Evangelos Petroutsos

MASTERING

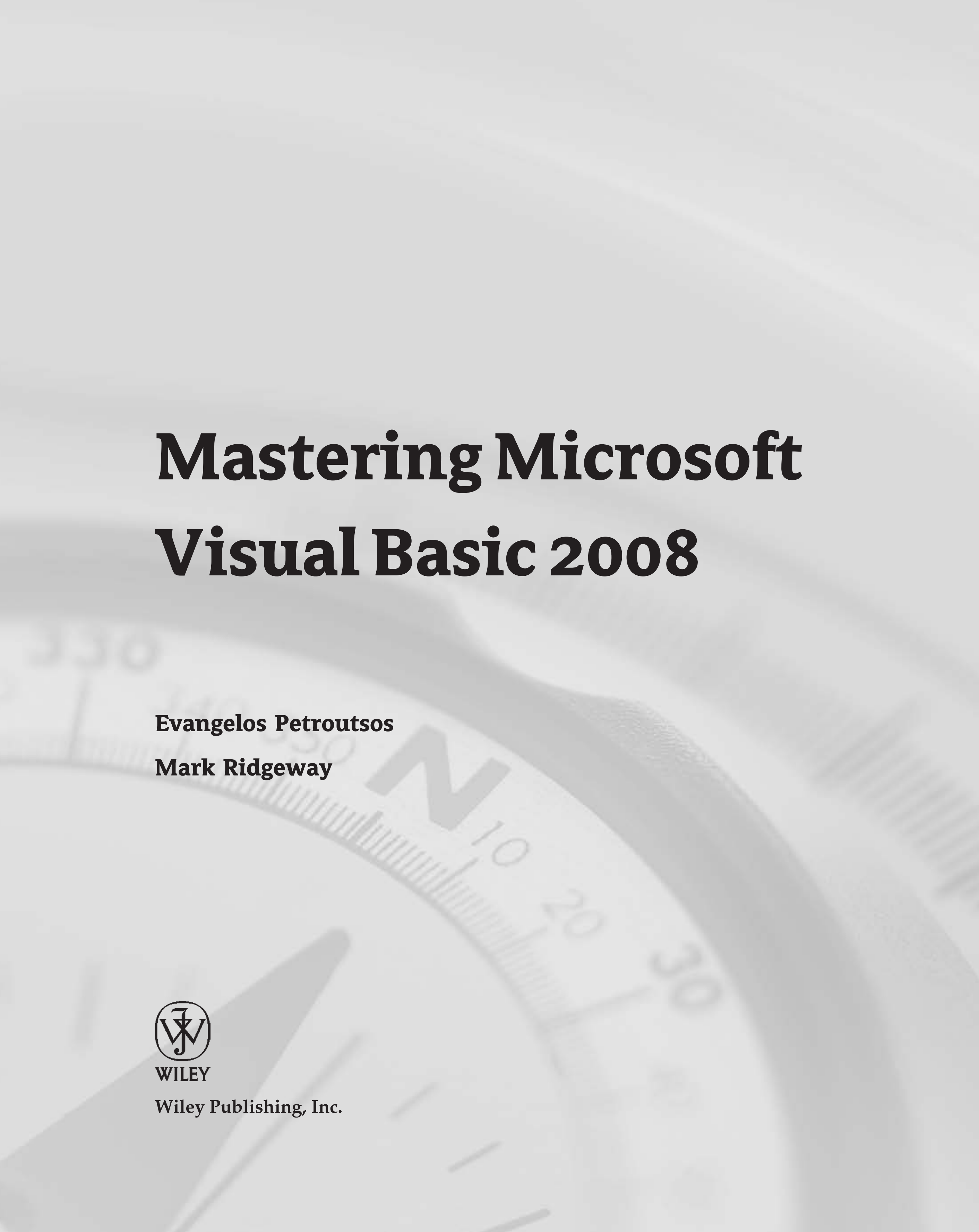
Microsoft®

Visual Basic® 2008

Work with the .NET Framework 3.5

Build Rich Client Applications with Visual Basic

SYBEX SERIOUS SKILLS.



