to align. In case you do not want the snaplines to be visible, you can press **ALT** while dragging or resizing the control.

With this, you have learned about the individual UI components of the Design view. Now, let's move on to explore the XAML view of the WPF Designer.

The XAML View

As the name suggests, the XAML view allows you to view and work with the XAML code for WPF 3.5 applications. The XAML view and the Design view are inter-related to each other such that whatever

As stated earlier, there are three buttons on the right side of the split view bar. The first button, Vertical Split, allows you to vertically split the Design view and XAML view, as shown in Fig.VB-6.16:


WpfApplication1 - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Data Tools Test Analyze Window Help
Debug
Any CPU
Window1.xaml*
Toolbox
Solution Explorer
Properties
Button
RadioButton
Grid (Grid1) Window/Grid
Ready
Ln 5
Col 9
Ch 9
INS

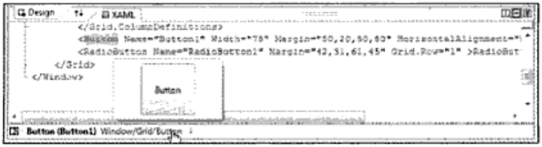
```
<Window x:Class="Window1"
    xmlns="http://schemas.microsoft.co
    xmlns:x="http://schemas.microsoft.
    Title="Window1" Height="300" Width
    <Grid Name="Grid1">
        <Grid.RowDefinitions>
            <RowDefinition Height="170
            <RowDefinition Height="42"
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="2
            <ColumnDefinition Width="5
        </Grid.ColumnDefinitions>
        <Button Name="Button1" Width="
        <RadioButton Name="RadioButton
    </Grid>
</Window>
```

Fig.VB-6.16

The second button, Horizontal Split, allows you to see both the Design view and XAML view horizontally, which is the default view. The third button, Collapse Pane, allows you to collapse or hide one of the panes. For example, if you click the Collapse Pane button, then the Design pane collapses and you get more space for the XAML view. In case you want to expand the collapsed pane, you can click the Expand Pane button that appears in place of the Collapse Pane button.

The Tag Navigator

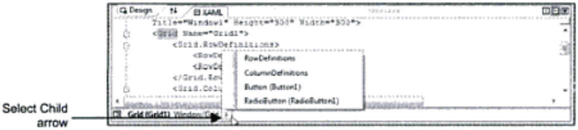
The tag navigator appears just below the XAML view. The tag navigator allows you to navigate to the parent XAML tag or element of the currently selected element. It displays the currently selected tag (in bold) and the hierarchy up to that tag, as shown in Fig.VB-6.17:



```
        </Grid.ColumnDefinitions>
        <Button Name="Button1" Width="75" Margin="50,20,50,60" HorizontalAlignment="
        <RadioButton Name="RadioButton1" Margin="42,31,61,45" Grid.Row="1" >RadioBut
    </Grid>
</Window>
```

Fig.VB-6.17

As shown in Fig.VB-6.17, the **Button** control (or **Button** tag) is the currently selected tag in the XAML view. Its name is displayed in bold in the tag navigator. The name of the **Button** tag is followed by hyperlinks that are separated from each other by a forward slash, that is, **Window/Grid/Button**. The hyperlinks represent the hierarchy up to the currently selected tag. If you place the mouse curser over a hyperlink, a thumbnail image is displayed (Fig.VB-6.17). You can click a hyperlink to select the tag that it represents. Note that if the currently selected tag has any child tags, then the **Select Child** arrow is enabled and when you click the arrow, a list of the child tags appear, as shown in Fig.VB-6.18:



```
    Title="Window1" Height="300" Width="300">
    <Grid Name="Grid1">
        <Grid.RowDefinitions>
            <RowDe
            <RowDe
        </Grid.Row
        <Grid.Colu
```

Fig.VB-6.18

XAML and WPF

WPF supports XAML (pronounced as *zammel*), which is a declarative markup language based on eXtensible Markup Language (XML) introduced by Microsoft Corporation. This markup language allows designers to easily and quickly define and describe the elements for the UI of WPF applications. Application developers can then specify program logic for the UI elements defined through XAML in the code-behind files, using any of the .NET languages, such as Visual Basic. However, you can also use the code-behind files to create UI elements at run time. Consequently, WPF allows both segregation and integration of the UI aspect and application logic in WPF applications.

Visual Studio 2008 extends support for XAML by incorporating XAML IntelliSense and allowing the debugging and compilation of XAML content. Here, you learn how to work with the XAML elements and attributes, namespaces, and markup extensions.

Note

XAML files have the .xaml extension. XAML files are also supported in Windows Workflow Foundation (WF) and Microsoft Silverlight.

XAML Elements and Attributes

XAML makes use of markup tags or elements and attributes to define the UI of WPF applications. The XAML elements directly correspond to various managed WPF classes, while the XAML attributes correspond to the properties and events of the classes.

In WPF, the XAML elements are represented as a logical tree with several nodes. Each element corresponds to a tree node while the attributes of the elements become the properties of the nodes. When you add an XAML element within an existing element, it becomes the child element of the existing element and therefore the child node of the existing node. In this way, as you keep adding XAML elements in a WPF application, the tree branches out to reflect the UI of the application. Note that in any XAML file, there is exactly one topmost or root element. In WPF standalone applications, the root element is the **Window** element, while in XBAPs, the root element is the **Page** element.

Let's take the example of a new WPF standalone application named **MyWPFApplication**. Listing 6.1 shows the XAML code of the application:

As shown in Listing 6.1, there are four XAML elements, namely **Window**, **Grid**, **Button**, and **Label**. The **Window** and **Grid** elements exist in the WPF application by default, while the **Button** and **Label** elements are added by dragging and dropping the respective controls from the Toolbox. These elements can be represented in a hierarchical manner as a tree, as shown in Fig.VB-6.19:

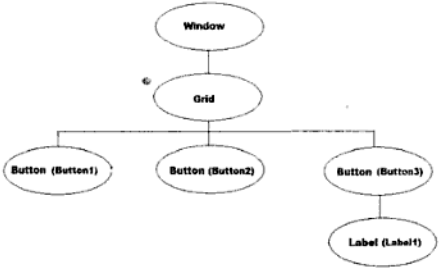


Fig.VB-6.19

In Fig.VB-6.19, the **Window** element contains the **Grid** element, which in turn has three **Button** elements, named **Button1**, **Button2**, and **Button3**, as its child elements. The **Button3** element itself has a **Label** element, named **Label1**, as its child element. Note that the **Window**, **Grid**, **Button**, and **Label** XAML elements correspond to the instances of the **Window**, **Grid**, **Button**, and **Label** classes, respectively, while the attributes of the XAML elements correspond to the properties of the respective classes.

You can recall from Listing 6.1 that the individual attributes are separated by whitespaces and are specified in name-value pairs. For every attribute, the name and value of an attribute is separated by the equal to operator (=). Note that the value is enclosed within double quotes, for example:

```
    <Button Height="23" Name="Button1">Welcome</Button>
```

In the preceding code snippet, the value of the **Height** attribute is **23** and the value of the **Name** attribute is **Button1**. The string **Welcome** is the text that appears on the button.

Although setting the properties through XAML attributes is easy, concise, and intuitive, WPF provides an alternate way, known as the *property element syntax*, of setting the property values. In the property element syntax, you can set the properties by specifying them as elements rather than as attributes. In the property element syntax, the property is set by using the following syntax:

In the preceding syntax,

- **element_name**: The name of the element to which the property belongs
- **property_name**: The name of the property
- **attribute1_name, attrubute2_name, . . . attributeN_name**: The names of the attributes that the property may contain
- **value1, value2, . . .valueN**: The values of the attributes
- **property_value**: The value of the property

As shown in the preceding syntax, the dot operator (.) separates the element and property names. In Listing 6.2, you can see the property element syntax for the properties of the **Button** element:

You can provide a markup extension as either attributes or property elements. If you want to provide a markup extension as an attribute, then you need to enclose it within the curly braces ({ }). The opening curly brace is followed by the string token for the markup extension class. The string token is followed by a whitespace, which is then followed by input to the markup extension. Let's take a look at the XAML code:

```
<Button Margin="100" Name="Button1" Content="Welcome" Style="{StaticResource
  mybackground}"/>
```

In the preceding code, the **StaticResource** markup extension is used in the **Button** element to set the value for its **Style** property. This implies that the **Style** property of the **Button** element takes the value of the **mybackground** resource. You learn more about resources and styles later in this chapter.

Common Controls in WPF 3.5

A control is an UI element that allows interaction between the application and users of the application. WPF has a rich set of controls that allow you to develop graphic-rich applications. Some of the WPF controls are similar to those in Windows Forms, such as the **Button** and **TextBox** controls. However, there are certain WPF controls, such as **Grid** and **Canvas** that are unique in terms of their appearance and behavior.

In WPF, you can add and work with the controls in both the Design view and code-behind file. However, it is recommended that you define the look of the controls in the Design view and define the behavior of the controls in the code-behind file. You can add a control in a WPF application by dragging it from the Toolbox and dropping it on the Design view. You can also add a control by double-clicking it in the Toolbox or writing the corresponding XAML code in the XAML View. Note that changes to a control in the Design view automatically reflects in the XAML view and changes in the XAML view reflects in the Design view.

Note

Here, you learn how to add and work with some of the common controls in WPF 3.5 that are as follows:

□ The **Grid** control

□ The **Button** control

□ The **TextBox** control

□ The **PasswordBox** control

□ The **TextBlock** control

□ The **Border** control

□ The **GridSplitter** control

□ The **Canvas** control

□ The **StackPanel** control

Let's start with the **Grid** control.

Using the Grid Control

When you created a WPF application, you may have noticed a **Grid** control is automatically added to the default window or page in the application. Every page in a WPF application (standalone or XBAP) has a **Grid** control by default.

The **Grid** control is one of the most common and flexible controls. A container control refers to a control that encompasses other controls and provides a layout for those controls. The controls that are encompassed or contained within a container control known as child controls or child elements. In a **Grid** control, the child elements are contained in the cells the grid. By default, a **Grid** control consists of a single cell; however, you can create additional cells in **Grid** control by using its properties.

The **Grid** control is an instance of the **Grid** class, which has properties that allow you to work with the **Grid** control. Table 6.2 lists the noteworthy properties of **Grid** class:

Let's a new standalone WPF application to the use of the **Grid** control. For this, perform the steps:

1. *Click* **All Programs→Microsoft Visual Studio Visual Studio 2008** to open the Studio 2008 IDE.
2. *Create* a new standalone WPF application with the name
3. In the **Window1.xaml** file, *add* the code given in Listing

Listing Adding Rows and Columns in a **Grid** Control

In Listing 6.3, the **ShowGridLines** property of the **Grid** control is set to **True**, which implies that straight lines are visible between the rows and columns of the grid. The grid has two child elements—**Grid.RowDefinitions** and **Grid.ColumnDefinitions**. The **Grid.RowDefinitions** element refers to the **RowDefinitions** property of the grid allowing you to define a collection of rows in the grid. The **Grid.ColumnDefinitions** element refers to the **ColumnDefinitions** property, which allows you to define a collection of columns in the grid. The **Grid.RowDefinitions** element has four **RowDefinition** elements, each corresponding to a row. Similarly, the **Grid.ColumnDefinitions** element has two **ColumnDefinition** elements, each of which corresponds to a column.

4. *Press* the **F5** key to run the application. The output of the application is shown in Fig.VB-6.20:

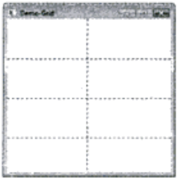


Fig.VB-6.20

Using the Button Control

The **Button** control in WPF is similar to the one available for Windows Forms applications. It allows you to perform an action when a user clicks it. The **Button** control in WPF is a basic UI component that can contain text as well as an image.

The **Button** control is an instance of the **Button** class. Table 6.3 lists the noteworthy properties of the **Button** class:

Let's create a new standalone WPF application to demonstrate the use of the **Button** control. For this, perform the following steps:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2008→Microsoft Visual Studio 2008** to open the Visual Studio 2008 IDE.
2. *Create* a new standalone WPF application with the name **ButtonDemoVB**.

2. *Create* a new standalone WPF
3. In the **Window1.xaml** file, add

Listing 6.5: Demonstrating the Use of

In Listing 6.5, a **TextBox** control is added to the control. The **TextBox** control is and has its **CharacterCasing** property set to **Upper**. This implies that all the characters of the enter appear in uppercase. The **TextWrapping** and **TextAlignment** properties of the **TextBox1** control are set to **Wrap** and **Center**, respectively. This implies that the entered text is wrapped around the control and is aligned at the center of the control. Note that another property, named **VerticalScrollBarVisibility**, of the **TextBox1** control is set to **Auto**. This property is inherited from the **TextBoxBase** class and refers to whether a vertical scroll bar appears in the control.

4. *Double-click* the **Button1** button in the Design view. This opens the code-behind file (**Window1.xaml.vb** file) and adds the **Click** event handler in the code-behind file.
5. In the **Click** event handler of **Button1**, *add* the following code in the **Window1** class:

6. *Press* the **F5** key to run the application and *enter* some text in the **TextBox** control (Fig.VB-6.22):

Fig.VB-6.22

7. *Click* the **Clear TextBox** button (Fig.VB-6.22). The text in the **TextBox** control is cleared, as shown in Fig.VB-6.23:

Fig.VB-6.23

Note

Using the Control

There may be a situation when you want users of your application to enter some confidential information such as passwords. You may use a **TextBox** control to allow users to enter their passwords. However, while a user enters the password, there is a possibility that some other user may accidentally see the password, which is an undesirable situation. In such a situation, you can mask the password while entering it by using the **PasswordBox** control. The **PasswordBox** control is a special type of text box that enables users to enter and manipulate passwords. In the **PasswordBox** control, the each character of the input text is masked with a given character such that the text appears as a string of that character. By default, any text entered in the **PasswordBox** control appears as a series of filled circles.

Note

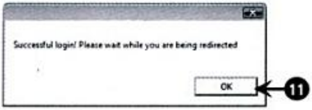
The control is represented by the **PasswordBox** class, which has various properties that allow you to work the **PasswordBox** control. Table 6.5 lists the noteworthy properties of the **PasswordBox** class:

Let's create a new standalone WPF application to demonstrate the use of the **PasswordBox** control. For this, perform the following steps:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2008→Microsoft Visual Studio 2008** to open the Visual Studio 2008 IDE.
2. *Create* a new standalone WPF application with the name **PasswordBoxDemoVB**.
3. In the **Window1.xaml** file, *add* the code given in Listing 6.6:

Listing 6.6: Demonstrating the Use of the **PasswordBox** Control

10. *Click* the **Login** button (Fig.VB-6.26). A message box appears, as shown in Fig.VB-6.27:



Successful login! Please wait while you are being redirected

Fig.VB-6.27

11. *Click* the **OK** button on the message box to close it (Fig.VB-6.27).

Note

> The TextBox and PasswordBox controls are simple controls, that is, controls that cannot have any child controls.

Using the TextBlock Control

The **TextBlock** control in WPF allows you to work with text in a flexible manner as compared to the **Label** control. The **TextBlock** control has several properties that facilitate easy manipulation of the appearance of the text that it holds.

The **TextBlock** control is an instance of the **TextBlock** class, which offers various properties. Some of the

Let's create a new standalone WPF application to demonstrate the use of the **TextBlock** control. For this, perform the following steps:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2008→Microsoft Visual Studio 2008** to open the Visual Studio 2008 IDE.
2. *Create* a new standalone WPF application with the name **TextBlockDemoVB**.
3. In the **Window1.xaml** file, *add* the code given in Listing 6.8:

Listing 6.8: Displaying Text by Using the **TextBlock** Control

properties of the control are set to **White** and **Black** respectively. This implies that the text in the **TextBlock** control appears in white, while the background appears black. The **FontSize** and **FontWeight** properties of the **TextBlock** control are set to **17** and **Bold** respectively so that the text appears large. In addition, the **TextWrapping** and **TextAlignment** properties are set to **Wrap** and **Center**, respectively. This implies that the text (if it spans multiple lines) is wrapped around the control and is displayed in the center of the control.

4. *Press* the **F5** key to run the application. The output of the application is shown in Fig.VB-6.28:

Fig.VB-6.28

Using the Border Control

The **Border** control in WPF provides the facility to add a border, background, or both to other WPF controls. The **Border** control can have only one child element. This implies that at a time, the **Border** control can apply a border or background to only one element. In other words, you cannot apply the **Border** control to multiple child elements.