Mastering Microsoft Visual Basic 2008

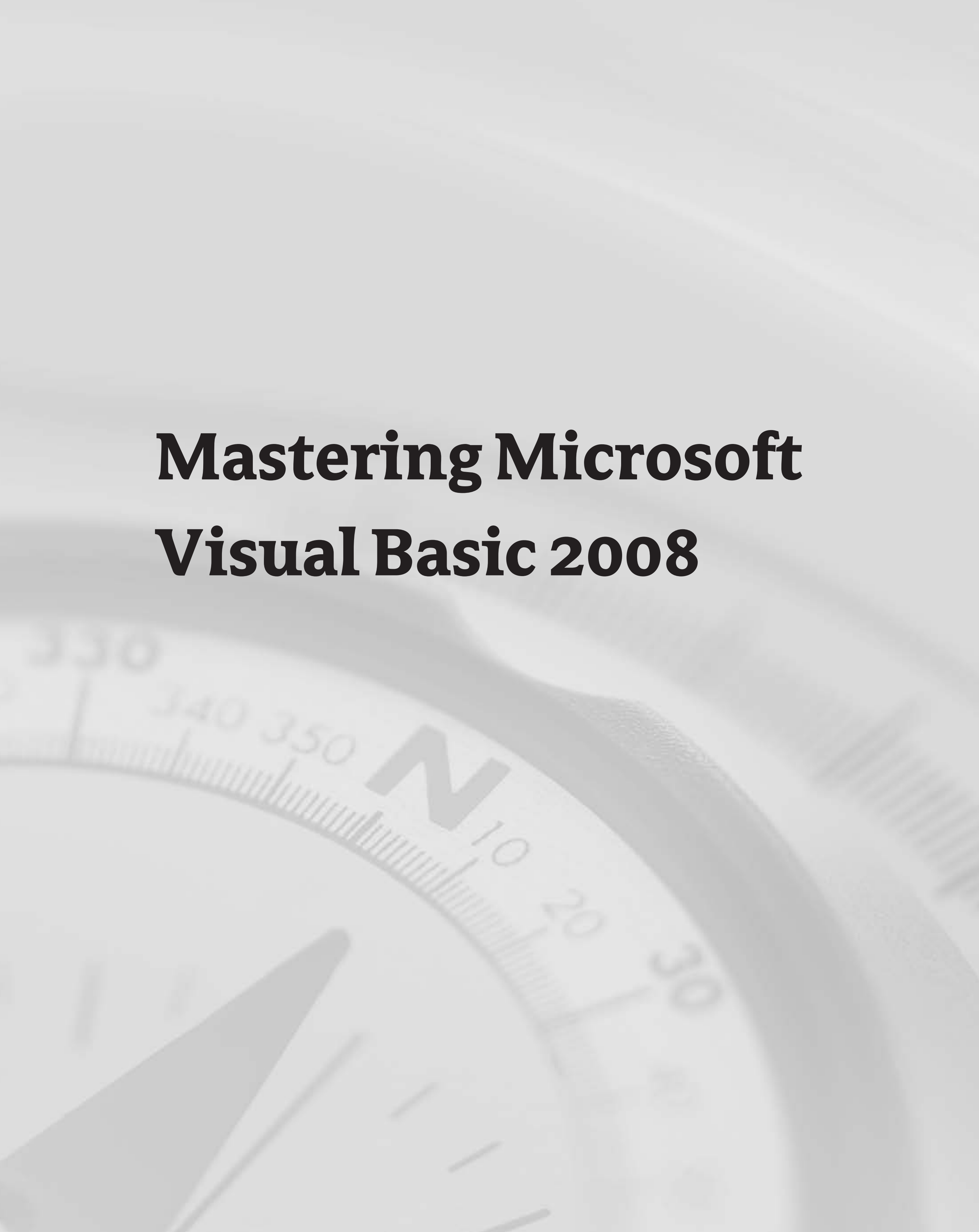
Evangelos Petroutsos

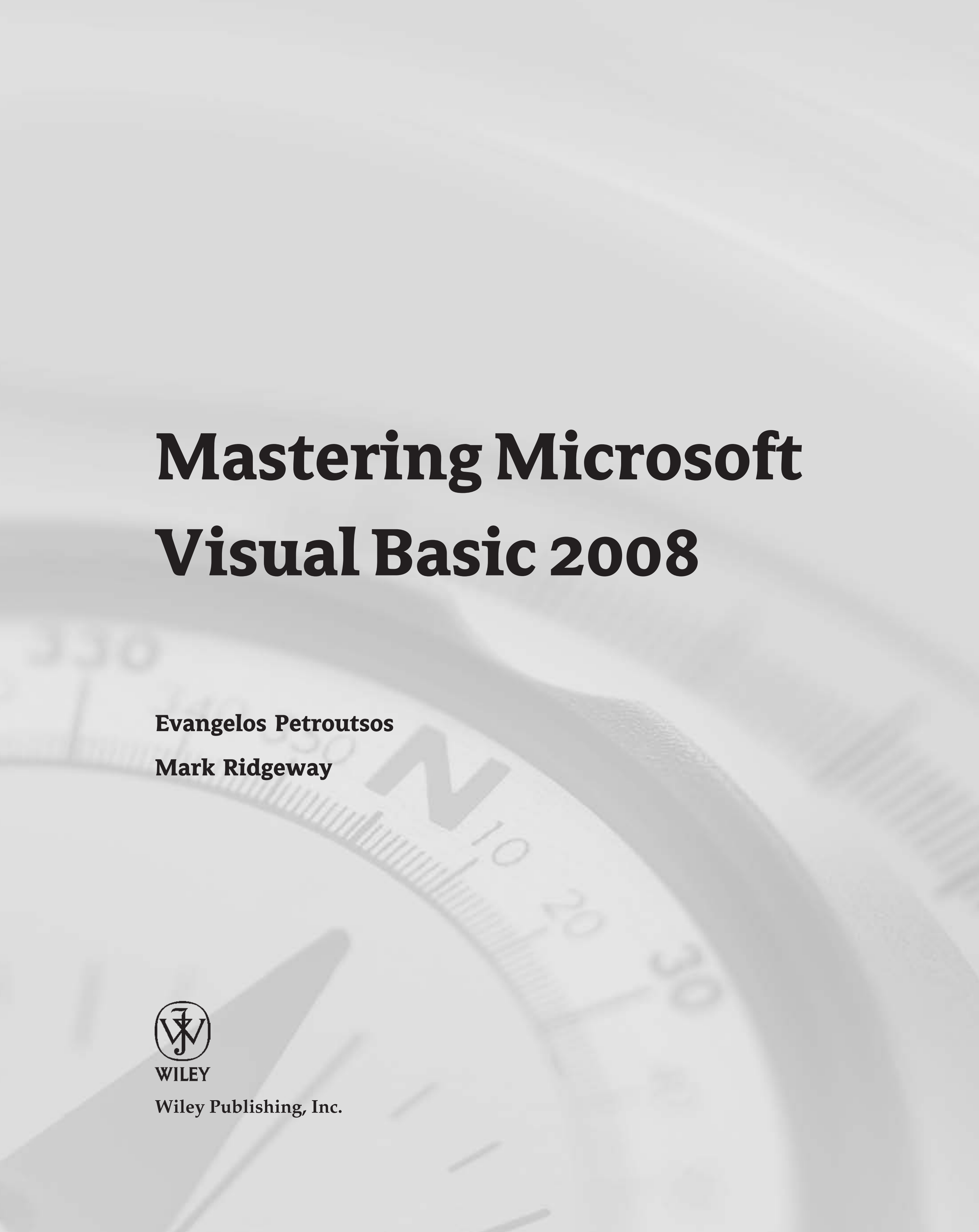
Mark Ridgeway

WILEY

Wiley Publishing, Inc.

Mastering Microsoft Visual Basic 2008





Mastering Microsoft Visual Basic 2008

Evangelos Petroutsos

Mark Ridgeway

WILEY

Wiley Publishing, Inc.

Acquisitions Editor: Thomas Cirtin
Development Editor: Toni Zaccarini Ackley
Technical Editor: John Mueller
Production Editor: Daniel Scribner
Copy Editor: Sharon Wilkey
Production Manager: Tim Tate
Vice President and Executive Group Publisher: Richard Swadley
Vice President and Executive Publisher: Joseph B. Wikert
Vice President and Publisher: Neil Edde
Book Designer: Maureen Forys and Judy Fung
Proofreader: Nancy Riddiough
Indexer: Ron Strauss
Cover Designer: Ryan Sneed
Cover Image: Pete Gardner / Digital Vision / Getty Images

ISBN: 978-0-4701-8742-5

Library of Congress Cataloging-in-Publication Data

Petroutsos, Evangelos.
  Mastering Microsoft Visual basic 2008 / Evangelos Petroutsos. — 1st ed.
      p. cm.
  ISBN 978-0-470-18742-5 (paper/website)
1. Microsoft Visual BASIC. 2. BASIC (Computer program language) I. Title.
  QA76.73.B3P492285 2008
  005.2′762 — dc22
2007051637

10 9 8 7 6 5 4 3 2 1

To my dearest and most precious ones, Nefeli and Eleni-Myrsini.

Acknowledgments

Many people contributed to this book, and I would like to thank them all. I first want to express my deep appreciation to Mark Ridgeway for contributing the Web-related chapters (Chapters 25, 26, and 27) and Chapter 5, ''The Vista Interface,'' and to John Mueller for providing the technical edit of this book. I want to thank the programmers at Microsoft for their commitment to Visual Basic. Visual Basic has evolved from a small, limited programming environment to a first-class development tool for building all types of Windows and Web applications, from small hobbyist projects to enterprise scale applications.

Special thanks to the talented people at Sybex — to all of them and to each one individually. Starting with editor Toni Zuccarini Ackley, who has taken this book under her wing and improved it in numerous ways. To acquisitions editor Tom Cirtin, who has followed the progress of the book, its ups and downs, and managed to coordinate the entire team. To production editor Daniel Scribner, who kept this project in order and on schedule. Thanks, Daniel. To copyeditor Sharon Wiley, proofreader Nancy Riddiough, indexer Ron Strauss and everyone else who added their expertise and talent

About the Author

Evangelos Petroutsos works as a consultant on medium to large projects, teaches, and writes articles — but he mostly writes code, VB code. He specializes in VB and SQL, and from the new technologies, he fancies XML. He has authored many articles and more than 10 programming books, including the best-selling titles *Mastering Microsoft Visual Basic 2005* and *Mastering Visual Basic .NET Database Programming*, both published by Sybex.

Contents at a Glance

Contents

Introduction

Welcome to Visual Basic 2008, the most mature version yet of the most popular programming language for building Windows and web applications. In modern software development, however, the language is only one of the components we use to build applications. The most important component is the .NET Framework, which is an indispensable component of every application; it's actually more important than the language itself. You can think of the Framework as an enormous collection of functions for just about any programming task. All drawing methods, for example, are part of the System.Drawing class. To draw a rectangle, you call the `DrawRectangle` method of the System.Drawing class, passing the appropriate arguments. To create a new folder, you call the `CreateDirectory` method of the Directory class, and to retrieve the files in a folder you call the `GetFiles` method of the same class.

The Framework contains all the functionality of the operating system and makes it available to your application through methods. The language and the Framework are the two ''programming'' components, absolutely necessary to build Windows applications. It's possible to develop applications with these two components alone, but the process would be awfully slow. The software development process relies on numerous tools that streamline the coding experience, and these tools are provided for us by Visual Studio 2008.

The third component is an integrated environment that hosts a number of tools enabling you to perform many common tasks with point-and-click operations. It's basically an environment in which you can design your forms with visual tools and write code as well. This environment, provided by Visual Studio 2008, is known as an integrated development environment, or IDE. You'll be amazed by the functionality provided by the tools of Visual Studio 2008: you can actually design a functional data-driven application without writing a single line of code. You can use similar tools in the same environment to design a fancy data-driven web page without a single line of code. Visual Studio even provides tools for manipulating databases and allows you to switch between tasks, all in the same, streamlined environment. You realize, of course, that Visual Studio isn't about writing applications without code; it just simplifies certain tasks through wizards and, more often than not, we step in and provide custom code to write a functional application. Even so, Visual Studio 2008 provides numerous tools, from debugging tools to help you track and fix all kinds of bugs in your code, to database manipulation tools.

This book shows you how to use Visual Studio 2008 and Visual Basic 2008 to design rich Windows and web applications. We'll start with the visual tools and then we'll explore Visual Basic and the Framework. A Windows application consists of a visual interface and code behind the elements of the interface. (The code handles the user actions on the visual interface, such as the click of a button, the selection of a menu item, and so on.) You'll use the tools of Visual Studio to build the visual interface and then you'll program the elements of the application with Visual Basic. For any nontrivial processing, such as file and folder manipulation, data storage, and so on, you'll use the appropriate classes of the .NET Framework. A substantial segment of this book deals with the most useful components of the Framework.

The Mastering Series

The *Mastering* series from Sybex provides outstanding instruction for readers with intermediate and advanced skills, in the form of top-notch training and development for those already working in their field and clear, serious education for those aspiring to become pros. Every *Mastering* book includes the following:

- Real-World Scenarios, ranging from case studies to interviews, that show how the tool, technique, or knowledge presented is applied in actual practice
- Skill-based instruction, with chapters organized around real tasks rather than abstract concepts or subjects.
- Self-review test questions, so you can be certain you're equipped to do the job right.