The **Border** control is an instance of the **Border** class, which has several properties. Some of the noteworthy properties are listed in Table 6.7:

Let's create a new standalone WPF application to demonstrate the use of the **Border** control. For this, perform the following steps:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2008→Microsoft Visual Studio 2008** to open the Visual Studio 2008 IDE.
2. *Create* a new standalone WPF application with the name **GridWithBorderVB**.
3. In the **Window1.xaml** file, add the code given in Listing 6.9:

Listing 6.9: Adding Border to a **Grid** Control

a **Border** control is added as a child element to the **Grid** in the file. **Background** property of the **Border** control, the background color of the **Grid** is set to and the **BorderThickness** properties of the **Border** control are set to **Black** and **9**, In addition, the **CornerRadius** property of the **Border** control is set to **45**, that is, the corners of the are inclined at **45** degrees.

4. *Press* the **F5** key to run the application. The output of the application is shown in Fig.VB-6.29:

Fig.VB-6.29

Using the GridSplitter Control

The **GridSplitter** control in WPF offers a unique facility of redistributing the space between two rows or two columns of a **Grid** control. With the **GridSplitter** control, the height and width of the **Grid** control do not change but the space, between two adjacent rows or two adjacent columns of the grid, changes.

The **GridSplitter** control is an instance of the **GridSpliter** class, which has various properties to allow you to manipulate the spaces between two rows or two columns of a **Grid** control. Table 6.8 lists the noteworthy properties of the **GridSplitter** class:

perform the following steps:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2008→Microsoft Visual Studio 2008** to open the Visual Studio 2008 IDE.
2. *Create* a new standalone WPF application with the name **GridSplitterDemoVB**.
3. In the **Window1.xaml** file, *add* the code given in Listing 6.10:

Listing 6.10: Splitting a **Grid** Control by Using a **GridSplitter** Control

In Listing 6.10, the **Grid** control is divided into three rows and three columns—that constitute nine cells. In the second row, a **GridSplitter** control is added. The **ShowsPreview** property of the **GridSplitter** control is set to **True**. Setting this property displays the preview of how the rows and/or columns are resized when the **GridSplitter** control is moved. The **ResizeDirection** and **ResizeBehavior** properties of the **GridSplitter** control are set to **Rows** and **CurrentAndNext**, respectively. This implies that the **GridSplitter** control can resize only the rows, specifically the current (second row) and the next row (third row).

4. *Press* the **F5** key to run the application. The output of the application is shown in Fig.VB-6.30:

Fig.VB-6.30

As shown in Fig.VB-6.30, the **GridSplitter** control appears in the second row of the grid.

5. *Drag* the **GridSplitter** control downwards, that is, towards the third row, as shown in Fig.VB-6.31:

Fig.VB-6.31

the top and left sides of the **Canvas** control. Similarly, the **Button4** button appears at a distance of **15** units from both the bottom and right sides of the **Canvas** control.

4. *Press* the **F5** key to run the application. The output of the application is shown in Fig.VB-6.32:

Fig.VB-6.32

Using the StackPanel Control

Suppose you want to place multiple controls on a WPF window such that one appears on top of the other, forming a vertical stack of those controls. In such scenarios, instead of using a **Grid** or **Canvas** control and explicitly placing those controls accordingly, you can use a **StackPanel** control. The **StackPanel** control is another container control that allows you to position its child controls in a vertical or horizontal stack. By default, the child controls of the **StackPanel** control are stacked vertically. The vertical stack increases from top to bottom. You can also stack the child controls horizontally. The horizontal stack increases from left to right.

The **StackPanel** control is represented by the **StackPanel** class, which is derived from the **Panel** class. The **StackPanel** class has several properties that assist you in working with the **StackPanel** control. The noteworthy properties of the **StackPanel** class are listed in Table 6.10:

Let's create a new standalone WPF application to demonstrate the use of the **StackPanel** control. For this, perform the following steps:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2008→Microsoft Visual Studio 2008** to open the Visual Studio 2008 IDE.
2. *Create* a new standalone WPF application with the name **StackPanelDemoVB**.
3. In the **Window1.xaml** file, *add* the code given in Listing 6.12:

Listing 6.12: Using the **StackPanel** Control

In Listing 6.12, the **Grid** control has two **StackPanel** controls as its child controls. Both the **StackPanel** controls have three buttons as the respective child controls. However, the child controls of the two **StackPanel** controls are stacked differently. In the first **StackPanel** control named **StackPanel1**, the three buttons (**Button1**, **Button2**, and **Button3**) are stacked vertically—one below the other. This is because the **Orientation** property of the **StackPanel1** control is set to **Vertical**. However, the **Orientation** property of the second **StackPanel** control (named **StackPanel2**) is set to **Horizontal** and hence has its child controls stacked horizontally.

4. *Press* the **F5** key to run the application. The output of the application is shown in Fig.VB-6.33:

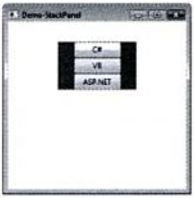


Fig.VB-6.33

Now that you have learned about some of the common controls used in WPF, let's move ahead to learn about resources and styles in WPF.

Resources and Styles

Consider a situation where you want to apply a background color on several elements in a WPF application. You may set the **Background** property on those elements; however, this may prove to be rather tedious. WPF allows you to simplify and ease the use of such commonly used objects through *resources*. A resource refers to an object, element, or value that is part of the resource dictionary and is reusable by other elements of the application. Resource dictionary is an instance of the **ResourceDictionary** class and refers to a collection of resources defined on an element.

Although you can define resources for any UI element; however, it is advisable to define the resources on the root element such as the **Window** or **Page** element. The resource that you define for an element also applies to all the child elements of that element. For example, if you define resource for a **Window** element with a **Grid** element as its child, then the resource for the **Window** element applies to the **Grid** element also. Furthermore, if you define a resource for the **Grid** element, then that resource applies to the child elements of the **Grid** element as well.

Note

In Listing 6.13, the **Grid** control uses the **Resources** property to define a set of resources. Two **SolidColorBrush** resources (**SolidColorBrush** elements that fill a given area with a solid color) with the keys **firstBrush** and **secondBrush** are defined. These two resources are used to set the **Background** property of the three buttons (**Button1, Button2,** and **Button3**). The **StaticResource** markup extension is used to set the **Background** property to the resource values.

4. *Press* the **F5** key to run the application. The output of the application is shown in Fig.VB-6.34:

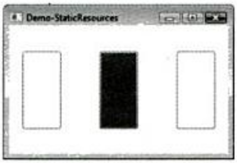


Fig.VB-6.34

Let's now move on to learn about dynamic resources.

Using a Dynamic Resource

When you use the **DynamicResource** markup extension, the resource is referred to as a dynamic resource. As compared to static resources, dynamic resources provide more flexibility and are useful in situations when you want to defer the assignment of a property value until run time. For instance, situations when you want to change the value of the resource at run time through user or system settings.

Unlike static resources that are looked up at the time of loading, dynamic resources are looked up only when they are used at run time. With dynamic resources, an expression is created for the requested resource. This expression is not evaluated until run time. When the dynamic resource is used at run time, the key of the requested resources is looked up in the resources of the element on which the property is being set. If the key is not found, then it is looked up upwards in the parent element and its resource dictionary up to the root element. If the key is still not found, then the application, theme, and system resources are looked up in that order.

Let's create a new standalone WPF application to learn how to use dynamic resources. For this, perform the following steps:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2008→Microsoft Visual Studio 2008** to open the Visual Studio 2008 IDE.
2. *Create* a new standalone WPF application with the name **DynamicResourcesDemoVB.**
3. In the **Window1.xaml** file, add the code given in Listing 6.14:

Listing 6.14: Using a Dynamic Resource in XAML

In Listing 6.14, you can see that the **Window** control has three **SolidColorBrush** resources with the keys **firstBrush**, **secondBrush**, and **thirdBrush**. The **firstBrush** resource is referenced in the **Background** property of the **Grid1** control as a static resource. The **Background** property of the **Grid1** control is then used as a dynamic resource in the **Button1** control. This implies that whenever the background of the **Grid1** control changes, the background of the **Button1** control also changes.

4. *Double-click* the **Button2** button in the Design view and *add* the following highlighted code for the **Button2_Click** event handler in the Code Editor:

In the preceding code, inside the **Button2_Click** event handler, the background of the **Grid1** control is set to the value of the **secondBrush** resource by using the **FindResource** method.

5. *Double-click* the **Button3** button in the Design view and *add* the following highlighted code for the **Button3_Click** event handler in the Code Editor:

the value of the **thirdBrush** resource by using the **FindResource** method.

6. *Press* the **F5** key to run the application. The output of the application is shown in Fig.VB-6.35:

Fig.VB-6.35

Note that the background of **Dynamic Button** is same as the background of **Grid1**.

7. *Click* the **Color1** button (Fig.VB-6.35). The color of **Window1** changes, as shown in Fig.VB-6.36:

Fig.VB-6.36

8. *Click* the **Color2** button (Fig.VB-6.36). The color of **Window1** changes, as shown in Fig.VB-6.37:

Fig.VB-6.37

Let's now move ahead to learn about setting styles through resources.

Setting Style Through a Resource

One of the most common uses of resources in WPF is to apply styles uniformly to several elements. In most cases, styles are defined as resources in WPF applications and therefore are included in the resource dictionary of an element. In this way, styles become reusable entities allowing them to be used with multiple elements in an application.

In WPF, you use the **Style** element while defining the resources for an element. Each **Style** element has one or more elements that specify the property and its value to which the style is to be applied. The syntax of the element is:

In the syntax,

- **x:Key:** The key name of the **Style** element
- **TargetType:** The name of the type of an element on which the style is to be applied
- **propertyName1**, **propertyName2**, . . . , **propertyNameK:** The names of the properties
- **propertyValue1**, **propertyValue2**, . . . , **propertyValueK:** The values of the properties

Let's create a new standalone WPF application to learn how to set styles through resources. For this, perform the following steps:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2008→Microsoft Visual Studio 2008** to open the Visual Studio 2008 IDE.
2. *Create* a new standalone WPF application with the name **StylesDemoVB**.
3. In the **Window1.xaml** file, *add* the code given in Listing 6.15:

Listing 6.15: Using Styles as Resources

Introduction

Windows Workflow Foundation (WF) build a workflow and add the workflow to an application. A that defines the flow of a process or a set of tasks that produces a result. WF along with three other foundations. These three foundations are (WPF), Windows Communication Foundation (WCF), and Windows 2008 has built-in templates (that were not available with Visual Studio 2005) to applications.

This chapter begins with the discussion on the principles of workflows and later explains the components of WF. Further, you learn to develop a simple workflow application, implement conditions in workflows, and use the workflows with other applications.

Let's begin with the principles of workflows.

Workflow Principles

The workflow platform that you use to develop workflow-based applications should embody some principles. These principles are as follows:

- Workflows coordinate work performed by people and software
- Workflows are long running and stateful
- Workflows are based on extensible models
- Workflows are transparent and dynamic throughout their lifecycle

Let's discuss these principles in detail one by one.

Workflows Coordinate Work Performed by People and Software

People play an important role in the world of software systems related to workflow and processes. Human interaction is often done through e-mail, Web pages, mobile devices, or other front ends. WF provides the necessary infrastructure to effectively handle human interaction and all the related issues.

Workflows are Long Running and Stateful

Humans are inherently less predictable than software systems because they are supposed to interact with the software systems on an ad hoc basis. For example, they may interact with the software systems after few minutes, hours, or months. Due to this reason, workflows need to be able to run for long periods. However, running a workflow for long periods, and storing a running workflow in memory is not practical due to many reasons. If every running workflow has to be stored in memory while waiting for something to happen, the server would run out of memory immediately. Additionally, if the server crashes, the volatile memory will be cleared and all data will be lost.

Workflows are based on Extensible Models

As stated earlier, workflows serve the purpose of automating business processes. Now, since each type of business has a wide range of problems; therefore, a workflow platform needs to be extensible. WF provides you a set of base activities such as IfElse, Code, and Delay, to build a workflow. You can extend these activities or build new activities to meet your requirements. Besides activities, you can also extend services such as, tracking, management, and persistence, provided by the runtime engine.

Workflows are Transparent and Dynamic throughout their Lifecycle

Workflows are transparent and dynamic both at design time and run time. As WF is based on a declarative and visual design-time model, existing workflows can be modified without changing the source code.

In this section, we discussed the principles of workflows. Next, we discuss the various components of WF.

Components of Windows Workflow Foundation

WF consists of several components that work together with your application to perform the desired workflow. Fig.VB-7.1 shows the architecture of WF, which displays how the components of WFs fit together:

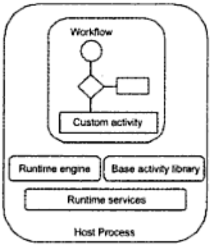


Fig.VB-7.1

In WF architecture, there are following six major components:

- Workflow
- Base activity library
- Custom Activities
- Host process
- Runtime engine
- Runtime services

Let's discuss these components one by one.

Workflow

A workflow is a declarative program wherein each program statement is represented in terms of a component, called an **activity**. In other words, a workflow defines the business logic upon which a program is based. A business process can involve applications as well as people. Workflows that are developed to automate interactions among applications are known as **system workflows**. Such workflows are usually static and predictable. In contrast to the system workflows, workflows that are intended to coordinate interactions among people are known as **human workflows**. Applications that involve human interactions generally need more flexibility than others, because people may change their minds, introduce new ideas and exceptions, and cancel a process unexpectedly. Due to the differences between system and human workflows, integrating these two together becomes a challenging task. Yet, WF tries to support both system and human workflows in a unified manner. To build both kind of workflows, WF provides the following two types of built-in workflows:

- Sequential workflow
- State machine workflow

Let's examine these two workflow types one by one.

Sequential Workflow

Sequential workflows are used in applications where the workflow's activities are executed in a well-defined order. This workflow executes a set of activities in order, one by one. It may involve branching or looping; however, the flow of the workflow generally moves from top to bottom. Fig.VB-7.2 shows an example of a sequential workflow:

Host Process

Since a workflow created with WF is not a standalone product, therefore it needs a host application to be hosted and run. Host process is the process within which a workflow is hosted and run. A host process may be a Windows Forms application, a Web application, or a Web service application. The host process can also be a place where generally user interaction takes place.

Runtime Engine

A runtime engine is not a separate service or process, it runs within the host process and is responsible to execute each workflow instance. A host process may have multiple runtime engines running concurrently and each engine executes multiple workflow instances simultaneously.

Runtime Services

Runtime services consist of predefined and user-defined classes that essentially live in the workflow runtime engine during execution. The followings are important runtime services provided by WF to your application:

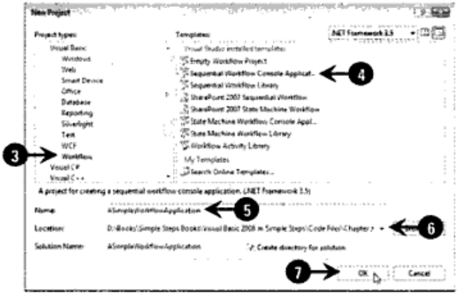
- **Persistence services:** These services enable you to save the state of a workflow for later use. You can restart the workflow as per the requirement, even after the weeks of inactivity.
- **Tracking services:** These services enable developers to monitor the state of the workflows. This is particularly useful when you have multiple workflows active at the same time (e.g. in a shopping cart application).
- **Transactions services:** These services provide the transaction support needed for data integrity.

In this section, we discussed the various components of WF. Next, we learn to create a simple workflow application.