Who Should Read This Book?

You don't need a solid knowledge of Visual Basic to read this book, but you do need a basic understanding of programming. You need to know the meaning of variables and functions and how an `If...Then` structure works. This book is aimed at the typical programmer who wants to get the most out of Visual Basic. It covers the topics I felt are of use to most VB programmers, and it does so in depth. Visual Basic 2008 and the .NET Framework 3.5 are two extremely rich programming tools, and I had to choose between a superficial coverage of many topics and an in-depth coverage of fewer topics. To make room for more topics, I have avoided including a lot of reference material and lengthy listings. For example, you won't find complete project listings or form descriptions. I assume that you can draw a few controls on a form and set their properties, and that you don't need long descriptions of the controls' properties. I'm also assuming that you don't want to read the trivial segments of each application. Instead, the listings concentrate on the ''meaty'' part of the code: the procedures that explain the topic at hand.

The topics covered in this book were chosen to provide a solid understanding of the principles and techniques for developing applications with Visual Basic. Programming isn't about new keywords and functions. I chose the topics I felt every programmer should learn in order to master the language. I was also motivated by my desire to present useful, practical examples. You will not find all topics equally interesting or important. My hope is that everyone will find something interesting and something of value for his or her daily work — whether it's an application that maps the folders and files of a drive to a TreeView control, an application that prints tabular data, a data-driven application for editing customers or products, or an application that saves a collection of objects to a file.

Many books offer their readers long, numbered sequences of steps to accomplish a task. Following instructions simplifies certain tasks, but programming isn't about following instructions. It's about being creative; it's about understanding principles and being able to apply the same techniques in several practical situations. And the way to creatively exploit the power of a language such as Visual Basic 2008 is to understand its principles and its programming model.

In many cases, I provide a detailed, step-by-step procedure that will help you accomplish a task, such as designing a menu, for example. But not all tasks are as simple as designing menus. I explain why things must be done in a certain way, and I present alternatives and try to connect new topics to those explained earlier in the book. In several chapters, I expand on applications developed in earlier chapters. Associating new knowledge with something you have mastered already provides positive feedback and a deeper understanding of the language.

This book isn't about the hottest features of the language; it's about solid programming techniques and practical examples. After you master the basics of programming Windows applications with Visual Basic 2008 and you feel comfortable with the more advanced examples of the book, you will find it easy to catch up with the topics not discussed in this book. Of course, you will find information about the latest data access techniques, as well as an introduction to LINQ (Language Integrated Query), which is the hottest new component of the Framework.

How about the Advanced Topics?

Some of the topics discussed in this book are nontrivial, and quite a few topics can be considered advanced. The TreeView control, for example, is not a trivial control, like the button or text box control, but it's ideal for displaying hierarchical information. (This is the control that displays the hierarchy of folders in Windows Explorer.) If you want to build an elaborate user interface, you should be able to program controls such as the TreeView and ListView controls, which are discussed in Chapter 9, ''The TreeView and ListView Controls.''

You may also find some examples to be more difficult than you expected. I have tried to make the text and the examples easy to read and understand, but not unrealistically simple. In Chapter 15, ''Accessing Folders and Files,'' you will find information about the File and Directory objects. You can use these objects to access and manipulate the file system from within your application, but this chapter wouldn't be nearly as useful without an application that shows you how to scan a folder recursively (scan the folder's files and then its subfolders, to any depth). To make each chapter as useful as I could, I've included nontrivial examples, which will provide a better understanding of the topics. In addition, many of these examples can be easily incorporated into your applications.

You can do a lot with the TreeView control with very little programming, but to make the most out of this control, you must be ready for some advanced programming — nothing terribly complicated, but some things just aren't trivial. Programming most of the operations of the TreeView control, for instance, is not complicated, but if your application calls for populating a TreeView control with an arbitrary number of branches (such as mapping a directory structure to a TreeView control), the code can get complex. The same goes for printing; it's fairly straightforward to write a program that prints some text, but printing tabular reports takes substantial coding effort.

The reason I've included the more advanced examples is that the corresponding chapters would be incomplete without them. If you find some material to be over your head at first reading, you can skip it and come back to it after you have mastered other aspects of the language. But don't let a few advanced examples intimidate you. Most of the techniques are well within the reach of an average VB programmer. The few advanced topics were included for the readers who are willing to take that extra step and build elaborate interfaces by using the latest tools and techniques.

There's another good reason for including advanced topics. Explaining a simple topic, such as how to populate a collection with items, is very simple. But what good is it to populate a collection if you don't know how to save it to disk and read back its items in a later session? Likewise, what good is it to learn how to print simple text files? In a business environment, you will most likely be asked to print a tabular report, which is substantially more complicated than printing text. In Chapter 20, ''Printing with Visual Basic 2008,'' you will learn how to print business reports with headers, footers, and page numbers, and even how to draw grids around the rows and columns of the report. One of my goals in writing this book was to exhaust the topics I've chosen to discuss and present all the information you need to do something practical.

The Structure of the Book

This book isn't meant to be read from cover to cover, and I know that most people don't read computer books this way. Each chapter is independent of the others, although all chapters contain references to other chapters. Each topic is covered in depth; however, I make no assumptions about the reader's knowledge of the topic. As a result, you may find the introductory sections of a chapter too simple. The topics become progressively more advanced, and even experienced programmers will find some new information in most chapters. Even if you are familiar with the topics in a chapter, take a look at the examples. I have tried to simplify many of the advanced topics and demonstrate them with clear, practical examples.

This book tries to teach through examples. Isolated topics are demonstrated with short examples, and at the end of many chapters you'll build a large, practical application (a real-world application) that ''puts together'' the topics and techniques discussed throughout the chapter. You may find some of the more advanced applications a bit more difficult to understand, but you shouldn't give up. Simpler applications would have made my job easier, but the book wouldn't deserve the *Mastering* title, and your knowledge of Visual Basic wouldn't be as complete.

The book starts with the fundamentals of Visual Basic 2008. You'll learn how to design visual interfaces with point-and-click operations and how to program a few simple events, such as the click of the mouse on a button. After reading the first two chapters, you'll understand the structure of a Windows application. Then you'll explore the elements of the visual interface (the basic Windows controls) and how to program them. You'll also learn about the My object and code snippets, two features that make Visual Basic so simple and fun to use. These two objects will also ease the learning process and make it much simpler to learn the features of the language.

I then discuss in detail the basic components of Windows applications. I explain the most common controls you'll use in building Windows forms in detail, as well as how to work with forms: how to design forms, how to design menus for your forms, how to create applications with multiple forms, and so on. You will find detailed discussions of many Windows controls, as well as how to take advantage of the built-in dialog boxes, such as the Font and Color dialog boxes, in your applications.

Visual Basic 2008 is a truly object-oriented language, and objects are the recurring theme in every chapter. The three following chapters (chapter 10, 11 and 12) contain a formal and more systematic treatment of objects. You will learn how to build custom classes and controls, which will help you understand object-oriented programming a little better. You will also learn about inheritance and will see how easy it is to add custom functionality to existing classes through inheritance.

The following few chapters deal with some of the most common classes of the .NET Framework. The Framework is at the very heart of Windows programming; it's your gateway to the functionality of the operating system itself, and it's going to be incorporated into the next version of Windows. You'll examine several extremely interesting topics such as collections (for example, ArrayLists and HashTables), the classes for manipulating files and folders, the String-Builder class that manipulates text, XML serialization, and a few more, including the Language Integrated Query component (LINQ, which is brand new to the latest version of the Framework).

Then you will find a few chapters on graphics. You'll learn how to use the classes of the Framework that generate graphics, and you'll learn how to create vector drawings as well as how to manipulate bitmaps. In Chapter 20, you'll learn everything you need to create printouts with Visual Basic 2008 and see a few practical examples.

The first twenty chapters deal with the fundamentals of the language and Windows applications. Following these chapters, you will find an overview of the data-access tools. The emphasis is on the visual tools, and you will learn how to query databases and present data to the user. You will also find information on programming the basic objects of ADO.NET and write simple data-driven Windows applications.

In the last few chapters of this book you will learn about web applications, the basics of ASP.NET 2, how to develop data-bound web applications, and how to write web services.

Downloading This Book's Code

The code for the examples and projects can be downloaded from the Sybex website (`www.sybex.com`). At the main page, you can find the book's page by searching for the author, the title, or the ISBN (9780470187425), and then clicking the book's link listed in the search results. On the book's page, click the Download link. It will take you to the download page. The downloaded source code is a zip file, which you can unzip with the WinZip utility.

How to Reach the Author

Despite our best efforts, a book of this size is bound to contain errors. Although a printed medium isn't as easy to update as a website, I will spare no effort to fix every problem you report (or I discover). The revised applications, along with any other material I think will be of use to the readers of this book, will be posted on the Sybex website. If you have any problems with the text or the applications in this book, you can contact me directly at `pevangelos@yahoo.com`.

Although I can't promise a response to every question, I will fix any problems in the examples and provide updated versions. I would also like to hear any comments you may have on the book, about the topics you liked or did not like, and how useful the examples are. Your comments will be taken into consideration in future editions.

Chapter 1

Getting Started with Visual Basic 2008

I'm assuming that you have installed one of the several versions of Visual Studio 2008. For this book, I used the Professional Edition of Visual Studio, but just about everything discussed in this book applies to the Standard Edition as well. Some of the features of the Professional Edition that are not supported by the Standard Edition concern database tools, which are discussed in Chapters 21 through 24 of this book.

You may have even already explored the new environment on your own, but this book starts with an overview of Visual Studio and its basic tools. It doesn't even require any knowledge of VB 6, just some familiarity with programming at large. As you already know, Visual Basic 2008 is just one of the languages you can use to build applications with Visual Studio 2008. I happen to be convinced that it is also the simplest, most convenient language, but this isn't really the issue; I'm assuming you have your reasons to code in VB, or else you wouldn't be reading this book. What you should keep in mind is that Visual Studio 2008 is an integrated environment for building, testing, debugging, and deploying a variety of applications: Windows applications, web applications, classes and custom controls, and even console applications. It provides numerous tools for automating the development process, visual tools for performing many common design and programming tasks, and more features than any author would hope to cover.

In this chapter, you'll learn how to do the following:

- Navigate the integrated development environment of Visual Studio
- Understand the basics of a Windows application

Exploring the Integrated Development Environment

Visual Basic 2008 is just one of the languages you can use to program your applications. The language is only one aspect of a Windows application. The visual interface of the application isn't tied to a specific language, and the same tools you'll use to develop your application's interface will also be used by all programmers, regardless of the language they'll use to code the application.

To simplify the process of application development, Visual Studio provides an environment that's common to all languages, which is known as an *integrated development environment (IDE)*. The purpose of the IDE is to enable the developer to do as much as possible with visual tools, before writing code.

The IDE provides tools for designing, executing, and debugging your applications. It will be a while before you explore all the elements of the IDE, and I will explain the various items as needed in the course of the book. In this section, you'll look at the basic components of the IDE needed to build simple Windows applications. You'll learn how its tools allow you to quickly design the user interface of your application, as well as how to program the application.

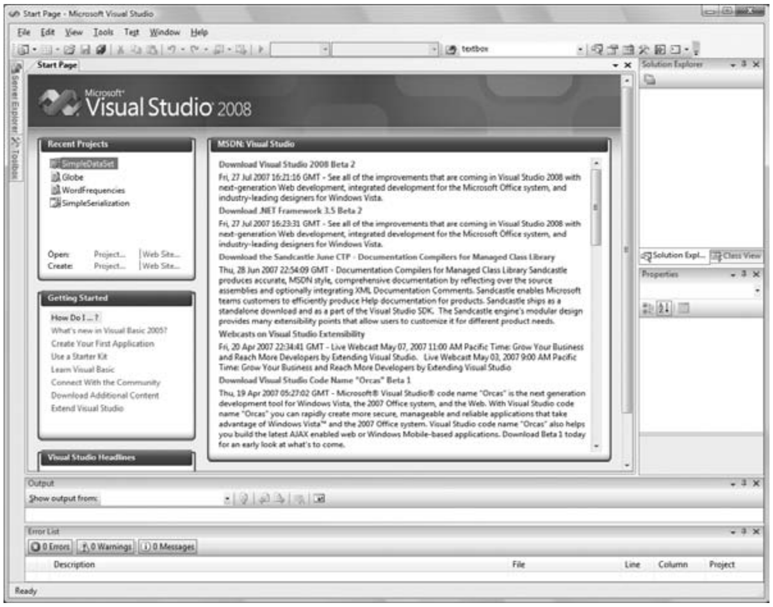
The IDE is your second desktop, and you'll be spending most of your productive hours in this environment.

The Start Page

When you run Visual Studio 2008 for the first time, you will be prompted to select the type of projects you plan to build with Visual Studio, so that the environment can be optimized for that specific type of development. I'm assuming that you have initially selected the Visual Basic Development settings, which will optimize your copy of Visual Studio for building Windows and web applications with Visual Basic 2008. You can always change these settings, as explained at the end of this section.

After the initial configuration, you will see a window similar to the one shown in Figure 1.1. The Recent Projects pane will be empty, of course, unless you have already created some test projects. Visual Studio 2008 will detect the settings of a previous installation, so if you're upgrading from an earlier version of Visual Studio, the initial screen will not be identical to the one shown in Figure 1.1.

FIGURE 1.1
This is what you'll see when you start Visual Studio for the first time.



Download Visual Studio 2008 Beta 2

Fri, 27 Jul 2007 16:21:16 GMT - See all of the improvements that are coming in Visual Studio 2008 with next-generation Web development, integrated development for the Microsoft Office system, and industry-leading designers for Windows Vista.

Download .NET Framework 3.5 Beta 2

Fri, 27 Jul 2007 16:23:31 GMT - See all of the improvements that are coming in Visual Studio 2008 with next-generation Web development, integrated development for the Microsoft Office system, and industry-leading designers for Windows Vista.

Download the Sandcastle June CTP - Documentation Compilers for Managed Class Library

Thu, 28 Jun 2007 22:54:09 GMT - Documentation Compilers for Managed Class Library Sandcastle produces accurate, MSDN style, comprehensive documentation by reflecting over the source assemblies and optionally integrating XML Documentation Comments. Sandcastle enables Microsoft teams customers to efficiently produce Help documentation for products. Sandcastle ships as a standalone download and as a part of the Visual Studio SDK. The Sandcastle engine's modular design provides many extensibility points that allow users to customize it for different product needs.

Webcasts on Visual Studio Extensibility

Fri, 20 Apr 2007 22:34:41 GMT - Live Webcast May 07, 2007 11:00 AM Pacific Time: Grow Your Business and Reach More Developers by Extending Visual Studio. Live Webcast May 03, 2007 9:00 AM Pacific Time: Grow Your Business and Reach More Developers by Extending Visual Studio

Download Visual Studio Code Name "Orcas" Beta 1

Thu, 19 Apr 2007 05:27:02 GMT - Microsoft® Visual Studio® code name "Orcas" is the next generation development tool for Windows Vista, the 2007 Office system, and the Web. With Visual Studio code name "Orcas" you can rapidly create more secure, manageable and reliable applications that take advantage of Windows Vista™ and the 2007 Office system. Visual Studio code name "Orcas" also helps you build the latest AJAX enabled web or Windows Mobile-based applications. Download Beta 1 today for an early look at what's to come.

On the Start Page of Visual Studio, you will see the following panes:

Recent Projects Here you see a list of the projects you opened most recently with Visual Studio, and you can select the one you want to open again — chances are that you will continue working on the same project as the last time. Each project's name is a hyperlink, and you can open it by clicking its name. At the bottom of the Recent Projects section are two hyperlinks, for opening or creating another project.

MSDN: Visual Studio This section is a browser window that displays an MSDN (the Microsoft Developer Network, which is the definitive resource for all Microsoft technologies and products) page when the computer is connected to the Internet. In this section, you will see news about Visual Studio, the supported languages, articles, and other interesting bits of information.

Getting Started This section contains links to basic programming tasks in the product's documentation.

Visual Studio Headlines This section contains links to announcements and other news of interest to VB developers.

Most developers will skip the Start Page. To do so, open the Tools menu and choose the Import And Export Settings command to start a configuration wizard. In the first dialog box of the wizard, select the Reset All Settings check box and click the Next button. The next screen of the wizard prompts you for the location where the new settings will be saved, so that Visual Studio can read them every time it starts. Leave the default location as is and click Next again to see the last screen of the wizard, in which you're prompted to select a default collection of settings. This collection depends on the options you've installed on your system. I installed Visual Studio 2008 with Visual Basic only on my system, and I was offered the following options: General Development Settings, Visual Basic Development Settings, and Web Development Settings. For the default configuration of my copy of Visual Studio, and for the purposes of this book, I chose the Visual Basic Development Settings, so that Visual Studio could optimize the environment for a typical VB developer. Click the Finish button to see a summary of the process and then close the wizard.

Starting a New Project

At this point, you can create a new project and start working with Visual Studio. To best explain the various items of the IDE, we will build a simple form. The form is the window of your application — it's what users will see on their Desktop when they run your application.