Developing a Simple Workflow Application

A workflow application is similar to a Windows Forms application and can be developed by performing the similar steps that are required to develop a Windows Forms application. Let's now learn how to develop a simple workflow application by performing the following steps:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2008→Microsoft Visual Studio 2008** to start Visual Studio 2008.
2. *Click* **File→New→Project** on the menu bar or *press* the **CTRL+SHIFT+N** keys together. This opens the **New Project** dialog box, as shown in Fig.VB-7.4:


New Project
Project types:
Visual Basic
Windows
Web
Smart Device
Office
Database
Reporting
Silverlight
Test
WCF
Workflow
Visual C#
Visual C++
Templates:
.NET Framework 3.5
Visual Studio installed templates
Empty Workflow Project
Sequential Workflow Console Applicat...
Sequential Workflow Library
SharePoint 2007 Sequential Workflow
SharePoint 2007 State Machine Workflow
State Machine Workflow Console Appl...
State Machine Workflow Library
Workflow Activity Library
My Templates
Search Online Templates...
Create directory for solution
OK
Cancel
3
4
5
6
7

A project for creating a sequential workflow console application. (.NET Framework 3.5)

Name: ASimpleWorkflowApplication

Location: D:\Books\Simple Steps Books\Visual Basic 2008 in Simple Steps\Code Files\Chapter 7

Solution Name: ASimpleWorkflowApplication

☑ Create directory for solution

Fig.VB-7.4

3. Now, *select* **Workflow** under the **Visual Basic** node in the **Project types** pane, as shown in Fig.VB-7.4.
4. Then, *select* **Sequential Workflow Console Application** in the **Templates** pane (Fig.VB-7.4).
5. Now, *type* a name for your application in the **Name** text box, as shown in Fig.VB-7.4. In this case, we have typed **ASimpleWorkflowApplication**.
6. Then, *enter* the complete path of the folder where you want to save your application in the **Location** box, as shown in Fig.VB-7.4. In this case, we have entered **D:\Books\Simple Steps Books\Visual Basic 2008 in Simple Steps\Code Files\Chapter 7**.
7. Then, *click* the **OK** button, as shown in Fig.VB-7.4. This closes the **New Project** dialog box and creates a new Sequential Workflow Console application, as shown in Fig.VB-7.5:

Fig.VB-7.5

Notice Fig.VB-7.5 that by default, a sequential workflow has only two steps, start and finish.

8. *Drag* a **Code** activity from the **Windows Workflow v3.0** tab of the **Toolbox,** *drop* it between the start, and finish markers on the form design view (Fig.VB-7.6).
9. Then, *change* the **Name** property of the **Code** activity to **ShowMessageActivity**, as shown in Fig.VB-7.6:

Fig.VB-7.6

10. Now, *double-click* the design view to open the Code Editor and *add* the

11. Now, *press* the **F5** key on of the application is in Fig.VB-7.7:

Fig.VB-7.7

In this section, we learned to develop a simple workflow application. Next, we learn how to implement conditions in workflows.

Implementing Conditions in Workflows

Similar to a Visual Basic program, you can also implement conditions in the workflow that you create with WF. Activities, such as IfElse and While, can be used to implement a condition in a workflow application. You can implement a condition by using one of the following ways:

- **By creating a rule condition:** In this approach, the condition is created either directly in code or using a tool, called the Rule Condition Editor. Rule conditions are stored in a separate eXtensible Markup Language (XML) file. When a rule condition is encountered in a workflow, the expression in the condition is evaluated and a **Boolean** value is returned.
- **By creating a code condition:** In this approach, the condition is directly expressed in code. A code condition can be created by writing a method in the code. The method contains code for the condition and returns a **Boolean** value. Now, when the workflow is executed and the condition is encountered, the method that contains code for the condition is called. In this process, the value returned by the method is used as the result for the code condition.

Let's now learn how to implement conditions in a workflow application by performing the following steps:

1. *Create* a new workflow application with the name **ImplementingConditions**.
2. Then, *add* a **While** activity and an **IfElse** activity from the Windows Workflow v3.0 tab of the Toolbox to the form design view (Fig.VB-7.8).
3. Then, *add* two **Code** activities, one for each branch of the **IfElse** activity and also *add* a third **Code** activity just before the finish marker, as shown in Fig.VB-7.8:

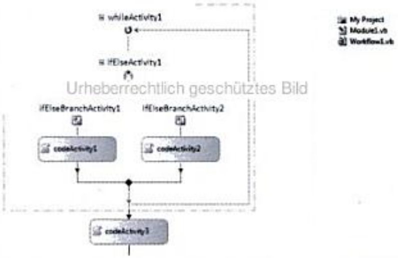


Fig.VB-7.8

4. , *add* the following code snippet inside the **Workflow1** class (outside all methods) in the **.vb** file:

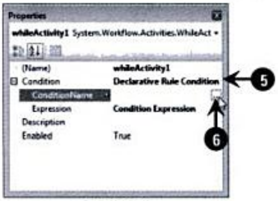
code snippet, we have defined two **Integer** type arrays, and **marks**, and an type **i**. The **rollNumbers** array stores roll numbers of five students and the **marks** array stores the of these five students.

Now, set conditions for the **While** activity and for each branch of the **IfElse** activity.

5. the **While** activity on the form design view and then *select* the **Declarative Rule Condition** the **Condition** property, in the **Properties** window. This option enables you to create a rule

Note

6. , *click* the ellipsis **(...)** button in front of the **ConditionName** property, as shown in Fig.VB-7.9:



Properties

whileActivity1 System.Workflow.Activities.WhileAct

(Name)	whileActivity1
Condition	Declarative Rule Condition
ConditionName	
Expression	Condition Expression
Description	
Enabled	True

Fig.VB-7.9

This opens the **Select Condition** dialog box, as shown in Fig.VB-7.10:

10. *Click* the **OK** button to close the **Select Condition** dialog box, as shown in Fig.VB-7.12.

Similar to the **While** activity, you can set conditions for each branch of the **IfElse** activity.

11. *Set* the following condition for the first branch of the **IfElse** activity by performing steps 5 to 9:

```
this.marks[i] >= 45
```

12. Similarly, *set* the following condition for the second branch of the **IfElse** activity by performing steps 5 to 9:

```
this.marks[i] < 45
```

13. Now, *double-click* the first **Code** activity, **codeActivity1**, on the form design view to open the Code Editor and *add* the following highlighted code snippet to the Code Editor:

14. Then, *double-click* the second **Code** activity, **codeActivity2**, on the form design view to open the Code Editor and *add* the following highlighted code snippet to the Code Editor:

15. Similarly, *double-click* the third **Code** activity, **codeActivity3**, on the form design view to open the Code Editor and *add* the following highlighted code snippet to the Code Editor:

16. Now, *press* the **F5** key on the keyboard to run the application. The output of the application is shown in Fig.VB-7.13:

Fig.VB-7.13

In this section, we learned how to implement conditions in workflows. Next, we learn how we can use our custom workflows with some other applications, such as a Windows Forms application.

Using Workflows with Other Applications

WF allows you to create standalone workflow applications that can later be integrated with some other applications, such as Windows Forms applications and Web applications. Here, we are going to discuss how we can use workflows with a Windows Forms application. For this purpose, first, we create a workflow library that contains a workflow and then we will add a Windows Forms project to the application that will make use of the workflow created in the workflow library. To create the complete application, perform the following steps:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2008→Microsoft Visual Studio 2008** to start Visual Studio 2008.

2. *Click* **File→New→Project** on the **CTRL+SHIFT+N** keys together. This opens the **New Project** dialog box, as shown in

Fig.VB-7.14

3. Now, *select* **Workflow** under the **Visual Basic** node in the **Project types** pane, as shown in Fig.VB-7.14.
4. Then, *select* **Sequential Workflow Library** in the **Templates** pane, (Fig.VB-7.14).
5. Now, *type* a name for your application in the **Name** text box, as shown in Fig.VB-7.14. In this case, we have typed **MyWorkflowLibrary**.
6. Then, *enter* the complete path of the folder where you want to save your application in the **Location** box, as shown in Fig.VB-7.14. In this case, we have entered **D:\Books\Simple Steps Books\Visual Basic 2008 in Simple Steps\Code Files\Chapter 7**.
7. Then, *click* the **OK** button, as shown in Fig.VB-7.14. This closes the **New Project** dialog box and creates your sequential workflow library, as shown in Fig.VB-7.15:

Fig.VB-7.15

8. Now, *add* two **Code** activities and a **Delay** activity to your workflow.
9. Then, *change* the **Name** properties of the two **Code** activities to **ShowMessageActivity** and **ShowEndMessageActivity**, respectively, as shown in Fig.VB-7.15.
10. Now, *double-click* the **ShowMessageActivity** activity on the form design view to open the Code Editor

and *add* the following highlighted code snippet to the Code Editor:

of 5 seconds to the execution of the workflow.

12. Similarly, *double-click* the **ShowEndMessageActivity** activity on the form design view to open the Code Editor and *add* the following highlighted code snippet to the Code Editor:

a .NET assembly, named **MyWorkflowLibrary.dll**, which you can use in other .NET applications.

We have created a reusable .NET code library that contains a custom workflow, now we add a Windows Forms project to the current application that will use the custom workflow.

14. *Add* a new Windows Forms project to your current solution by right-clicking the solution in the **Solution Explorer** and selecting the **Add→New** Project option from the context menu, and *rename* the project as **WorkflowTestApplication**.
15. Then, *add* a **Button** control from the Windows Workflow v3.0 tab of the Toolbox to **Form1** and *change* its **Text** property to **Execute Workflow**, using the Properties window.

Now, we want to use the workflow that we created in the workflow library, **MyWorkflowLibrary**, in the **WorkflowTestApplication** project. To do so, we need to add a reference of the **MyWorkflowLibrary.dll** assembly to the **WorkflowTestApplication** project.

16. *Right-click* **References** under the the **WorkflowTestApplication** node in the **Solution Explorer** and *select* the **Add Reference** option, as shown in Fig.VB-7.16:

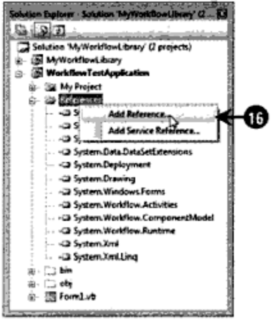
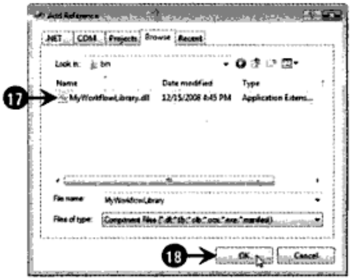


Fig.VB-7.16

This opens the **Add Reference** dialog box, as shown in Fig.VB-7.17:



Look in: bin

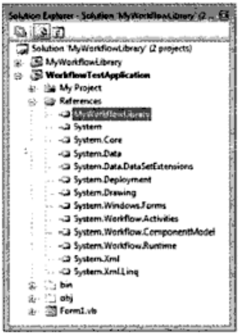
Name	Date modified	Type
MyWorkflowLibrary.dll	12/15/2008 4:45 PM	Application Extens...

File name: MyWorkflowLibrary

Files of type: Component Files (*.dll;*.tlb;*.olb;*.ocx;*.exe;*.manifest)

Fig.VB-7.17

17. Then, first *click* the **Browse** tab, and *select* the **MyWorkflowLibrary.dll** file, as shown in Fig.VB-7.17.
18. Then, *click* the **OK** button, as shown in Fig.VB-7.17. This adds the reference of the **MyWorkflowLibrary.dll** assembly to the **WorkflowTestApplication** project, as shown in Fig.VB-7.18:



```
Solution 'MyWorkflowLibrary' (2 projects)
  MyWorkflowLibrary
  WorkflowTestApplication
    My Project
    References
      MyWorkflowLibrary
      System
      System.Core
      System.Data
      System.Data.DataSetExtensions
      System.Deployment
      System.Drawing
      System.Windows.Forms
      System.Workflow.Activities
      System.Workflow.ComponentModel
      System.Workflow.Runtime
      System.Xml
      System.Xml.Linq
    bin
    obj
    Form1.vb
```

Fig.VB-7.18

19. Now, *perform* the steps 16 and 17 to add a reference of each of the following .NET assemblies to the **WorkflowTestApplication** project (as these assemblies are predefined .NET assemblies, so you need to use the **.NET** tab of the **Add Reference** dialog box to select each of these assembly):

20. Then, *add* the following **Imports** statements to **Form1.vb** file:

21. Now, *double-click* the **Execute Workflow** button on the form design view to open the Code Editor and *add* the following highlighted code snippet to the Code Editor:

22. Now, *set* the **WorkflowTestApplication** project as the startup project of the application using **Solution Explorer** and *press* the **F5** key on the keyboard to run the application. The output of the application is shown in Fig.VB-7.19:

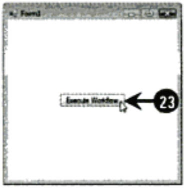


Fig.VB-7.19

23. *Click* the **Execute Workflow** button to execute the workflow (Fig.VB-7.19). This instantly displays the message box, as shown in Fig.VB-7.20:

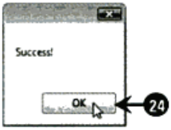

Success!

Fig.VB-7.20

24. *Click* the **OK** button in the message box (Fig.VB-7.20). This displays another message box after 5 seconds, as shown in Fig.VB-7.21:

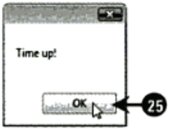

Time up!

Fig.VB-7.21

25. *Click* the **OK** button to close the message box.

Summary

In this chapter, we learned about:

- Principles of workflows
- Components of Windows Workflow Foundation
- How to develop a simple workflow application
- How to implement conditions in workflows
- How to use workflows with other applications

Introduction

Business is growing faster day by day and so is the need to store data. Data, as we know, is a collection of facts, which is generally stored in a database in the form of tables. A database is a collection of tables and each table stores large amount of data systematically in a computer, so that information can be accessed from the database quickly and efficiently whenever required. Databases that are used to relate data in multiple tables are called relational databases. Some popular relational databases are Structured Query Language (SQL) Server, Oracle, and Microsoft Access.

For retrieving and manipulating data directly from a database requires the knowledge of basic SQL commands. The person who is not familiar with SQL commands is not able to use the data stored in a database. In such a situation, most business applications provide a user-friendly interface, which helps in retrieving data from a database without the need to write SQL commands. Microsoft ActiveX Data Objects.NET (ADO.NET) is a model used by .NET applications using which you can communicate with the database directly for retrieving and manipulating data.

This chapter familiarizes you with ADO.NET, its new features and components, and also deals with the different types of data binding. In addition, you also learn to bind data to Windows Forms application and Windows Presentation Foundation (WPF).

Let's first explore ADO.NET in brief.

Introducing ADO.NET

ADO.NET is the main data access system that .NET applications use. ADO.NET uses a disconnected data architecture, which means that the data you work with is just a copy of the data in the database. Microsoft chose disconnected data architecture because of a number of reasons. In traditional client/server applications, while the application is running, you get a connection to a database and keep it open. However, maintaining these connections require a lot of server resources. When you migrate to the Internet, you should follow disconnected data architecture, instead of maintaining direct and continuous connections with the server to reduce the load on servers. Let's now focus on the new features and components of ADO.NET.

New Features in ADO.NET

The new features of ADO.NET are as follows:

- **Language-Integrated Query (LINQ):** Language-Integrated Query (LINQ) is a new innovation and one of the components of .NET Framework 3.5 that adds native data querying capabilities to .NET languages using syntax similar to that of SQL. LINQ to ADO.NET is a LINQ technology to enable querying in ADO.NET using LINQ programming model. LINQ to ADO.NET consists of two related technologies: LINQ to DataSet and LINQ to SQL. LINQ to DataSet provides faster querying of data on the contents of a DataSet. LINQ to SQL enables you to directly query SQL Server databases.
- **LINQ to DataSet:** LINQ to DataSet provides LINQ capabilities for disconnected data stored in a dataset. LINQ to DataSet makes it easier and faster to query data cached in a DataSet object. The LINQ to DataSet feature enables you to work more efficiently.
- **LINQ to SQL:** LINQ to SQL is a component of .NET 3.5 Framework that provides a run-time infrastructure for managing relational data as objects. You can use LINQ to SQL technology for translating a query into a SQL query, and then issue it against tables in a SQL Server database. LINQ to SQL supports all the key capabilities that you would expect while working with SQL. You can insert, update, and delete the information from the table.

For more information on LINQ, refer to Chapter 9, Introducing Language-Integrated Query.

Oracle. Some commonly used methods of the Connection object is Open and Close methods. The Open method is used to open a connection with the database and the Close method is used to close the connection.