Open the File menu and choose New Project, or click Create Project/Solution in the Start Page. In the New Project dialog box that pops up (see Figure 1.2), you'll see a list of project types you can create with Visual Studio. The most important ones are Windows Forms Applications, which are typical Windows applications with one or more forms (windows); Console Applications, which are simple applications that interact with the user through a text window (the console); Windows Forms Control Libraries, which are collections of custom controls; and Class Libraries, which are collections of classes. These are the project types we'll cover in depth in this book.

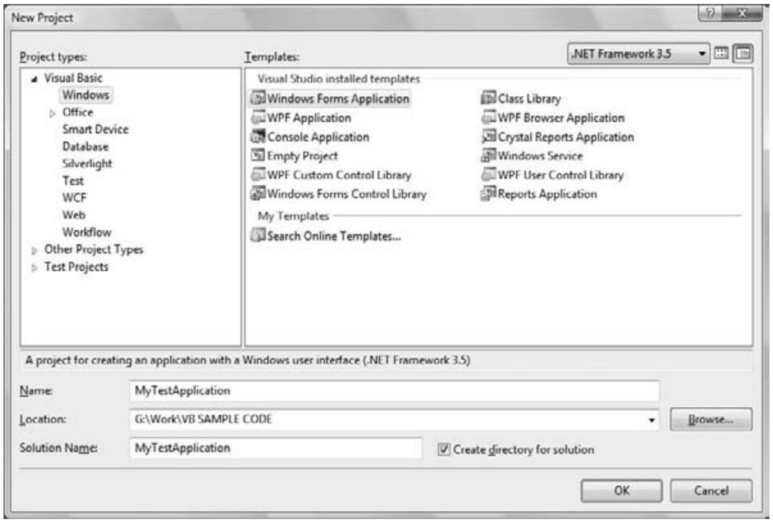
If you have installed Visual Basic 2008 Express Edition, you will see fewer project types in the New Project dialog box, but the projects discussed in this book are included.

Notice the Create Directory For Solution check box in the dialog box of Figure 1.2. By default, Visual Studio creates a new folder for the project under the folder you have specified in the Location box. If you want to put together a short application to test a feature of the language, or perform some trivial task, you may not wish to save the project. In this case, just clear the check box to skip the creation of a new project folder.

You can always save a project at any time by choosing the Save All command from the File menu. You'll be prompted at that point about the project's folder, and Visual Studio will save the project under the folder you specified. If you decide to discard the project, you can create a new project or close Visual Studio. Visual Studio will prompt you about an open project that hasn't been saved yet, and you can choose not to save it.

You may discover at some point that you have created too many projects, which you don't really need. You can remove these projects from your system by deleting the corresponding folders — no special action is required. You'll know it's time to remove the unneeded project folder when Visual Studio suggests project names such as WindowsApplication9 or WindowsApplication49.

FIGURE 1.2
The New Project dialog box



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

Name: MyTestApplication

Location: G:\Work\VB SAMPLE CODE

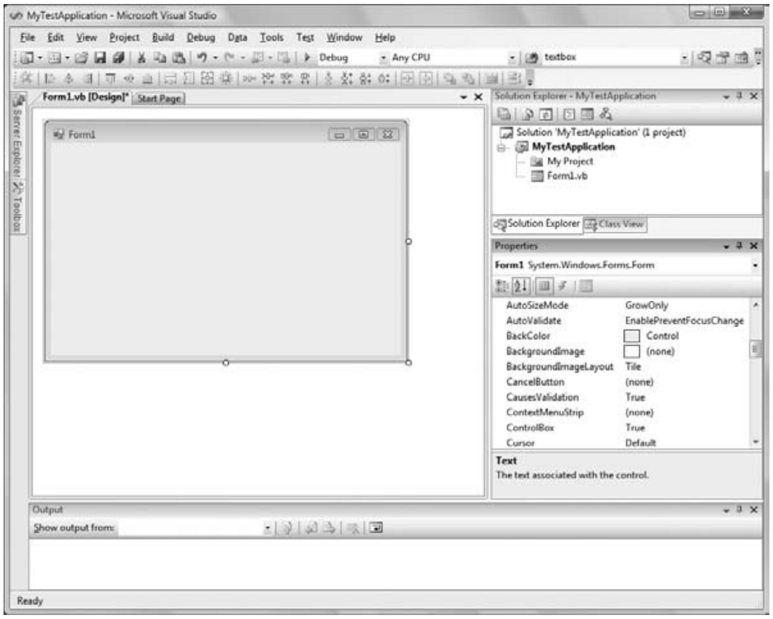
Solution Name: MyTestApplication

☑ Create directory for solution

For our project, select the Windows Forms Application template; Visual Studio suggests the name WindowsApplication1 as the project name. Change it to **MyTestApplication**, select the Create Directory For Solution check box, and then click the OK button to create the new project.

What you see now is the Visual Studio IDE displaying the Form Designer for a new project, as shown in Figure 1.3. The main window of your copy of Visual Studio may be slightly different, but don't worry about it. I'll go through all the components you need to access in the process of designing, coding, and testing a Windows application.

FIGURE 1.3
The integrated development environment of Visual Studio 2008 for a new project



AutoSizeMode	GrowOnly
AutoValidate	EnablePreventFocusChange
BackColor	Control
BackgroundImage	(none)
BackgroundImageLayout	Tile
CancelButton	(none)
CausesValidation	True
ContextMenuStrip	(none)
ControlBox	True
Cursor	Default

Text
The text associated with the control.

The new project contains a form already: the `Form1` component in the Solution Explorer. The main window of the IDE is the Form Designer, and the gray surface on it is the window of your new application in design mode. Using the Form Designer, you'll be able to design the visible interface of the application (place various components of the Windows interface on the form and set their properties) and then program the application.

The default environment is rather crowded, so let's hide a few of the toolbars that we won't use in the projects of the first few chapters. You can always show any of the toolbars at any time. Open the View menu and choose Toolbars. You'll see a submenu with 28 commands that are toggles. Each command corresponds to a toolbar, and you can turn the corresponding toolbar on or off by clicking one of the commands in the Toolbars submenu. For now, turn off all the toolbars except for the Layout and Standard toolbars. These are the toolbars shown by default and you shouldn't hide them; if you do, this is the place to make them visible again.

The last item in the Toolbars submenu is the Customize command, which leads to a dialog box in which you can specify which of the toolbars and which of the commands you want to see. After you have established a work pattern, use this menu to customize the environment for the way you want to work with Visual Studio. You can hide just about any component of the IDE, except for the main menu — after all, you have to be able to undo the changes!

Using the Windows Form Designer

To design the form, you must place on it all the controls you want to display to the user at runtime. The controls are the components of the Windows interface (buttons, text boxes, radio buttons, lists, and so on). Open the Toolbox by moving the pointer over the Toolbox tab at the far left; the Toolbox, shown in Figure 1.4, pulls out. This Toolbox contains an icon for each control you can use on your form.

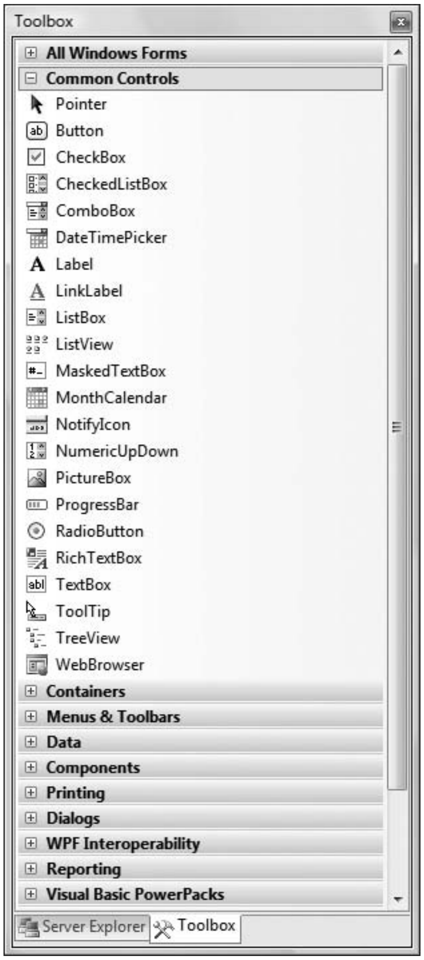
The controls are organized into groups according to each control's function on the interface. In the first part of the book, we'll create simple Windows applications and we'll use the controls on the Common Controls tab. When you develop web applications, you will see a different set of icons in the Toolbox.

To place a control on the form, you can double-click the icon of the control. A new instance with a default size will be placed on the form. Then you can position and resize it with the mouse. Or you can select the control from the Toolbox with the mouse and then click and drag the mouse over the form and draw the outline of the control. A new instance of the control will be placed on the form, and it will fill the rectangle you specified with the mouse. Start by placing a TextBox control on the form.

The control's properties will be displayed in the Properties window (see Figure 1.5). This window, at the far right edge of the IDE and below the Solution Explorer, displays the properties of the selected control on the form. If the Properties window is not visible, open the View menu and choose Properties Window, or press F4. If no control is selected, the properties of the selected item in the Solution Explorer are displayed.

In the Properties window, also known as the Properties Browser, you see the properties that determine the appearance of the control and (in some cases) its function. The properties are organized in categories according to their role. The properties that determine the appearance of the control are listed alphabetically under the header Appearance, the properties that determine the control's behavior are listed alphabetically under the header Behavior, and so on. You can click the AZ button on the window's title bar to display all properties in alphabetical order. After you familiarize yourself with the basic properties, you will most likely switch to the alphabetical list.

FIGURE 1.4
Windows Forms Toolbox of the Visual Studio IDE



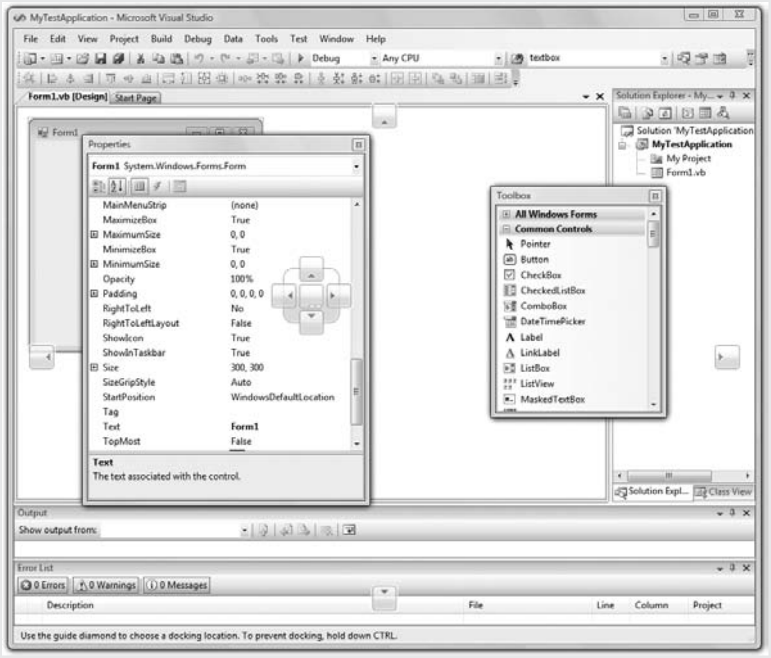
REARRANGING THE IDE WINDOWS

As soon as you place a control on the form, the Toolbox retracts to the left edge of the Designer. You can fix this window on the screen by clicking the icon with the pin on the Toolbox's toolbar. (It's the icon next to the Close icon at the upper-right corner of the Toolbox window, and it appears only when the Toolbox window is docked, but not while it's floating.)

You can easily rearrange the various windows that make up the IDE by moving them around with the mouse. Move the pointer to a window's title bar, press the left mouse button, and drag the window around. A window may not follow the mouse, because its position is locked. In this case, click the pin icon in the upper-right corner of the window to unlock the window's position and then move it around with the mouse.

As you move the window, eight semitransparent buttons with arrows appear on the screen, indicating the area where the window can be docked. Keep moving the window until the pointer hovers over one of these buttons, and the docking area appears in semitransparent blue color. Find the desired docking location for the window and release the mouse. If you release the mouse while the pointer is not on top of an arrow, the window is not docked. Instead, it remains at the current location as a floating window, and you can move it around at will with your mouse.

Most developers would rather work with docked windows, and the default positions of the IDE windows are quite convenient. If you want to open even more windows and arrange them differently on the screen, use the docking feature of the IDE to dock the additional windows.



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	300, 300
SizeGripStyle	Auto
StartPosition	WindowsDefaultLocation
Tag	
Text	**Form1**
TopMost	False

Text
The text associated with the control.

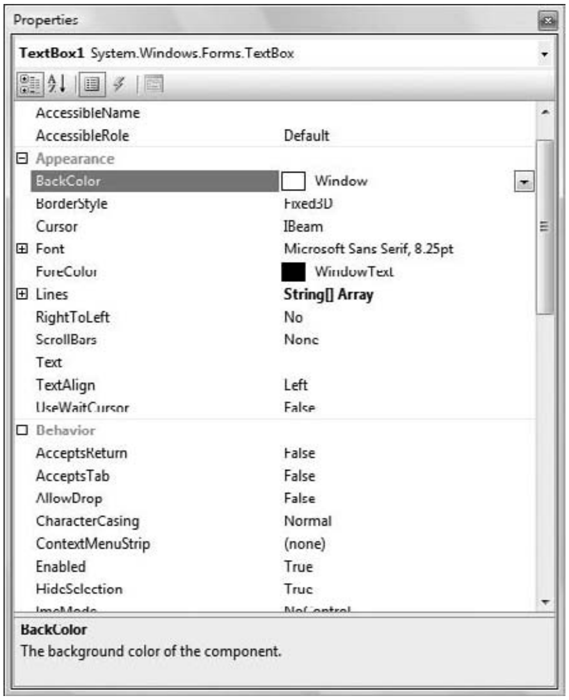
Use the guide diamond to choose a docking location. To prevent docking, hold down CTRL.

Locate the TextBox control's `Text` property and set it to **My TextBox Control** by entering the string into the box next to the property name. The control's `Text` property is the string that appears in the control (the control's caption), and most controls have a `Text` property.

Next locate its `BackColor` property and select it with the mouse. A button with an arrow appears next to the current setting of the property. Click this button, and you'll see a dialog box with three tabs (Custom, Web, and System), as shown in Figure 1.6. In this dialog box, you can select the color that will fill the control's background. Set the control's background color to yellow and notice that the control's appearance changes on the form.

One of the settings you'll want to change is the font of the various controls. While the TextBox control is still selected on the form, locate the control's `Font` property in the Properties window. You can click the plus sign in front of the property name and set the individual properties of the font, or you can click the ellipsis button to invoke the Font dialog box. Here you can set the control's font and its attributes and then click OK to close the dialog box. Set the TextBox control's `Font` property to Verdana, 14 points, bold. As soon as you close the Font dialog box, the control on the form is adjusted to the new setting.

FIGURE 1.5
Properties of a TextBox control



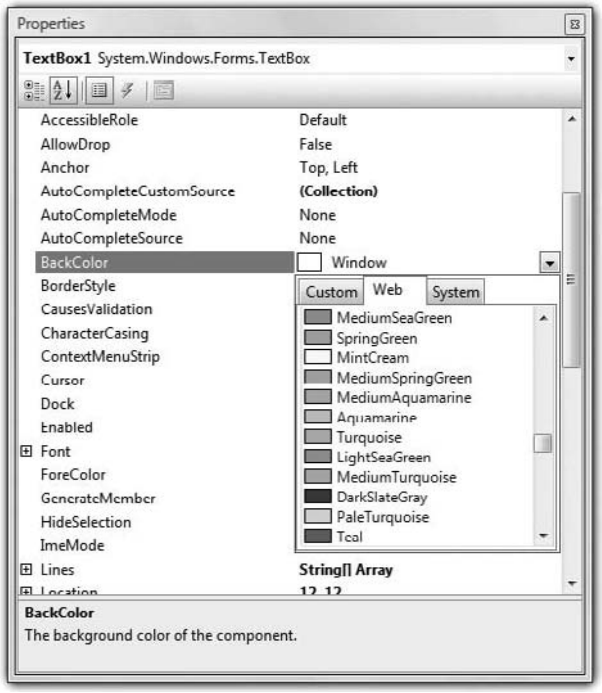
Properties

TextBox1 System.Windows.Forms.TextBox

```
AccessibleName
AccessibleRole           Default
⊟ Appearance
BackColor                Window
BorderStyle              Fixed3D
Cursor                   IBeam
⊞ Font                   Microsoft Sans Serif, 8.25pt
ForeColor                WindowText
⊞ Lines                  String[] Array
RightToLeft              No
ScrollBars               None
Text
TextAlign                Left
UseWaitCursor            False
⊟ Behavior
AcceptsReturn            False
AcceptsTab               False
AllowDrop                False
CharacterCasing          Normal
ContextMenuStrip         (none)
Enabled                  True
HideSelection            True
```

BackColor
The background color of the component.