- **Command:** Executes a command against the database and retrieves a **DataReader** or a **DataSet**. It also executes the **INSERT**, **UPDATE**, or **DELETE** command against the database. The base class for all Command objects is the **DbCommand** class. The Command object is represented by two classes: **SqlCommand** and **OleDbCommand**. The Command object provides three methods that are used to execute commands on the database. The **ExecuteNonQuery** method executes the commands that have no return value such as INSERT, UPDATE, or DELETE. The **ExecuteScalar** method returns a single value from a database query. The ExecuteReader method returns a result set in the form of a **DataReader** object.
- **DataReader:** Retrieves data from the database in a forward-only, read-only mode. The base class for all DataReader objects is the **DbDataReader** class. The **DataReader** object is returned as a result of calling the ExecuteReader method of the Command object.
- **DataAdapter:** Retrieves data from the database and stores data in a dataset and reflects the changes made in the dataset to the database. The base class for all **DataAdapter** objects is the **DbDataAdapter** class. The **DataAdapter** object acts as intermediary for all the communication between the database and the **DataSet** object. The Fill method of a **DataAdapter** object is used to fill a **DataTable** or **DataSet** objects with data from the database. The **DataAdapter** object commits the changes to the database by calling the Update method. The **DataAdapter** provides four properties that represent four database commands: **SelectCommand**, **InsertCommand**, **DeleteCommand**, and **UpdateCommand**.

Datasets

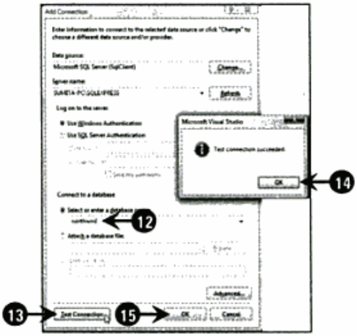
The other major component of ADO.NET is the **DataSet** object. The **DataSet** object always remains disconnected from the database, consequently reducing load on the database. A dataset is connected to a data provider using the **DataAdapter** object. The **DataAdapter** object is used as an intermediary between the dataset and the data provider. The data in the dataset can be manipulated and updated independent of the database since the dataset maintains a cached copy of the data from a database.

Table 8.2 lists the various components that make up a dataset:

11. *Select* the server name from the **Server name** combo box, as shown in Fig.VB-8.6.

By default, the radio button beside the **Use Windows Authentication** option is selected (Fig.VB-8.7). If you want to use SQL Server authentication, then *click* the radio button beside the **Use SQL Server Authentication** option, and provide user name and password for authentication. In this example, to connect to the server, we make use of Windows authentication.

12. Now, *select* the database or enter the name of the database you want to connect to in the **Select or enter a database name** combo box, as shown in Fig.VB-8.7:



Add Connection

Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

Data source:
Microsoft SQL Server (SqlClient)

Server name:
SUMITA-PC\SQLEXPRESS

Log on to the server

● Use Windows Authentication
○ Use SQL Server Authentication

Connect to a database

● Select or enter a database name:
northwind
○ Attach a database file:

Microsoft Visual Studio

Test connection succeeded.

Fig.VB-8.7

13. *Click* the **Test Connection** button. A message box appears with the message **Test connection** succeeded, as shown in Fig.VB-8.7.
14. *Click* the **OK** button of the message box to close the message box, as shown in Fig.VB-8.7.
15. Now, *click* the **OK** button of the **Add Connection** dialog box. The Add Connection dialog box closes and the data source is added to your application in the **Server Explorer**, as shown in Fig.VB-8.8:

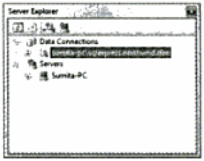


Fig.VB-8.8

Let's now see how you can establish a connection to the database using a connection string added in a code-behind file.

Using a Connection String

You saw how to create a connection using a wizard. You can also create a connection string through the code-behind file. For this, you should be aware of the syntax of the connection string for each of the data source. The connection string for your application depends upon the type of data source you are connecting to. For an OLE DB data source, such as Microsoft Access, use the **ConnectionString** property of the **OleDbConnection** class, for SQL Server, use the **ConnectionString** property of the **SqlConnection** class, and for an Oracle data source, use the **ConnectionString** property of the **OracleConnection** class.

The basic syntax of the connection string includes a series of keywords separated by semicolons. The equal sign connects each keyword and its value. The following code snippet shows a connection string used to connect to an SQL Server database:

```
Data Source=SUMITA-PC\\SQLEXPRESS;Initial Catalog=northwnd;Integrated Security=True
```

The explanation of the terms used in the previous code snippet is as follows:

- **Data Source:** Represents the name or the network address of the SQL Server instance to which you want to connect.
- **Initial Catalog:** Represents the name of the database.
- **Integrated Security:** Accepts a Booiean value, which can be either True or False. If the value is False then you need to specify the user id and password in the connection string. If the value is True, then the current Windows account credentials are used for authentication. The default value is True.

Next, let's see how to execute commands by using the Command object.

Using a Command Object

After establishing a connection with the database, you can execute commands and also return results from the database. To access a database, a data command should provide information about the connection, the SQL statement or the name of the stored procedure to execute. Listing 8.1 shows how to use the Command object:

Listing 8.1: Using the Command Object

Adding and Configuring a Data Adapter

A data adapter enables you to access data from a database in a disconnected way. Let's now learn how we can access data in a data adapter from a database, by performing the following steps:

1. *Open* the **DatabaseOperationsExample** application.
2. *Set* the **Text** property of **Form1** to **Basic Database Operations.**

Note

3. *Right-click* the **Data** tab of the Toolbox and *select* the **Choose** Items option from the context menu that appears, as shown in Fig.VB-8.9:

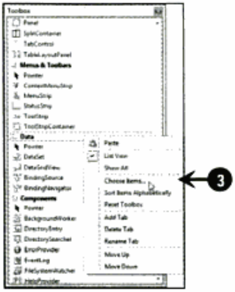
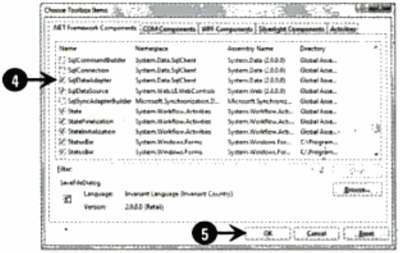


Fig.VB-8.9

The **Choose Toolbox Items** dialog box appears, as shown in Fig.VB-8.10:



Name	Namespace	Assembly Name	Directory
SqlCommandBuilder	System.Data.SqlClient	System.Data (2.0.0.0)	Global Asse...
SqlConnection	System.Data.SqlClient	System.Data (2.0.0.0)	Global Asse...
SqlDataAdapter	System.Data.SqlClient	System.Data (2.0.0.0)	Global Asse...
SqlDataSource	System.Web.UI.WebControls	System.Web (2.0.0.0)	Global Asse...
SqlSyncAdapterBuilder	Microsoft.Synchronization.D...	Microsoft.Synchroniz...	Global Asse...
State	System.Workflow.Activities	System.Workflow.Act...	Global Asse...
StateFinalization	System.Workflow.Activities	System.Workflow.Act...	Global Asse...
StateInitialization	System.Workflow.Activities	System.Workflow.Act...	Global Asse...
StatusBar	System.Windows.Forms	System.Windows.For...	C:\Program...
StatusBar	System.Windows.Forms	System.Windows.For...	C:\Program...

SaveFileDialog
Language: Invariant Language (Invariant Country)
Version: 2.0.0.0 (Retail)

Fig.VB-8.10

4. *Under* the **.NET Framework Components** tab in the **Choose Toolbox Items** dialog box, *select* the check box beside the **SqlDataAdapter** option, as shown in Fig.VB-8.10.
5. Then *click* the **OK** button to close the **Choose Toolbox** Items dialog box, as shown in Fig.VB-8.10. The Choose Toolbox Items dialog box closes and the **SqlDataAdapter** control is added to the Toolbox, as shown in Fig.VB-8.11:

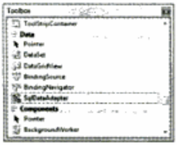
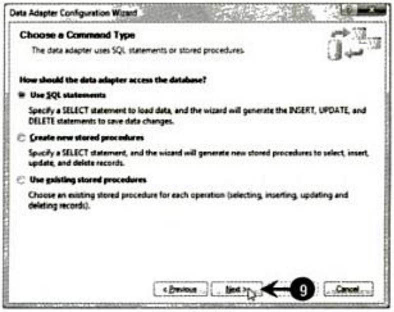


Fig.VB-8.11

6. Now, *drag* a **SqlDataAdapter** control to the form. The **Data Adapter Configuration Wizard** begins, as shown in Fig.VB-8.12:

Fig.VB-8.12

7. *Select* a data connection from the **Which data connection should the data adapter use?** combo box or create a new connection by *clicking* the New Connection button. *Clicking* the New Connection button opens the **Add Connection** dialog box (Fig.VB-8.4). In this case, we use the connection we created earlier for connecting to the **northwnd** database (Fig.VB-8.12).
8. *Click* the **Next** button, as shown in Fig.VB-8.12. The next page for the **Data Adapter Configuration Wizard** appears, which prompts the user to *select* a command type, as shown in Fig.VB-8.13:



Data Adapter Configuration Wizard

Choose a Command Type

The data adapter uses SQL statements or stored procedures.

How should the data adapter access the database?

● **Use SQL statements**

Specify a SELECT statement to load data, and the wizard will generate the INSERT, UPDATE, and DELETE statements to save data changes.

○ **Create new stored procedures**

Specify a SELECT statement, and the wizard will generate new stored procedures to select, insert, update, and delete records.

○ **Use existing stored procedures**

Choose an existing stored procedure for each operation (selecting, inserting, updating and deleting records).

Fig.VB-8.13

The page displays the following three options to select:

- **Use SQL statements:** Enables the data adapter to use a SQL statement to populate a table in a dataset. This option is by default selected.
- **Create new stored procedures:** Creates a new stored procedure to read and update a table in the database.
- **Use existing stored procedures:** Enables the data adapter to use an existing stored procedure to read and update a table in the database.

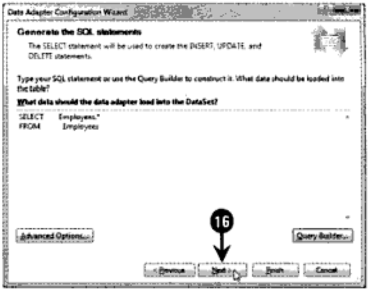
In this case, we go with the first option that is the Use SQL statements option.

9. *Click* the **Next** button, as shown in Fig.VB-8.13. The next page for the **Data Adapter Configuration Wizard** appears, which prompts the user to generate an SQL statement, as shown in Fig.VB-8.14:

Fig.VB-8.16

At the top of the Query Builder dialog box, you can see all the fields of the Employees table. You can select either some fields or all the fields from the table.

14. *Select* the check box beside the * **(All Columns)** option from the Employees table, as shown in Fig.VB-8.16.
15. *Click* the **OK** button to create the SQL statement (Fig.VB-8.16). The SQL statement appears in the **Data Adapter Configuration Wizard**, as shown in Fig.VB-8.17:



Generate the SQL statements

The SELECT statement will be used to create the INSERT, UPDATE, and DELETE statements.

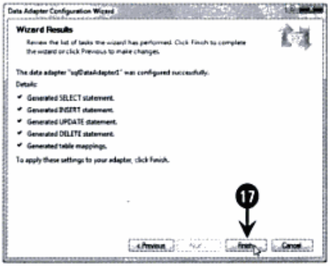
Type your SQL statement or use the Query Builder to construct it. What data should be loaded into the table?

What data should the data adapter load into the DataSet?

```
SELECT     Employees.*
FROM         Employees
```

Fig.VB-8.17

16. *Click* the **Next** button (Fig.VB-8.17). The **Data Adapter Configuration Wizard** displays the wizard results, as shown in Fig.VB-8.18:



Data Adapter Configuration Wizard

Wizard Results

Review the list of tasks the wizard has performed. Click Finish to complete the wizard or click Previous to make changes.

The data adapter "sqlDataAdapter1" was configured successfully.

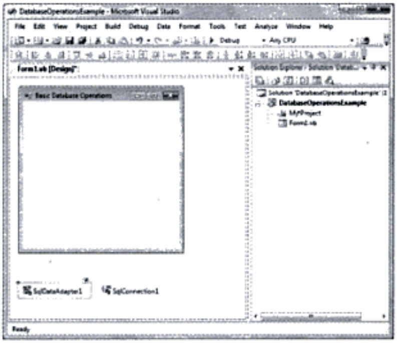
Details:

- ✓ Generated SELECT statement.
- ✓ Generated INSERT statement.
- ✓ Generated UPDATE statement.
- ✓ Generated DELETE statement.
- ✓ Generated table mappings.

To apply these settings to your adapter, click Finish.

Fig.VB-8.18

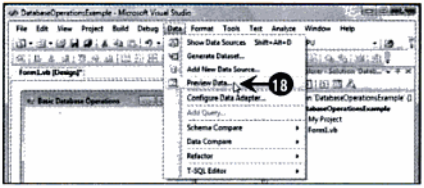
17. *Click* the **Finish** button to end the **Data Adapter Configuration Wizard**. This adds a **DataAdapter** object, **SqlDataAdapter1**, and a Connection object, **SqlConnection1**, to the component tray, as shown in Fig.VB-8.19:



DatabaseOperationsExample - Microsoft Visual Studio
File Edit View Project Build Debug Data Format Tools Test Analyze Window Help
Debug
Any CPU
Form1.vb [Design]*
Basic Database Operations
SqlDataAdapter1
SqlConnection1
Solution Explorer - Solution 'Datab...
Solution 'DatabaseOperationsExample' (1
DatabaseOperationsExample
My Project
Form1.vb
Ready

Fig.VB-8.19

18. *Click* **Data→Preview Data** on the menu bar to preview the data from the data adapter, as shown in Fig.VB-8.20:



DatabaseOperationsExample - Microsoft Visual Studio
File Edit View Project Build Debug Data Format Tools Test Analyze Window Help
Form1.vb [Design]*
Basic Database Operations
Show Data Sources Shift+Alt+D
Generate Dataset...
Add New Data Source...
Preview Data...
Configure Data Adapter...
Add Query...
Schema Compare
Data Compare
Refactor
T-SQL Editor
18
'DatabaseOperationsExample' (1
DatabaseOperationsExample
My Project
Form1.vb

Fig.VB-8.20

The **Preview Data** dialog box appe.:rs, as :

19. *Click* the **Preview** button to view the data from the database in the **Results** pane, as shown in Fig.VB-8.21.
20. *Click* the **Close** button to close the Preview Data dialog box (Fig.VB-8.21).

Creating a Dataset

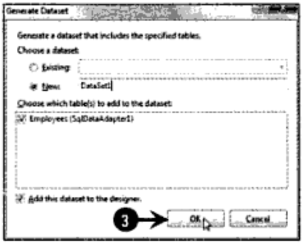
Datasets, as already explained, contains a cached copy of the tables of a database. The dataset is independent of the database and therefore the interaction between a dataset and a database is done through a data adapter. You can fill data in a dataset by using the Fill method of the data adapter.

Now, let's create a dataset by performing the following steps:

1. *Open* the **DatabaseOperationsExample** application.
2. *Click* **Data→Generate Dataset** on the menu bar, as shown in Fig.VB-8.22:

Fig.VB-8.22

The **Generate Dataset** dialog box appears, which prompts the user to select a dataset, as shown in Fig.VB-8.23:



Generate Dataset

Generate a dataset that includes the specified tables.

Choose a dataset:

○ Existing:

● New: DataSet1

Choose which table(s) to add to the dataset:

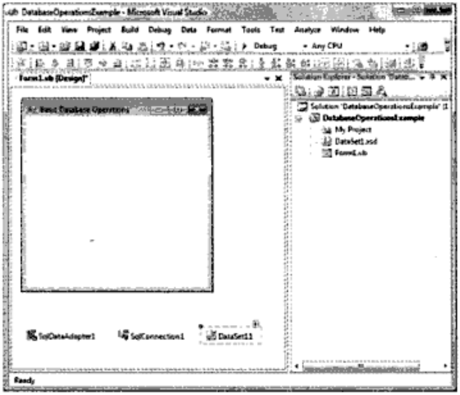
☑ Employees (SqlDataAdapter1)

☑ Add this dataset to the designer.

Fig.VB-8.23

You can select an existing dataset or create a new one. By default, the **New** radio button is selected. You can provide a new name for the new dataset or you can use the default name, **DataSet1**. The **Generate Dataset** dialog box also prompts the user to select a table to add to the dataset. By default, the table **Employees** is selected, as shown in Fig.VB-8.23.