FIGURE 1.6
Setting a color property in the Properties window



Properties

TextBox1 System.Windows.Forms.TextBox

```
AccessibleRole              Default
AllowDrop                   False
Anchor                      Top, Left
AutoCompleteCustomSource    (Collection)
AutoCompleteMode            None
AutoCompleteSource          None
BackColor                   Window
BorderStyle                 Custom | Web | System
CausesValidation              MediumSeaGreen
CharacterCasing               SpringGreen
ContextMenuStrip              MintCream
Cursor                        MediumSpringGreen
Dock                          MediumAquamarine
Enabled                       Aquamarine
⊞ Font                        Turquoise
ForeColor                     LightSeaGreen
GenerateMember                MediumTurquoise
HideSelection                 DarkSlateGray
ImeMode                       PaleTurquoise
                              Teal
⊞ Lines                     String[] Array
```

BackColor
The background color of the component.

There's a good chance that the string you assigned to the control's `Text` property won't fit in the control's width when rendered in the new font. Select the control on the form with the mouse, and you will see eight handles along its perimeter. Rest the pointer over any of these handles, and it will assume a shape indicating the direction in which you can resize the control. Make the control long enough to fit the entire string. If you have to, resize the form as well. Click somewhere on the form, and when the handles along its perimeter appear, resize it with the mouse.

Some controls, such as the Label, Button, and CheckBox controls, support the `AutoSize` property, which determines whether the control is resized automatically to accommodate its caption. The TextBox control, as well as many others, doesn't support the `AutoSize` property. If you attempt to make the control tall enough to accommodate a few lines of text, you'll realize that you can't change the control's height. By default, the TextBox control accepts a single line of text, and you must set its `MultiLine` property to True to resize the TextBox control vertically.

Real World Scenario

The Font Is a Design Element

Like documents, forms should be designed carefully and follow the rules of a printed page design. At the very least, you shouldn't use multiple fonts on your forms, just as you shouldn't mix different fonts on a printed page. You could use two font families on rare occasions, but you shouldn't overload your form. You also shouldn't use the bold style in excess.

To avoid adjusting the `Font` property of multiple controls on the form, you should set the form's font first, because each control you place on a form inherits the form's font. If you change the form's font, the controls will be adjusted accordingly, but this may throw off the alignment of the controls on the form. You should experiment with a few Label controls, select a font that you like that's appropriate for your interface (you shouldn't use a handwritten style with a business application, for example) and then set the form's `Font` property to the desired font. Every time you add a new form to the application, you should start by setting its `Font` property to the same font, so that the entire application will have a consistent look.

The font is the most basic design element, whether you're designing forms or a document. Various components of the form may have a different font size, even a different style (like bold or italics), but there must be a dominant font family that determines the look of the form. The Verdana family was designed for viewing documents on computer monitors and is a popular choice. Another great choice is Segoe UI, a new font family introduced with Windows Vista. The Segoe Print font has a distinguished handwritten style, and you can use it with graphics applications.

The second most important design element is color, but you shouldn't get too creative with colors unless you're a designer. I recommend that you stay with the default colors and use similar shades to differentiate a few elements of the interface.

The design of a modern interface has become a new discipline in application development, and there are tools for designing interfaces. One of them is Microsoft's Expression Studio, which enables designers to design the interface and developers to write code, without breaking each other's work. You can download a trial version of Expression Studio from `www.microsoft.com/expression`.

So far, you've manipulated properties that determine the appearance of the control. Now you'll change a property that determines not only the appearance, but also the function of the control. Locate the `Multiline` property. Its current setting is False. Expand the list of available settings and change it to True. (You can also change it by double-clicking the name of the property. This action toggles the True/False settings.) Switch to the form, select the TextBox control, and make it as tall as you wish.

The `Multiline` property determines whether the TextBox control can accept one (if `Multiline` = False) or more (if `Multiline` = True) lines of text. Set this property to True, go back to the `Text` property, set it to a long string, and press Enter. The control breaks the long text into multiple lines. If you resize the control, the lines will change, but the entire string will fit in the control because the control's `WordWrap` property is True. Set it to False to see how the string will be rendered on the control.

Multiline TextBox controls usually have a vertical scroll bar so users can quickly locate the section of text that they're interested in. Locate the control's `ScrollBars` property and expand the list of possible settings by clicking the button with the arrow. This property's settings are *`None`*, *`Vertical`*, *`Horizontal`*, and *`Both`*. Set it to *`Vertical`*, assign a very long string to its `Text` property, and watch how the control handles the text. At design time, you can't scroll the text on the control; if you attempt to move the scroll bar, the entire control will be scrolled. The scroll bar will work as expected at runtime (it will scroll the text vertically).

You can also make the control fill the entire form. Start by deleting all other controls you may have placed on the form and then select the multiline TextBox. Locate the `Dock` property in the Properties window and keep double-clicking the name of the property until its setting changes to *`Fill`*. (You'll learn a lot more about docking controls in Chapter 7, ''Working with Forms.'') The TextBox control fills the form and is resized as you resize the form, both at design time and runtime.

To examine the control's behavior at runtime, press F5. The application will be compiled, and a few moments later, a window filled with a TextBox control will appear on the Desktop (like the one shown in Figure 1.7). This is what the users of your application would see (if this were an application worth distributing, of course).

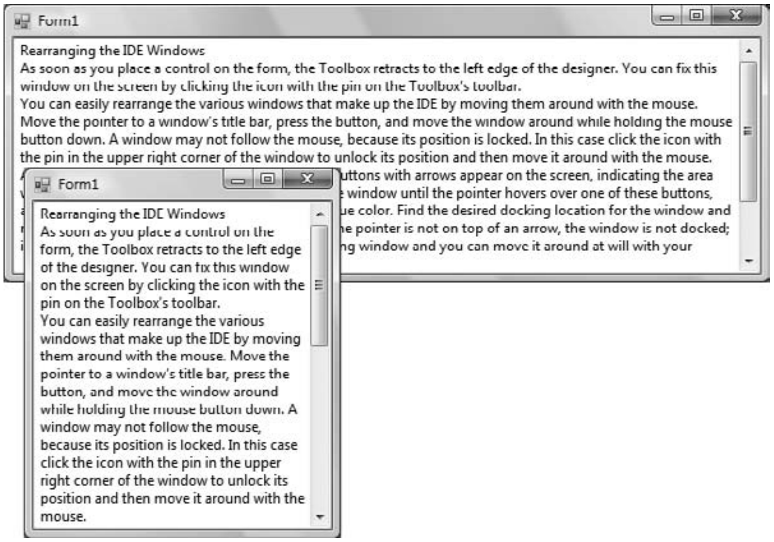


FIGURE 1.7
A TextBox control displaying multiple text lines

Rearranging the IDE Windows
As soon as you place a control on the form, the Toolbox retracts to the left edge of the designer. You can fix this window on the screen by clicking the icon with the pin on the Toolbox's toolbar.
You can easily rearrange the various windows that make up the IDE by moving them around with the mouse. Move the pointer to a window's title bar, press the button, and move the window around while holding the mouse button down. A window may not follow the mouse, because its position is locked. In this case click the icon with the pin in the upper right corner of the window to unlock its position and then move it around with the mouse.
uttons with arrows appear on the screen, indicating the area
window until the pointer hovers over one of these buttons,
ue color. Find the desired docking location for the window and
he pointer is not on top of an arrow, the window is not docked;
ng window and you can move it around at will with your

Rearranging the IDE Windows
As soon as you place a control on the form, the Toolbox retracts to the left edge of the designer. You can fix this window on the screen by clicking the icon with the pin on the Toolbox's toolbar.
You can easily rearrange the various windows that make up the IDE by moving them around with the mouse. Move the pointer to a window's title bar, press the button, and move the window around while holding the mouse button down. A window may not follow the mouse, because its position is locked. In this case click the icon with the pin in the upper right corner of the window to unlock its position and then move it around with the mouse.

Enter some text on the control, select part of the text, and copy it to the Clipboard by pressing Ctrl+C. You can also copy text from any other Windows application and paste it on the TextBox control. Right-click the text on the control and you will see the same context menu you get with

Notepad; you can even change the reading order of the text — not that you'd want to do that with a Western language. When you're finished, open the Debug menu and choose Stop Debugging. This will terminate your application's execution, and you'll be returned to the IDE. The Stop Debugging command is also available as a button with a blue square icon on the toolbar. Finally, you can stop the running application by clicking the Close button in the application's window.

The design of a new application starts with the design of the application's form, which is the application's user interface, or UI. The design of the form determines to a large extent the functionality of the application. In effect, the controls on the form determine how the application will interact with the user. The form itself is a prototype, and you can demonstrate it to a customer before even adding a single line of code. By placing controls on the form and setting their properties, you're implementing a lot of functionality before coding the application. The TextBox control with the settings discussed in this section is a functional text editor.

Creating Your First VB Application

In this section, we'll develop a simple application to demonstrate not only the design of the interface, but also the code behind the interface. We'll build an application that allows the user to enter the name of his favorite programming language, and the application will evaluate the choice. Objectively, VB is a step ahead of all other languages, and it will receive the best evaluation. All other languages get the same grade — good — but not VB.