3. *Click* the **OK** button to add the dataset to the application, as shown in Fig.VB-8.23. The dataset, **DataSet11**, is added to the component tray, as shown in Fig.VB-8.24:



DatabaseOperationsExample - Microsoft Visual Studio
File
Edit
View
Project
Build
Debug
Data
Format
Tools
Test
Analyze
Window
Help
Debug
Any CPU
Form1.vb [Design]*
Basic Database Operations
SqlDataAdapter1
SqlConnection1
DataSet11
Solution Explorer - Solution 'Datab...
Solution 'DatabaseOperationsExample' (1
DatabaseOperationsExample
My Project
DataSet1.xsd
Form1.vb
Ready

Fig.VB-8.24

Using a Data Adapter to Retrieve Data in a Dataset

So far, we have created a data adapter and a dataset. Let's now see how to display data in a **DataGridView** control using data adapter and dataset. To do so, perform the following steps:

1. *Open* the **DatabaseOperationsExample** application.
2. *Drag* a **DataGridView** control to the form from the **Data** tab of the Toolbox, as shown in Fig.VB-8.25:

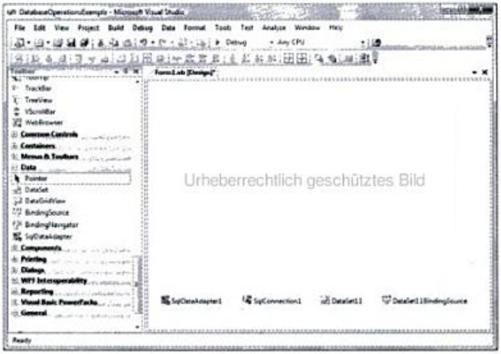


Fig.VB-8.27

4. In the Smart Tag of the **DataGridView** control, *select* the **Employees** table from the **Choose Data Source** combo box, as shown in Fig.VB-8.27. The **EmployeesBindingSource** binding source is added to the component tray, as shown in Fig.VB-8.28:

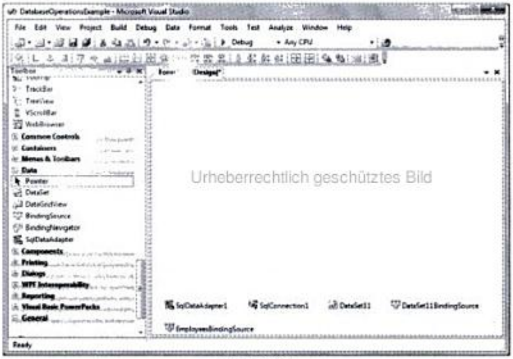


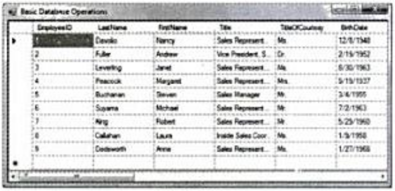
Fig.VB-8.28

5. *Resize* the form to accommodate the data of the **Employees** table.
6. *Add* the highlighted code snippet shown in Listing 8.2 to the **Load** event handler of the form to display the data of the Employees table in a **DataGridView** control:

Listing 8.2: Populating the Dataset

As shown in Listing 8.2, the Fill method is used to populate the dataset, DataSet11, with the data of the Employees table.

7. **Press** the **F5** key on the keyboard to execute the **DatabaseOperationsExample** application. The output is shown in Fig.VB-8.29:



Basic Database Operations

EmployeeID	LastName	FirstName	Title	TitleOfCourtesy	BirthDate
1	Davolio	Nancy	Sales Represent...	Ms.	12/8/1948
2	Fuller	Andrew	Vice President, S...	Dr.	2/19/1952
3	Leverling	Janet	Sales Represent...	Ms.	8/30/1963
4	Peacock	Margaret	Sales Represent...	Mrs.	9/19/1937
5	Buchanan	Steven	Sales Manager	Mr.	3/4/1955
6	Suyama	Michael	Sales Represent...	Mr.	7/2/1963
7	King	Robert	Sales Represent...	Mr.	5/29/1960
8	Callahan	Laura	Inside Sales Coor...	Ms.	1/9/1958
9	Dodsworth	Anne	Sales Represent...	Ms.	1/27/1966

Fig.VB-8.29

Types of Data Binding in Windows Forms

Data binding means to bind controls to data from data sources. Data binding can be used to bind either a particular field in a table or bind the entire table to a control. For example, you can bind a text box to the **ProductName** field of a table or bind the entire table to a data grid view.

There are two types of data binding in Windows Forms: simple data binding and complex data binding. Let's see them in detail next.

Simple Data Binding

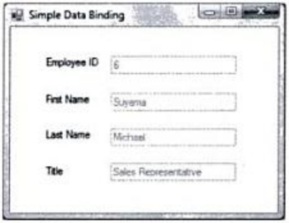
Simple data binding allows you to display one data element, such as a field's value from a table, in a control. In Visual Basic, you can simple bind any property of a control to a data value. For example, you can bind a **Text** property of a text box, **Size** and **Image** properties of a picture box or the **BackColor** property of a label to a data source. You can bind a property of a control to a data source by using the **DataBindings** property of the control.

In simple terms, you can say that simple data binding is the ability of a control to bind a single data element. Let's see an example of simple data binding. For that, perform the following steps:

1. *Create* an application, named **WindowsFormsDataBinding**.
2. *Set* the **Text** property of **Form1** to Simple Data Binding.
3. *Drag* the controls listed in Table 8.3 on Form1 from the Toolbox and also *set* their properties, as given in Table 8.3:

5. Now, add the highlighted code snippet shown in Listing 8.3 to the Load event handler of **Form1** to display the details of the employee whose **EmployeeID** is **6**:

6. *Press* the **F5** key on the keyboard to execute the **WindowsFormsDataBinding** application. The output is shown in Fig.VB-8.30:



Employee ID: 6
First Name: Suyama
Last Name: Michael
Title: Sales Representative

Fig.VB-8.30

As shown in Fig.VB-8.30, the details of the employee whose ID is 6, have been displayed on text boxes.

Using the BindingContext Class

The **BindingContext** class in Windows Forms application provides information about the binding and binding elements that is required to build the channel for data binding. You use the **BindingContext** class to access the data bindings in a control. The inheritance hierarchy of the **BindingContext** class is shown here:

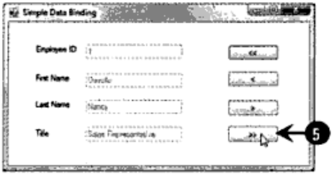
```
    System.Object
      System.Windows.Forms.BindingContext
```

Each object that inherits from the **Control** class can have a single **BindingContext** object. Using this object, you can access the data bindings in a form, which allows you to set the current record displayed in simple-bound controls, using the **Position** property. Let's see an example of simple data binding using the **BindingContext** class. For that, perform the following steps:

1. *Open* the application, named **WindowsFormsDataBinding**.

displays the next record of the dataset. Finally, for **Button4**, the **Position** property is set to a value one less than the number of records to display the last record of the dataset.

4. *Press* the **F5** key on the keyboard to execute the **WindowsFormsDataBinding** application. The output is shown as in Fig.VB-8.31, which displays the details of the first record, by default:



Simple Data Binding

Employee ID 1

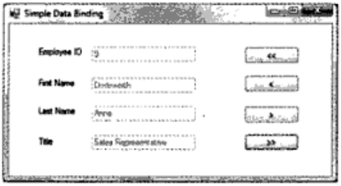
First Name Davolio

Last Name Nancy

Title Sales Representative

Fig.VB-8.31

5. *Click* the **>>** button, as shown in Fig.VB-8.31. The last record of the dataset is displayed, as shown in Fig.VB-8.32:



Simple Data Binding

Employee ID 9

First Name Dodsworth

Last Name Anne

Title Sales Representative

Fig.VB-8.32

Complex Data Binding

Complex data binding enables binding of a control to more than one data element, such as more than one record in a database. Some of the controls that support complex data binding are: **DataGridView**, **ComboBox**, **ListBox**, and **CheckedListBox**. Complex data binding can be performed using the following properties:

- **DataSource:** Represents the data source, typically a dataset, such as DataSet11.
- **DataMember:** Represents the data member you want to work with, in the data source, typically a table in a dataset, such as the Customers table in the **northwnd** database. The **DataGridView** control uses the **DataMember** property to determine which table to display.
- **DisplayMember:** Represents the field you want a control to display, such as the customer's id, **CustomerID**. The **ListBox** control uses the **DisplayMember** and **ValueMember** properties instead of a **DataMember** property for data binding.
- **ValueMember:** Represents the field you want the control to return in properties, such as returning the customer ID by using the **SelectedValue** property. The **ListBox** control uses the **DisplayMember** and **ValueMember** properties, instead of a **DataMember** property.

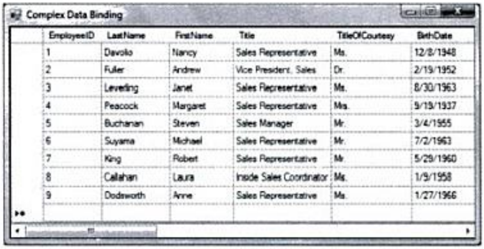
Let's see an example of complex data binding through code. For that, perform the following steps:

1. *Open* the application, named **WindowsFormsDataBinding.**
2. *Right-click* the **WindowsFormsDataBinding** application in the **Solution Explorer** and *select* **Add→Windows Form** from the context menu to add a new form to the application, as shown in Fig.VB-8.33:

Listing 8.5: Displaying Complex Data Binding

source and the **DataMember** property is used to set the table name from which the employee details need to be displayed.

9. *Set* **Form2** as the startup form for the application.
10. *Press* the **F5** key on the keyboard to execute the **WindowsFormsDataBinding** application. The output is shown in Fig.VB-8.35:



Complex Data Binding

EmployeeID	LastName	FirstName	Title	TitleOfCourtesy	BirthDate
1	Davolio	Nancy	Sales Representative	Ms.	12/8/1948
2	Fuller	Andrew	Vice President, Sales	Dr.	2/19/1952
3	Leverling	Janet	Sales Representative	Ms.	8/30/1963
4	Peacock	Margaret	Sales Representative	Mrs.	9/19/1937
5	Buchanan	Steven	Sales Manager	Mr.	3/4/1955
6	Suyama	Michael	Sales Representative	Mr.	7/2/1963
7	King	Robert	Sales Representative	Mr.	5/29/1960
8	Callahan	Laura	Inside Sales Coordinator	Ms.	1/9/1958
9	Dodsworth	Anne	Sales Representative	Ms.	1/27/1966

Fig.VB-8.35

Data Binding in Windows Presentation Foundation

As we know WPF makes it easier to design robust and visually appealing user interface. The added capability that WPF provides is the data binding. In WPF, you can perform data binding using the Framework Code, **eXtensible** Application Markup Language (XAML) or a combination of both. You can bind the WPF controls and their properties to make data binding flexible and easy. For more information on WPF, refer to *Chapter 6, Introducing Windows Presentation Foundation*.

As in other types of applications, such as the Window Forms and the ASP.NET Web application, WPF also needs to have a target and a source for data binding. You can select the public property of the control in WPF to bind your data, which includes properties of other controls, or any data source such as the **northwnd** database. The target of the binding can be accessible to public property of the control, for example, the Text property of the **TextBox** control. In all, you can say that data binding is one of the most powerful features included in WPF.

Data Flow Directions

WPF data binding supports different types of binding modes between the target and the source. The data flow in a binding can either move from the target to the source or from the source to the target. You should specify the mode of data binding while binding the data in a WPF application. The Mode property defines the binding mode that determines how the data will flow between the source and the target. There are four types of binding mode available in WPF: **OneTime** data binding, **OneWay** data binding, **TwoWay** data binding, and **OneWayToSource** data binding. Let's learn about the functioning of each of these in detail.

OneTime Data Binding

In **OneTime** data binding in WPF, the data flows from the source to the target. The binding occurs only once when the application is started or the data context changes. The best time to use the **OneTime** binding is when your data source does not implement the **INotifyPropertyChanged** interface. For example, when you do not have to change the data or add any other data to your database, you can use **OneTime** binding in your application. The **OneTime** binding can only retrieve the data but cannot update data in your database.

OneWay Data Binding

In **OneWay** data binding in WPF, the data flows from the source to the target. This type of binding is useful for read-only data as it is not possible to change the data from the user interface. The **OneWay** binding mode in WPF is the default binding mode.

TwoWay Data Binding

The data in **TwoWay** data binding in WPF moves in both directions, that is, from source to the target and from target to source. In **TwoWay** data binding, you can make changes to the data in the user interface. In this binding, the data is sent to the target, and if there is any change in the target property value, it is sent back to the source. You can use the **TwoWay** binding when you want to change the data in the user interface which is reflected in the data source.

OneWayToSource Data Binding

In **OneWayToSource** data binding, if the target property changes, the source property is updated automatically. You can use the **OneWayToSource** binding when you want to change the data and get it updated in the source.

Declaration of Data Binding in WPF

You can declare binding in WPF in different ways and formats. You can create binding in the XAML format using the markup file of a WPF application. You can also create binding through code using the code-behind file of a WPF application. The third way to create binding in WPF is by specifying the Path property. You can specify the source value that you want to bind by using the Path property. Let's see how you create data binding in a WPF application in these three ways.

Using XAML

You can bind the WPF application in the XAML format by specifying the Binding property. Binding is the markup extension. When you use Binding property as an extension to declare binding, the declaration consists a series of clauses. The clauses are in the form of Name = Value pairs, where Name is the name of the Binding property and Value is the value you are setting for the property. You should note that when you are creating binding in the XAML format, the Binding property must be attached to the specific dependency property of the target object. The following code snippet explains the basic syntax by using the Binding property in the XAML format:

```
    <TextBox Text="{Binding Source={StaticResource myDataSource}, Path=ContactName}"/>
```

In the preceding code snippet, the Text property of the TextBox control is using the Binding property. The data binding in WPF provides you a simple and consistent way for the applications to present and interact with data. In WPF, you establish a binding using the Binding property. Each binding in WPF has four components: binding target, binding source, target property, and a path to the source value to use. Let's see a binding example in WPF using XAML:

1. *Create* a WPF application, named **WPFDataBinding**.
2. *Set* the title of **Window1** to **Binding** using **XAML** using the **Title** property.
3. *Add* the code given in Listing 8.6 to the **Window1.xaml** file:

Listing 8.6: Creating Data Binding Using XAML

property to a **ListBox** control's selected value. In the code, the Binding property within the **Ellipse.Fill** property sets the binding from Ellipse control to **ListBox** control by specifying control ID in the **ElementName** property and the Path property holds the value of the selected item in the list box.

4. *Press* the **F5** key on the keyboard to execute the **WPFDataBinding** application. The output is shown in Fig.VB-8.36:

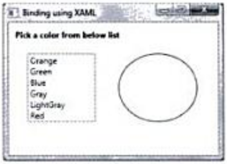


Fig.VB-8.36

5. Now, *click* the **Green** color in the list box, this fills the ellipse with the green color, as shown in Fig.VB-8.37:

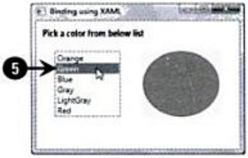


Fig.VB-8.37

Using the Code-Behind File

Another way to specify the binding is to set the properties directly on the Binding object in code. The **FrameworkElement** class and the **FrameworkContentElement** class, both expose the **SetBinding** method. You can call the **SetBinding** method directly in your application to bind the control in code. The following code snippet explains the basic syntax of how to bind your WPF application in code:

```
me.MyText.SetBinding(TextBlock.TextProperty, Binding1)
```

In this code snippet, you are binding the Text property of the **TextBlock** control.

Let's now see a binding example in WPF using the code-behind file:

1. *Open* the WPF application, named **WPFDataBinding**.
2. *Add* a new window to the application by *right-clicking* the application name in the **Solution Explorer** and *selecting* the **Add→Window** from the context menu. The **Add New Item** dialog box appears with the name **Window2.xaml** in the **Name** text box. When you click the **OK** button, the **Window2.xaml** file is added to the application.
3. *Set* the title of **Window2** to **Binding** using code-behind file using the **Title** property.
4. *Add* the code given in Listing 8.7 to the **Window2.xaml** file:

Listing 8.7: Adding Code for the Window2.xaml File

As shown in Listing 8.7, the **Window2.xaml** file is same as **Window1.xaml** file except for the **Binding** property used for the **Ellipse** control in the **Window1.xaml** file. Here, we create the binding through code.

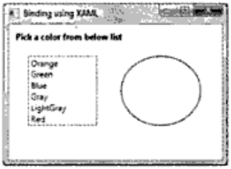
5. *Add* the code given in Listing 8.8 to the **Window2.xaml.vb** file:

Listing 8.8: Creating Data Binding Using the Code-behind File

the binding source. Similarly, the Path property is used to set the property of the binding source and finally the **SetBinding** property is used to bind the target control to the source.

6. *Set* **Window2.xaml** as the startup object by double-clicking the **Application.xaml** file in the **Solution Explorer** and changing the **StartupUri** attribute of the **Application.xaml** file to **Window2.xaml**.

7. *Press* the **F5** key on the keyboard to execute the **WPFDataBinding** application. As a result, the output is shown in Fig.VB-8.38:



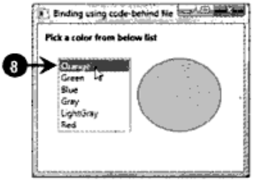
Binding using XAML

Pick a color from below list

Orange
Green
Blue
Gray
LightGray
Red

Fig.VB-8.38

8. Now, *click* the **Orange** color in the list box. This fills the ellipse with the orange color, as shown in Fig.VB-8.39:



Binding using code-behind file

Pick a color from below list

Orange
Green
Blue
Gray
LightGray
Red

Fig.VB-8.39

Using the Path Property

You can also use the Path property to specify the source value that you want to bind in a WPF application. The Path property is the name of the property of the source object used for binding. For example, you can bind the Text property of the **TextBox** control. You can also bind an attached property of a control using the Path property. For example, to bind the attached property **DockPanel.Dock**, the syntax that you should use with the Path property is given here:

```
Path = (DockPanel.Dock)
```

You can also bind the property of a control to a particular field of the database using the Path property. The syntax for this is given here:

```
Path=Employees.FirstName
```

In the preceding code snippet, you can bind the control to the FirstName column of the Employees table.

Let's now see a binding example in WPF using the Path property:

1. *Open* the WPF application, named **WPFDataBinding**.
2. *Add* a new window to the application by *right-clicking* the application name in the Solution Explorer and *selecting* the **Add→Window** from the context menu. A window with the name **Window3.xaml** is added to the application.
3. *Set* the title of **Window3** to **Binding using the Path Property** by using the **Title property**.
4. *Add* the code given in Listing 8.9 to the **Window3.xaml** file:

Listing 8.9: Creating Data Binding Using the Path Property

property to a ListBox control's selected value. In the code, notice that the DataContext attribute for the Ellipse control is set to the binding definition. The Binding property within the Ellipse control sets the binding for Ellipse control and ListBox control. Binding for the Ellipse control is set by specifying control ID in the ElementName property and binding for the ListBox control is set through the value of the selected item in the list box using the Path property.

5. *Set* **Window3** as the startup object.
6. *Press* the **F5** key on the keyboard to execute the **WPFDataBinding** application. The output is shown in Fig.VB-8.40:

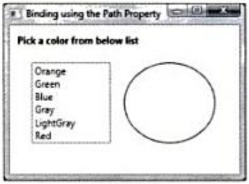


Fig.VB-8.40

7. Now, *click* the **Blue** color in the list box. This fills the ellipse with the blue color, as shown in Fig.VB-8.41:

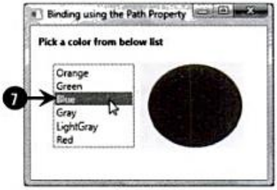


Fig.VB-8.41

Binding Sources in WPF

In data binding, the source refers to the object that you obtain data from. WPF supports four types of binding source: Common Language Runtime (CLR) object, ADO.NET data, **eXtensible** Markup Language (XML) data, and **DependencyObject**. For CLR object, the data binding works as long as the binding engine is able to access the source property using reflection. The XML data fails to bind when it does not have permission to access the given data. You can always bind to an ADO.NET object and **DependencyObject**.

Binding to CLR Objects

In WPF, you can bind to the public properties, or the sub-properties of any CLR object. The binding object in WPF uses the CLR reflection to retrieve the values of the properties. When you are using the CLR object for data binding in WPF, you should implement the **INotifyPropertyChanged** interface. This interface helps you to update the target when the source property changes by implementing the **INotifyPropertyChanged** interface. This helps in ensuring that the data used in binding stays current.

If the source object implements a proper notification mechanism, the target is updated automatically. You can also use the **UpdateTarget** method to update the target property to provide property change notification.

Let's see an example to implement CLR object binding. For this, perform the following steps:

1. *Create* a WPF application, named **BindingtoCLRObject**.
2. *Set* the title of **Window1** to Binding to a CLR Object using the **Title** property.
3. *Add* the code given in Listing 8.10 to the **Window1.xaml** file:

Listing 8.10: Preparing the User Interface

After adding the preceding code to the **Window1.xaml** file, add a class to your application.

4. For adding the class, *right-click* the application name in the **Solution Explorer** and *select* the **Add→Class** from the context menu, as shown in Fig.VB-8.42:

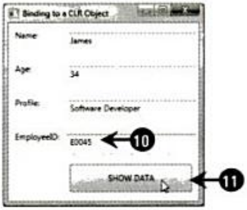
Listing 8.11: Complete Code for the **Data.vb** File

In the preceding code, a value is added to the user interface that you have created in Listing 8.10.

8. Now, add the code given in Listing 8.12 to **Window1.xaml**.vb:

Listing 8.12: Complete Code for the Window1.xaml.vb File

9. *Press* the **F5** key on the keyboard to execute the **BindingtoCLRObject** application. A window appears wherein you can enter employee details, as shown in Fig.VB-8.44:



Binding to a CLR Object

Name: James

Age: 34

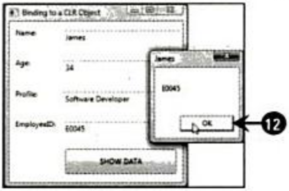
Profile: Software Developer

EmployeeID: E0045

Fig.VB-8.44

10. *Enter* employee details in the text boxes, as shown in Fig.VB-8.44.

11. *Click* the **SHOW DATA** button. A message box appears, as shown in Fig.VB-8.45:



Binding to a CLR Object

Name: James

Age: 34

Profile: Software Developer

EmployeeID: E0045

James

E0045

Fig.VB-8.45

12. *Click* the **OK** button in the message box to close the message box.

Binding to ADO.NET Objects

You can also bind an ADO.NET object to a WPF application. For example, you can bind a data table to a WPF application. You can implement the **IBindingList** interface to provide change notifications. The **IBindingList** interface provides features to support both simple and complex data binding to a data source. While binding your WPF application to an ADO.NET object, the first step is to create a connection. After establishing the connection, the adapter, which executes the SQL statement to retrieve the record from the database, is created. The result is stored in the data table of the dataset by calling the Fill method of the adapter. This result is then displayed in the WPF control.

To bind a WPF application to an ADO.NET object, first you need to create a connection string to bind your WPF application to an ADO.NET object. Then you also need to bind your controls by specifying the **ItemSource** and **ItemTemplate** properties. To display a particular column data, you should specify the **DisplayMemberBinding** property. Let's create an application in which you can bind an ADO.NET object to a WPF application. For this, perform the following steps:

1. *Create* a WPF application, named **BindingtoADO.NET**.
2. *Set* the title of **Window1** to **Binding to ADO.NET** using the Title property.
3. *Add* the code given in Listing 8.13 to the **Window1.xaml** file:

Listing 8.13: Preparing the User Interface

With the help of the code given in Listing 8.13, you can prepare the interface on which you display your data.

4. For database connectivity, import two namespaces, **System.Data** and **System.Data.SqlClient,** in the **Window1.xaml.vb** file.
5. Now, add the code given in Listing 8.14 to the **Window1.xaml.vb** file.

Listing 8.14: Code for Data Binding

database by using the DataContext object.

6. *Press* the **F5** key on the keyboard to execute the **BindingtoADO.NET** application. The output is shown in Fig.VB-8.46:

Fig.VB-8.46

7. *Click* the **Get Data** button. The employee details are displayed in a **ListView** control, as shown in Fig.VB-8.46.

Summary

In this chapter, you learned about:

- ❑ Features and components of ADO.NET
- ❑ Types of data binding
- ❑ Implementation of data binding in Windows Forms
- ❑ Implementation of data binding in WPF

Introduction

Language Integrated Query (LINQ) is a new component of .NET Framework 3.5. The basic function of LINQ is to add native data querying capabilities to .NET Framework using syntax similar to that of Structured Query Language (SQL). LINQ allows you to define statements that interrogate a data source to yield a requested result set. LINQ is an attempt to provide a consistent way of obtaining and manipulating the data. You can use LINQ directly within the VB programming language entities called query expressions. These query expressions are based on numerous query operators that have been designed to work in a manner similar to that of SQL. LINQ defines the set of query operators as the operators used to query, project, and filter the data. The difference, however, is that the query expressions can be used to interact with numerous types of data, even with the data that does not belong to a relational database. LINQ integrates the query syntax within a VB program, which makes it possible to access different data sources with the same syntax. LINQ makes it possible by offering an abstraction layer.

In this chapter, you will learn about LINQ queries, standard query operators used in LINQ, LINQ to ADO.NET, anonymous types, and lambda expressions.

LINQ Queries

A query is an expression that is used to retrieve data from a data source. It specifies, sorts, and groups the data that is retrieved from a data source. LINQ queries are written in a specialized query language. In the past, you had to learn different languages for different data sources, for example, SQL for relational database and XQuery for Extensible Markup Language (XML). A LINQ query simplifies this situation by providing a consistent syntax to work with data across various kinds of data sources and with data of different formats.

You can use various clauses, such as **From**, **Where**, **Order By**, and **Select**, with a LINQ query. These are predefined clauses that are used for the execution of a LINQ query.

The basic syntax of a LINQ query starts with the **From** clause and ends with the **Select** or **Group By** clause. In addition, you can use the **Where**, **Order By**, and **Order By Descending** clauses to perform additional functions, such as filtering data.

The following are the three basic steps to execute a LINQ query:

- Obtain the data source. The data source can be either a SQL database or an XML file.
- Create the query.
- Execute the query.

Now, let's learn how to execute a simple LINQ query.

Executing a Simple LINQ Query

A LINQ query is executed in a **For Each** statement. The **For Each** statement in VB requires the **IEnumerable** or **IEnumerable(Of T)** interface. A LINQ query contains three clauses: **From, Where,** and **Select.** The **From** clause specifies the data source, the **Where** clause applies the filter, and the **Select** clause specifies the type of result.

Now, let's create a new Windows Forms application, **LINQQuery**, in which you can use a simple LINQ query to retrieve data. Perform the following steps to do this:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2008→ Microsoft Visual Studio 2008** to open the Visual Studio 2008 IDE.
2. In the Visual Studio 2008 IDE, *click* **File→New→Project** on the menu bar to open the **New Project** dialog box.
3. In the **New Project** dialog box, *select* the **Visual Basic→Windows** option in the **Project types** pane and the **Windows Forms Application** option in the **Templates** pane.
4. *Enter* **LINQQuery** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** box.
5. *Click* the **OK** button. The **LINQQuery** application is created.

6. *Add* a **ListView** control and a **Button** control to **Form1** and *change* the **Text** property of the **Button1** to **Click**, as shown in Fig.VB-9.1:

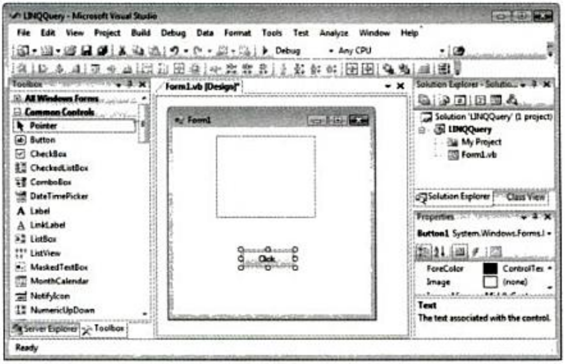


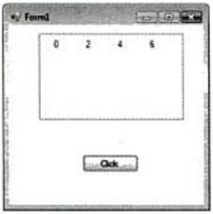
Fig.VB-9.1

7. *Add* the code given in Listing 9.1 to the **Click** event handler of **Button1**:

Listing 9.1: Code for the **Button1_Click** Event Handler

Listing 9.1 shows how the three parts of a LINQ query are expressed in the source code. The code uses an integer array as the data source.

8. *Press* the **F5** key to run the application and *click* the **Click** button. The output of the application is shown in Fig.VB-9.2:



```
0    2    4    6
```

Fig.VB-9.2

Note

The Standard Query Operators

The standard query operators are clauses that are used to create and refine data based on the query and the needs of an application. Standard query operators provide querying capabilities, including filtering, projection, aggregation, and sorting. The standard query operator in LINQ is an Application Programming Interface (API) that enables querying of any .NET array or collection. You can use these standard query operators to perform an operation on a sequence of data.

Standard query operators differ in the time they take to execute the query. The time taken by a query depends on whether the query returns a single value or a sequence of values. The methods that return a single value execute immediately. The methods that return a sequence of values reschedule the execution of the query and return an output.

A list of all standard query operators used in LINQ is listed in Table 9.1:

Aggregate operator	**Computes a single value from a collection**

Now, let's know about each of the standard query operators in detail.

The Sorting Operators

The sorting operators in LINQ order the elements of a sequence based on one or more attributes. You can sort the data with one specific attribute and perform primary sorting on the elements. You can then specify the second sorting criterion and sort the elements within the primary sorted group. The different sorting operators are the **Order By** and **Order By Descending** clauses.

The sorting functionality is achieved by using the **Order By** clause. The sorting operator can sort data either in ascending order or in descending order. The default behavior of the **Order By** clause is to sort data in ascending order. If you want to order your data in descending order, you need to use the **Order By Descending** clause.

Now, let's create a new Windows Forms application, **SortingOperator**, where you can use the **Order By** and **Order By Descending** clauses to sort data. Perform the following steps to do this:

1. *Repeat* steps 1 to 3 as discussed earlier in the case of the **LINQQuery** application.
2. *Enter* **SortingOperator** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **SortingOperator** application is created.

8. *Click* the **Descending** button, as shown in Fig.VB-9.4. The data in the list view is sorted in the descending order, as shown in Fig.VB-9.5:

Fig.VB-9.5

The Set Operators

The set operators in LINQ refer to the operators that are used to produce a result set. The result is based on the presence or absence of the equivalent elements that are present within the same or separate collection. The **Distinct** clause and the **Union**, **Intersect**, and **Except** methods are classified as the set operators.

The **Distinct** clause removes duplicate values from a collection. The **Union** method returns all the elements in the two sets. The **Intersect** method returns the elements that appear in each of the two collections. The **Except** method returns the elements of the first collection that do not appear in the second collection.

Now, let's create a new Windows Forms application, **SetOperator**, where you can use the **Distinct** clause to remove duplicate values from a set of data. Perform the following steps to do this:

1. *Repeat* steps 1 to 3 as discussed earlier in the case of the **LINQQuery** application.
2. *Enter* **SetOperator** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **SetOperator** application is created.
4. *Add* a **ListView** control and a **Button** control to **Form1** and change the **Text** property of **Button1** to **Distinct**, as shown in Fig.VB-9.6:

Fig.VB-9.6

5. *Add* the code given in Listing 9.4 to the **Click** event handler of **Button1**:

Listing 9.4: Code for the **Button1_Click** Event Handler

Ihe code in Listing 9.4 removes the duplicate numbers in the set and displays the result when you click the **Distinct** button.

6. *Press* the **F5** key to run the application and click the **Distinct** button. The output of the application is shown in Fig.VB-9.7:

Fig.VB-9.7

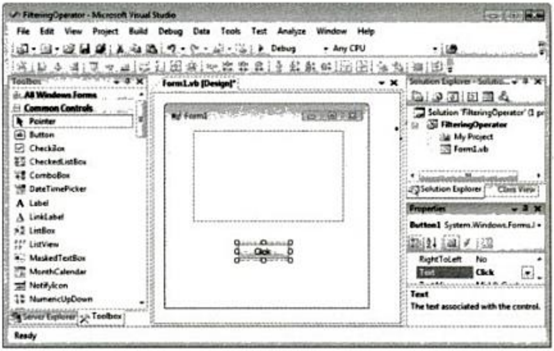
Filtering Operator

Filtering, as the name suggests, refers to the operation of filtering the result set so that it contains only those elements that satisfy a specified condition. The **Where** clause is used as the filtering operator in LINQ. It filters a sequence based on the given condition. The **Where** clause is used in a query expression to specify which elements from the data source will be returned in the query expression.

Now, let's create a Windows Forms application, **FilteringOperator**, where you can use the **Where** clause. Perform the following steps to do this:

1. *Repeat* steps 1 to 3 as discussed earlier in the case of the **LINQQuery** application.
2. *Enter* **FilteringOperator** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **FilteringOperator** application is created.

4. *Add* a **ListView** control and a **Button** control to **Form1** and *change* the **Text** property of **Button1** to **Click**, as shown in Fig.VB-9.8:



RightToLeft No
Text Click

Text
The text associated with the control.

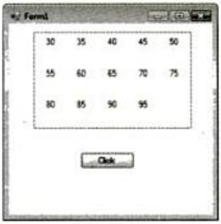
Fig.VB-9.8

5. *Add* the code given in Listing 9.5 to the **Click** event handler of **Button1**:

Listing 9.5: Code for the **Button1_Click** Event Handler

The code in Listing 9.5 generates the numbers that are greater than 25, which is the condition specified in the **Where** clause.

6. *Press* the **F5** key to run the application and *click* the **Click** button. The output of the application is shown in Fig.VB-9.9:



```
30   35   40   45   50

55   60   65   70   75

80   85   90   95
```

Fig.VB-9.9

The Quantifier Operators

The quantifier operators return a Boolean value if the elements of a sequence satisfy a specific condition. The operators that are classified as the quantifier operators are the **Any**, **All**, and **Contains** methods.

The **Any** method determines whether or not any elements in a sequence satisfy a condition. This method enumerates the source sequence and returns **True** if any element satisfies the condition. The enumeration of the source sequence terminates as soon as the result is known. The **ArgumentNullException** exception is thrown if the argument is null.

The **All** method determines whether or not all the elements in the sequence satisfy the given condition. This method enumerates the source sequence and returns **True** if no element fails the condition specified in the query. The **All** method returns a **True** value for an empty sequence. The **ArgumentNullException** exception is thrown if any argument is null.

The **Contains** method checks the source sequence to determine whether or not it contains the specified element. When the matching element is found, the **Contains** method returns the result.

The Projection Operators

The projection operators refer to the operators that are used to transform an object into a new form that consists of only those properties that are subsequently used. The projection operators are used to transform an object into a new object of a different type. By using the projection operators, you can construct a new object of a different type that is built from each object. The **Select** clause and the **SelectMany** method are the projection operators used in LINQ.

The **Select** clause performs a projection over a sequence and projects the value that is based on a transform function. The **Select** clause in LINQ performs the same function as performed by the **Select** statement in SQL. The **Select** clause specifies which elements are to be retrieved.

The **SelectMany** method projects a sequence of values that are based on a transform function and then retrieves them into one sequence.

Now, let's create a new Windows Forms application, **ProjectionOperator**, where you can use the **Select** clause. Perform the following steps to do this:

1. *Repeat* steps 1 to 3 as discussed earlier in the case of the **LINQQuery** application.
2. *Enter* **ProjectionOperator** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **ProjectionOperator** application is created.
4. Add a **ListView** control and a **Button** control to **Form1** and change the **Text** property of **Button1** to **Select Clause**, as shown in Fig.VB-9.10:

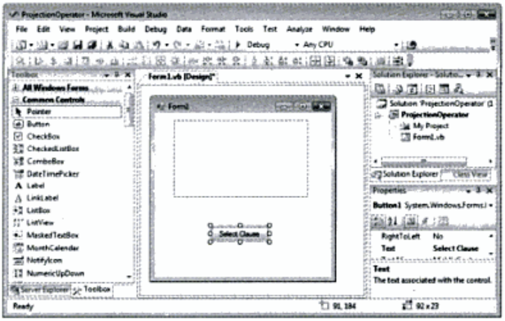


Fig.VB-9.10

5. *Add* the code given in Listing 9.6 to the **Click** event handler of **Button1**:

Listing 9.6: Code for the **Button1_Click** Event Handler

6. *Press* the **F5** key to run the application and click the **Select Clause** button. The output of the application is shown in Fig.VB-9.11:

In Fig.VB-9.11, each word from the string of words is displayed. The complete string is displayed in three lines.

Note

The Partitioning Operators

The partitioning operators in LINQ are used to divide an input sequence into two sections, without rearranging the elements, and then returning the result set with one of the sections that satisfies the given condition. The **Take**, **Skip**, **Take While**, and **Skip While** clauses are referred to as the partitioning operators.

The **Take** clause takes the elements up to a specified position in a sequence. The **Take While** clause takes the elements based on the specified function until an element does not satisfy the given condition. The **Skip** clause skips elements up to the specified position in the sequence. The **Skip While** clause skips the elements based on the given function until an element does not satisfy the given condition.

Now, let's create a new Windows Forms application, **PartitioningOperator**, where you can use the **Take** and **Skip** clauses. Perform the following steps to do this:

1. *Repeat* steps 1 to 3 as discussed earlier in the case of the **LINQQuery** application.
2. *Enter* **PartitioningOperator** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **PartitioningOperator** application is created.
4. *Add* a **ListView** control and two **Button** controls to **Form1** and *change* the **Text** property of **Button1** to **Take** and the **Text** property of **Button2** to **Skip**, as shown in Fig.VB-9.12:

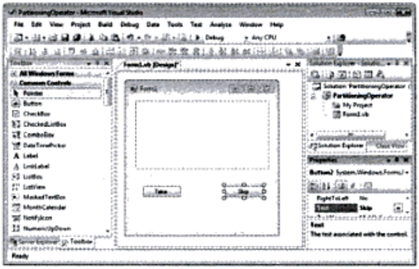


Fig.VB-9.12

5. *Add* the code given in Listing 9.7 to the **Click** event handler of **Button1**:

6. *Add* the code given in Listing 9.8 to the **Click** event handler of **Button2**:

Listing 9.8: Code for the **Button2_Click** Event Handler

7. *Press* the **F5** key to run the application and *click* the **Take** button. This displays the first five elements from the input sequence in the list view, as shown in Fig.VB-9.13:

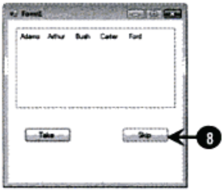
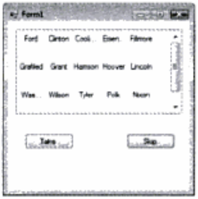


Fig.VB-9.13

8. *Click* the **Skip** button, as shown in Fig.VB-9.13. This displays all the elements excluding the first four elements from the input sequence in the list view, as shown in Fig.VB-9.14:



```
Ford     Clinton  Cooli...  Eisen...  Filmore
Garfield Grant    Harrison  Hoover    Lincoln
Was...   Wilson   Tyler     Polk      Nixon
```

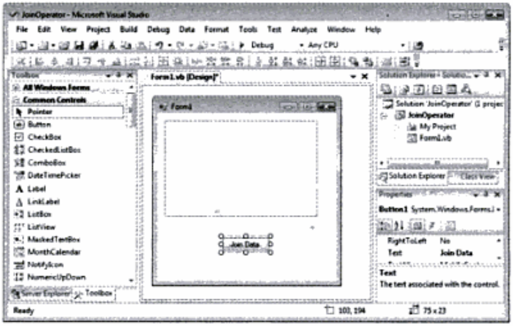
Fig.VB-9.14

The Join Operators

The join operators in LINQ are used to join objects in one data source with objects that share a common attribute in another data source. The join operators provided in LINQ are the **Join** and **Group Join** clauses. The **Join** clause implements an inner join that is a type of join in which only those objects that have a match in other data sets are returned. The **Group Join** clause joins two sequences based on a **keyselector** function and groups the results.

Now, let's create a new Windows Forms application, **JoinOperator**, where you can use the **Join** clause. Perform the following steps to do this:

1. *Repeat* steps 1 to 3 as discussed earlier in the case of the **LINQQuery** application.
2. *Enter* **JoinOperator** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **JoinOperator** application is created.
4. Add a **ListView** control and a **Button** control to **Form1** and *change* the **Text** property of **Button1** to **Join Data**, as shown in Fig.VB-9.15:



Button1 System.Windows.Forms.I
RightToLeft No
Text Join Data

Text
The text associated with the control.

Fig.VB-9.15

5. *Click* the Smart Tag of the **ListView** control and *select* **Tile** from the **View** combo box, as shown in Fig.VB-9.16:

Fig.VB-9.16

6. Now, *add* the code shown in Listing 9.9 to **Form1.vb** file, which is the code behind file of **JoinOperator** application:

Listing 9.9: Code for the **Form1.vb** File

In Listing 9.9, the Customer data and the Order data are joined by using the **Join** clause.

7. *Press* the **F5** key to run the application and *click* the **Join Data** button. The output of the application is shown in Fig.VB-9.17:

The Grouping Operators

The grouping operators in LINQ are used to put data into groups so that the elements in each group share a common attribute. The **Group By** clause is the grouping operator used in LINQ. The **Group By** clause returns a sequence of the **IGrouping(Of TKey, TElement)** objects that contain zero or more items that match the key for the group.

Now, let's create a new Windows Forms application, **GroupingOperator**, where you can use the **Group** clause. Perform the following steps to do this:

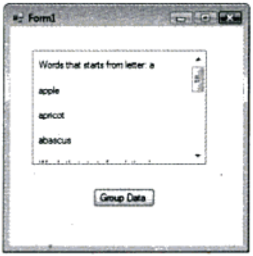
1. *Repeat* steps 1 to 3 as discussed earlier in the case of the **LINQQuery** application.
2. *Enter* **GroupingOperator** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **GroupingOperator** application is created.
4. Add a **ListView** control and a **Button** control to **Form1** and change the **Text** property of **Button1** to **Group Data**, as shown in Fig.VB-9.18:

Fig.VB-9.18

5. *Click* the Smart Tag of the **ListView** control and select **Tile** from the **View** combo box, as done earlier in the **JoinOperator** example.
6. *Add* the code given in Listing 9.10 to the **Click** event handler of **Button1**:

Listing 9.10: Code for the **Button1_Click** Event Handler

7. *Press* the **F5** key to run the application and *click* the **Group Data** button. The output of the application is shown in Fig.VB-9.19:



Words that starts from letter: a

apple

apricot

abascus

Fig.VB-9.19

The Generation Operators

The generation operators help ~~in creating~~ a new sequence of values. The generation operators are the **DefaultIfEmpty**~~**, Empty, Range**~~, and **Repeat** methods.

The **DefaultIfEmpty** method replaces an empty collection with a default single collection. The **Empty** method generates an empty collection. The **Range** method generates a collection that contains a sequence of numbers. The **Repeat** method generates a collection that contains at least one repeated value. The **Range** method throws an **ArgumentOutOfRangeException** exception if the count is less than 0, or if the expression, **start + count - 1**, evaluates to a value that is greater than the maximum value.

The Element Operators

The element operators in LINQ return just one element. The **ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single**, and **SingleOrDefault** methods are referred to as element operators. The **ElementAt** method returns the element at a specified index in a collection. The **ElementAtOrDefault** method returns the element at a specified index in a collection or the default value if the index is out of range. The **First** method returns the first element of the collection or the first element that satisfies the given condition. The **FirstOrDefault** method returns the first element of a collection or the first element that satisfies a given condition. It returns a default value if there is no such element. The **Last** method returns the last element of the collection or the last element that satisfies a given condition. The **LastOrDefault** method returns the last element of the collection or the last element that satisfies the given condition. It returns a default value if there is no matching element. The **Single** method returns the only element of the collection that satisfies the given condition. The **SingleOrDefault** method returns the only element of the collection that satisfies the condition. It returns a default value if there is no such element

An **ArgumentNullException** exception is thrown if any argument is null. An **InvalidOperationException** exception is thrown if no element matches the condition or the source sequence is empty.

The Conversion Operators

The conversion operators convert a collection to an array. They change the type of input objects. The different conversion operators are the **ToSequence**, **ToArray**, **ToList**, **ToDictionary**, **ToLookup**, **OfType**, and **Cast** methods.

The **ToSequence** method simply returns the source argument by changing it to the **IEnumerable(Of T)** interface. The **ToArray** method enumerates the source sequence and returns an array containing the elements of the sequence. The **ToList** method enumerates the source sequence and returns an object of the **List(Of T)** interface type, containing the elements of the sequence. The **ToDictionary** method lists the source sequence and evaluates the **keySelector** and **elementSelector** functions for each element to produce the element key and value of the source sequence. The **ToLookup** method implements one-to-many dictionaries that map the key to the sequence of values. The **OfType** method allocates and returns an enumerable object that captures the source argument. The **Cast** method also allocates and returns an enumerable object that captures the source argument.

The Aggregate Operators

The aggregate operators compute a single value from a collection. The different aggregate operators are the **Aggregate**, **Average**, **Count**, **LongCount**, **Max**, **Min**, and **Sum** methods. The **Aggregate** method calculates the sum value of the values in the collection. The **Average** method calculates the average value of the collection of the values. The **Count** method counts the elements in the collection. The **LongCount** method counts the elements in a large collection. The **Max** method determines the maximum value in a collection. The **Min** method determines the minimum value in the collection. The **Sum** method calculates the sum of values in the collection.

LINQ to ADO.NET

LINQ to ADO.NET is the term that describes the database-centric aspects of LINQ. LINQ to ADO.NET consists of two separate technologies, LINQ to DataSet and LINQ to SQL.

LINQ to DataSet is a set of extensions to the standard ADO.NET DataSet programming model that allows **DataSet**, **DataTable**, and **DataRow** objects to be a natural target for the LINQ query expression. LINQ to DataSet provides richer, optimized querying over datasets.

LINQ to SQL allows you to interact with a relational database by removing the ADO.NET data types through the use of entity classes. LINQ to SQL enables you to directly query SQL Server database schemas.

Now, let's know about each of them in detail.

LINQ to SQL

LINQ to SQL is a component of .NET Framework 3.5 and is specifically designed to work with an SQL server database. It allows you to write queries to retrieve and manipulate data from the SQL server. In other words, using LINQ to SQL, you can perform various operations, such as retrieving data from the database, or inserting, updating, and deleting information from a table in the database. Visual Basic 2008 provides you the functionality to create LINQ to SQL classes from the existing database. It also provides a simple way to bind the controls in your forms to your database.

LINQ to SQL creates an Object-Relational Mapping (ORM) layer between the tables in the SQL database and the objects in a Visual Basic program. With the help of LINQ to SQL ORM mapping, the classes that match the database tables are created automatically from the database itself and you can start using the classes immediately.

Now, let's create a new Windows Forms application, **LINQtoSQL**, where you can implement LINQ to SQL. Perform the following steps to do this:

1. *Repeat* steps 1 to 3 as discussed earlier in the case of the **LINQQuery** application.
2. *Enter* **LINQtoSQL** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** box.
3. *Click* the **OK** button. The **LINQtoSQL** application is created.

After creating the interface of **Form1**, the next step in creating a LINQ to SQL application is to add a **LINQ to SQL** class. Perform the following steps to add a **LINQ to SQL** class:

8. *Right-click* the project name **LINQtoSQL** in the **Solution Explorer** and *select* **Add→New Item** from the context menu that appears, as shown in Fig.VB-9.23:

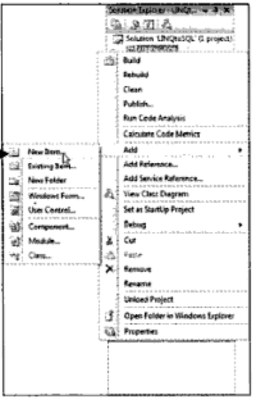
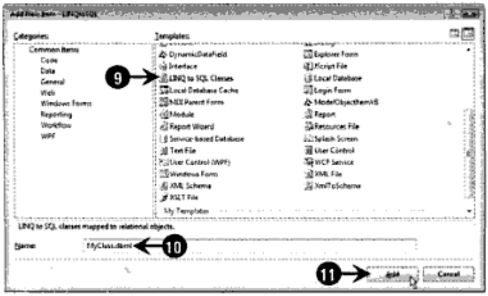


Fig.VB-9.23

The **Add New Item** dialog box opens, as shown in Fig.VB-9.24:



LINQ to SQL classes mapped to relational objects.

Name: MyClass.dbml

Fig.VB-9.24

9. In the **Add New Item** dialog box, select **LINQ to SQL Classes** in the **Templates** pane, as shown in Fig.VB-9.24.
10. *Provide* the name **MyClass.dbml** in the **Name** text box, as shown in Fig.VB-9.24.
11. *Click* the **Add** button, as shown in Fig.VB-9.24. The LINQ to SQL class, **MyClass.dbml**, is added to the **LINQtoSQL** project, opening the **Object Relational Designer** window, as shown in Fig.VB-9.25:

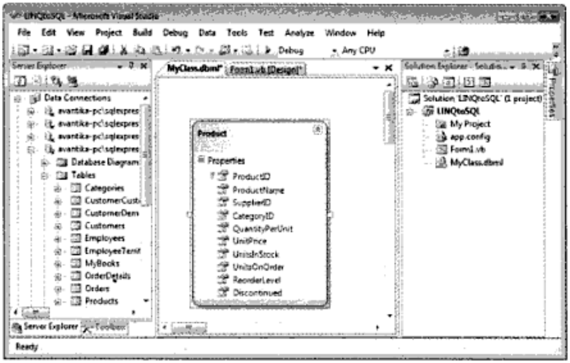
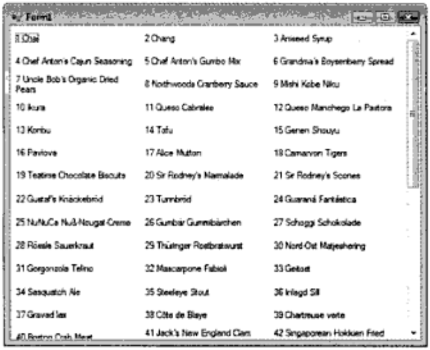


Fig.VB-9.25

12. Now, *drag* a table(s) on the **Object Relational Designer** window from the **Server Explorer**, as shown in Fig.VB-9.25. In this case, we have added the **Products** table of the **Northwind** database.
13. *Click* **Build→Build Solution** on the menu bar.
14. Now, the code given in Listing 9.11 to the **Load** event handler of **Form1**:

Listing 9.11: for the **Form1_Load** Event Handler

15. *Press* the key to run the application. The output of the application is shown in Fig.VB-9.26:



```
1 Chai                          2 Chang                          3 Aniseed Syrup
4 Chef Anton's Cajun Seasoning  5 Chef Anton's Gumbo Mix         6 Grandma's Boysenberry Spread
7 Uncle Bob's Organic Dried     8 Northwoods Cranberry Sauce     9 Mishi Kobe Niku
Pears
10 Ikura                        11 Queso Cabrales                12 Queso Manchego La Pastora
13 Konbu                        14 Tofu                          15 Genen Shouyu
16 Pavlova                      17 Alice Mutton                  18 Carnarvon Tigers
19 Teatime Chocolate Biscuits   20 Sir Rodney's Marmalade        21 Sir Rodney's Scones
22 Gustaf's Knäckebröd          23 Tunnbröd                      24 Guaraná Fantástica
25 NuNuCa Nuß-Nougat-Creme      26 Gumbär Gummibärchen           27 Schoggi Schokolade
28 Rössle Sauerkraut            29 Thüringer Rostbratwurst       30 Nord-Ost Matjeshering
31 Gorgonzola Telino            32 Mascarpone Fabioli            33 Geitost
34 Sasquatch Ale                35 Steeleye Stout                36 Inlagd Sill
37 Gravad lax                   38 Côte de Blaye                 39 Chartreuse verte
40 Boston Crab Meat             41 Jack's New England Clam       42 Singaporean Hokkien Fried
```

Fig.VB-9.26

Introduction

Imagine a scenario where you have built a software and now your friend also wants the same software, but how are you going to make it available? Definitely you are not going to develop the same application again. Here comes the need for deployment.

Deployment is the process that makes software available for use by just installing it on the computer. In the scenario stated above, we create setup files and then install the software on the user's computer.

Visual Basic 2008 applications are designed to be deployed and installed with the Windows installer program, which uses Microsoft Installer (.msi) files. In addition, Visual Basic 2008 also enables you to deploy your applications by using another technique called the ClickOnce deployment.

In this chapter, we will see how to create .msi files for applications. To install an application, you just need to copy and execute the .msi file on the user machine and Installer Wizard does the rest for you in a series of steps.

Deploying Applications by Using Windows Installer

Windows Installer allows you to deploy a Windows application by creating a Windows Installer Package. The installer package has an extension of .msi and it contains the application, any dependent files, and registry entries. After creating an application, all you need to do is to transfer the .msi file to the target machine and then double-click the .msi file to install it. Before installing the application on the target machine, ensure that the target machine supports Windows Installer and .NET Framework, so that your application can function. You can create .msi installer files with **Setup and Deployment** projects in Visual Basic 2008.

Let's now follow these steps to develop and deploy an application named **FillColor**:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2008→Microsoft Visual Studio 2008** to open the Visual Studio 2008 IDE.
2. In the Visual Studio 2008 IDE, *click* the **File→New→Project** menu item to open the **New Project** dialog box.
3. In the **New Project** dialog box, *select* the **Visual Basic→Windows** option in the **Project types** pane and the **Windows Forms Application** option in the **Templates** pane.
4. *Specify* the name as **FillColor** in the **Name** text box and an appropriate location for the application in the **Location** box.
5. *Click* the **OK** button.
6. *Select* **Form1** in the design view and *set* the **Text** property of **Form1** as **Fill Color**, as shown in Fig.VB-10.1:

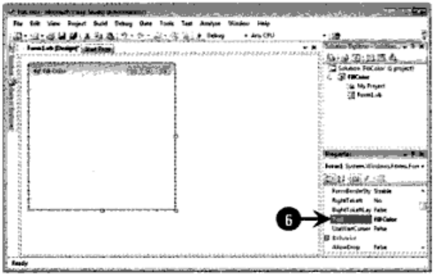


Fig.VB-10.1

7. *Drag* and *drop* the **PictureBox, Button, ProgressBar, Timer,** and **ColorDialog** controls from the **Toolbox** to the form, as shown in Fig.VB-10.2:

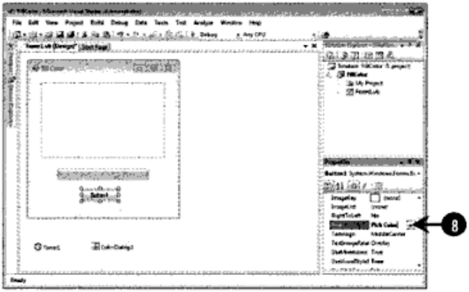


Fig.VB-10.2

8. *Set* the **Text** property of the button as **Pick Color**, as shown in Fig.VB-10.2.
9. *Upload* an image in the PictureBox control through its **Image** property and *set* its **BackColor** property to **ButtonHighlight**, as shown in Fig.VB-10.3:

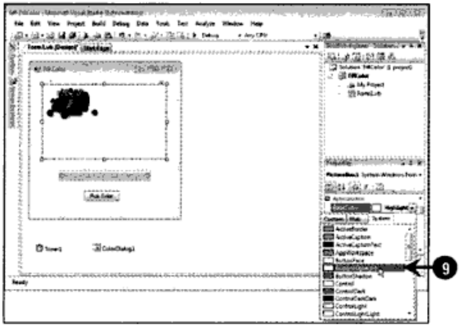
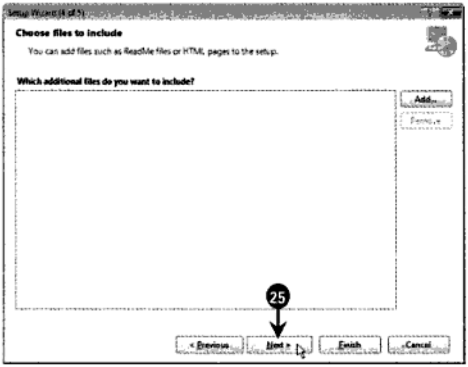


Fig.VB-10.3

10. Now, in the design view, *double-click* the **Pick Color** button and *enter* the following code snippet on the **Click** event of the button:

When the user clicks on the **Pick Color** button, the preceding code opens a **Color** dialog box. When the user selects any color from it and clicks the **OK** button, the **Timer** control is enabled.

11. Now, again in the design view, *double-click* the **Timer** control and *enter* the following code snippet on the **Tick** event of the timer:



Setup Wizard (4 of 5)

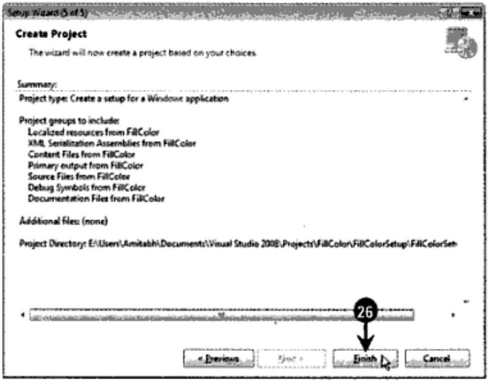
Choose files to include

You can add files such as ReadMe files or HTML pages to the setup.

Which additional files do you want to include?

Fig.VB-10.11

25. In this page, you can include other files to be deployed, such as readme.txt files, licensing agreements, and so on. In this case, we are not including such files, so *click* the **Next** button (Fig.VB-10.11) to move to the **Create Project** page of the **Setup Wizard**, as shown in Fig.VB-10.12:



Setup Wizard (5 of 5)

Create Project

The wizard will now create a project based on your choices.

Summary:

Project type: Create a setup for a Windows application

Project groups to include:
Localized resources from FillColor
XML Serialization Assemblies from FillColor
Content Files from FillColor
Primary output from FillColor
Source Files from FillColor
Debug Symbols from FillColor
Documentation Files from FillColor

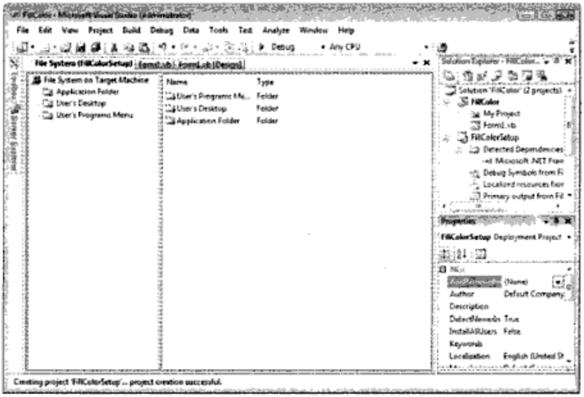
Additional files: (none)

Project Directory: E:\Users\Amitabh\Documents\Visual Studio 2008\Projects\FillColor\FillColorSetup\FillColorSet

Fig.VB-10.12

26. This is the final page of the **Setup Wizard**. *Click* the **Finish** button (Fig.VB-10.12) to create the installer file.

Once the setup is complete, you can see a new project added to your solution named **FillColorSetup**, as shown in Fig.VB-10.13:

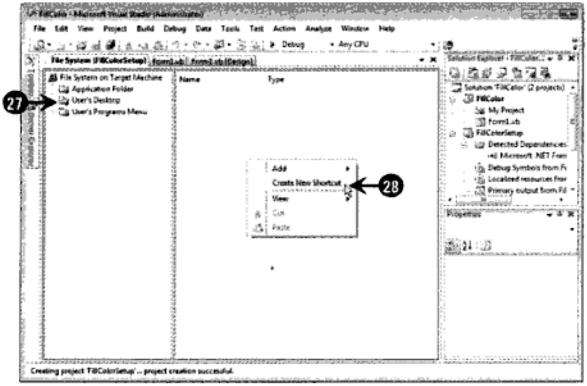


Creating project 'FillColorSetup'... project creation successful.

Fig.VB-10.13

In the **Properties** window, you can set various properties for your setup project, such as **ManufactureURL, ProductCode, and ProductName.** You can also define the items to be created on the user desktop or user program menu, such as a shortcut for the application.

27. *Select* the **User's Desktop** folder under **File System on Target Machine**, as shown in Fig.VB-10.14:



Creating project 'FillColorSetup'... project creation successful.

Fig.VB-10.14

28. *Right-click* the right pane and *select* **Create New Shortcut** from the context menu, as shown in Fig.VB-10.14. The **Select Item in Project** dialog box appears, as shown in Fig.VB-10.15:

32. Now, *select* the U Machine (Fig.VB-10.18), and then *repeat* the st ıu.

Fig.VB-10.18

33. Now, *select* the **Application Folder** under **File System on Target Machine** and *set* the **AlwaysCreate** property to **True**, as shown in Fig.VB-10.19:

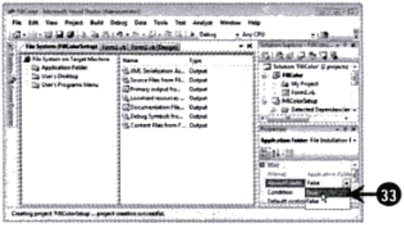

File Edit View Project Build Debug Data Tools Test Analyze Window Help
File System (FillColorSetup)
File System on Target Machine
Application Folder
User's Desktop
User's Programs Menu
Name
Type
Solution 'FillColor' (2 projects)
FillColor
My Project
Form1.vb
FillColorSetup
Detected Dependencies
Properties
Application Folder File Installation F
AlwaysCreate
False
True
Condition
Creating project 'FillColorSetup'... project creation successful.
33

Fig.VB-10.19

34. Similarly, *set* the **AlwaysCreate** property to **True** also for the **User's Desktop** and **User's Programs Menu** folders, as shown in Fig.VB-10.20:

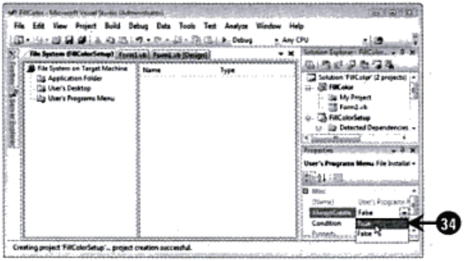

File Edit View Project Build Debug Data Tools Test Analyze Window Help
File System (FillColorSetup)
Form1.vb
Form1.vb [Design]
File System on Target Machine
Application Folder
User's Desktop
User's Programs Menu
Name
Type
Solution Explorer - FillColor
Solution 'FillColor' (2 projects)
FillColor
My Project
Form1.vb
FillColorSetup
Detected Dependencies
Properties
User's Programs Menu File Installat
(Name)
User's Programs M
AlwaysCreate
False
True
Condition
Creating project 'FillColorSetup'... project creation successful.
34

Fig.VB-10.20

35. *Select* the **Build→Build FillColorSetup** menu item when the **FillColorSetup** project is developed, as shown in Fig.VB-10.21:

38. *Click* the **Next** button (Fig.VB-10.23) to move to the **Select Installation Folder** page of the wizard, as shown in Fig.VB-10.24:

39. In this page, *b* and then *click*
the **Next** butto

40. *Click* the **Next**

IN SIMPLE STEPS

Visual Basic 2008

VISUAL BASIC 2008 IN SIMPLE STEPS is a book that helps you to learn Visual Basic using Visual Studio 2008. Precision, an easy-to-understand style, real-life examples in support of the concepts and practical approach in presentation are some of the features that make the book unique in itself. The text in the book is presented in such a way that it is equally helpful to beginners as well as professionals.

The book covers:

- Introduction to .NET Framework 3.5 and Visual Studio 2008
- Fundamentals of Visual Basic 2008 programming language
- Working with Windows Forms and common Windows controls
- Windows Presentation Foundation (WPF) and its controls
- Windows Workflow Foundation (WF)
- Working with databases in Visual Basic 2008
- Language Integrated Query (LINQ)
- Deployment of applications in Visual Basic 2008
- Windows Communication Foundation (WCF)

Published by:

dreamtech PRESS

Dreamtech Press
19-A, Ansari Road, Daryaganj,
New Delhi-110002
Tel.: 91-11-23284212, 23243075
Fax: 91-11-23243078
Email: feedback@dreamtechpress.com
http://www.dreamtechpress.com

ISBN: 978-81-7722-918-9

KOGENT
Solutions Inc.

SPECIAL INDIAN PRICE

Rs. 169/-

International Price $9.99