The project is called WindowsApplication1. You can download the project from the book's website and examine it, but I suggest you follow the steps outlined in this section to build the project from scratch. Start a new project and use the default name, **WindowsApplication1**, and place a TextBox and a Button control on the form. Use the mouse to position and resize the controls on the form, as shown in Figure 1.8.

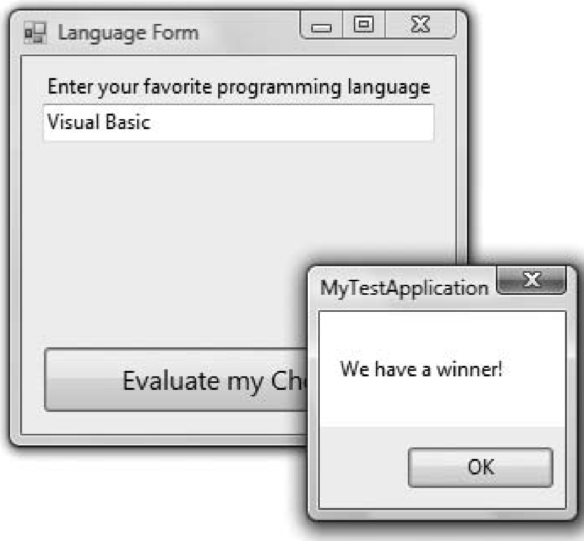
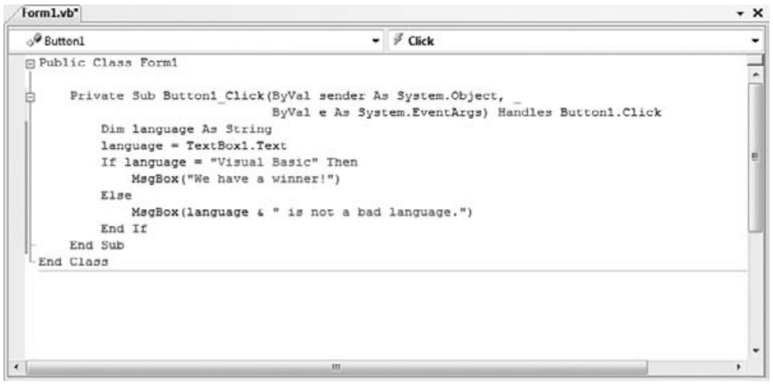


FIGURE 1.8
A simple application that processes a user-supplied string

Start by setting the form's `Font` property to Segoe UI, 9 pt. Arrange and size the controls as shown in Figure 1.8. Then place a Label control on the form and set its `Text` property to **Enter your favorite programming language**. The Label will be resized according to its caption, because the control's `AutoSize` property is True. As you move the controls around on the form, you'll see some blue lines connecting the edges of the controls when they're aligned. These lines are called *snap lines*, and they allow you to align controls on the form.

Now you must insert some code to evaluate the user's favorite language. Windows applications are made up of small code segments, called *event handlers*, which react to specific actions such as the click of a button, the selection of a menu command, the click of a check box, and so on. In the case of our example, we want to program the action of clicking the button. When the user clicks the button, we want to execute some code that will display a message.

To insert some code behind the Button control, double-click the control. You'll see the code window of the application, which is shown in Figure 1.9. You will see only the definition of the procedure, not the code that is shown between the two statements in the figure. The line `Private...` is too long to fit on the printed page, so I inserted a *line continuation character* (an underscore) to break it into two lines. When a line is too long, you can break it into two (or more) lines by inserting this character. Alternatively, you can turn on the WordWrap feature of the editor (you'll see shortly how to adjust the editor's properties). Notice that I also inserted quite a bit of space before the second half of the first code line. It's customary to indent continued lines so they can be easily distinguished from the other lines. If you enter the line continuation character in the editor, the following line will be indented automatically.



```
Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles Button1.Click
        Dim language As String
        language = TextBox1.Text
        If language = "Visual Basic" Then
            MsgBox("We have a winner!")
        Else
            MsgBox(language & " is not a bad language.")
        End If
    End Sub
End Class
```

FIGURE 1.9
Outline of a subroutine that handles the `Click` event of a Button control

The editor opens a subroutine, which is delimited by the following statements:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles Button1.Click

End Sub
```

At the top of the main pane of the Designer, you will see two tabs named after the form: the `Form1.vb [Design]` tab and the `Form1.vb` tab. The first tab is the Windows Form Designer (in which you build the interface of the application with visual tools), and the second is the code editor (in which you insert the code behind the interface). At the top of the code editor, which is what you see in Figure 1.9, are two ComboBoxes. The one on the left contains the names of the controls on the form. The one on the right contains the names of events each control recognizes. When you select a control (or an object, in general) in the left list, the other list's contents are adjusted accordingly. To program a specific event of a specific control, select the name of the control in the left list (the Objects list) and the name of the event in the right list (the Events list). While Button1 is selected in the Objects list, open the Events list to see the events to which the button can react.

The Click event happens to be the default event of the Button control, so when you double-click a Button on the form, you're taken to the Button1_Click subroutine. This subroutine is an event handler, which is invoked automatically every time an event takes place. The event of interest in our example is the Click event of the Button1 control. Every time the Button1 control on the form is clicked, the Button1_Click subroutine is activated. To react to the Click event of the button, you must insert the appropriate code in this subroutine.

There are more than two dozen events for the Button control, and it is among the simpler controls (after all, what can you do to a button besides clicking it?). Most of the controls recognize a very large number of events.

The definition of the event handler can't be modified; this is the event handler's signature (the arguments it passes to the application). All event handlers in VB 2008 pass two arguments to the application: the *sender* argument, which is an object that represents the control that fired the event, and the *e* argument, which provides additional information about the event.

The name of the subroutine is made up of the name of the control, followed by an underscore and the name of the event. This is just the default name, and you can change it to anything you like (such as EvaluateLanguage, for this example, or StartCalculations). What makes this subroutine an event handler is the keyword Handles at the end of the statement. The Handles keyword tells the compiler which event this subroutine is supposed to handle. Button1.Click is the Click event of the *Button1* control. If there were another button on the form, the *Button2* control, you'd have to write code for a subroutine that would handle the Button2.Click event. Each control recognizes many events, and you can provide a different event handler for each control and event combination. Of course, we never program every possible event for every control.

The controls have a default behavior and handle the basic events on their own. The TextBox control knows how to handle keystrokes. The CheckBox control (a small square with a check mark) changes state by hiding or displaying the check mark every time it's clicked. The ScrollBar control moves its indicator (the button in the middle of the control) every time you click one of the arrows at the two ends. Because of this default behavior of the controls, you need not supply any code for the events of most controls on the form.

If you change the name of the control after you have inserted some code in an event handler, the name of the event handled by the subroutine will be automatically changed. The name of the subroutine, however, won't change. If you change the name of the *Button1* control to **bttnEvaluate**, the subroutine's header will become

```
Private Sub Button1_Click(ByVal sender As System.Object, _
             ByVal e As System.EventArgs) Handles bttnEvaluate.Click

End Sub
```

Rename the Button1_Click subroutine to **EvaluateLanguage**. You must edit the code to change the name of the event handler. I try to name the controls before adding any code to the application, so that their event handlers will be named correctly. Alternatively, you can use your own name for each event handler. The default names of the controls you place on a form are quite generic, and you should change them to something more meaningful. I usually prefix the control names with a few characters that indicate the control's type (such as txt, lbl, bttn, and so on), followed by a meaningful name. Names such as *txtLanguage* and *bttnEvaluate* make your code far more readable. It's a good practice to change the default names of the controls as soon as you add the controls to the form. Names such as *Button1*, *Button2*, *Button3*, and so on, don't promote the readability of your code. With the exception of this first sample project, I'm using more-meaningful names for the controls used in this book's projects.

Let's add some code to the `Click` event handler of the `Button1` control. When this button is clicked, we want to examine the text in the text box. If it's *Visual Basic*, we display a message; if not, we display a different message. Insert the lines of Listing 1.1 between the `Private Sub` and `End Sub` statements. (I'm showing the entire listing here; there's no reason to retype the first and last statements.)

Listing 1.1: Processing a User-Supplied String

```
Private Sub EvaluateLanguage(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles Button1.Click
   Dim language As String
   language = TextBox1.Text
   If language = "Visual Basic" Then
      MsgBox("We have a winner!")
   Else
      MsgBox(language & "is not a bad language.")
   End If
End Sub
```

Here's what this code does. First, it assigns the text of the TextBox control to the variable *language*. A *variable* is a named location in memory where a value is stored. Variables are where we store the intermediate results of our calculations when we write code. All variables are declared with a `Dim` statement and have a name and a type.

You could also declare and assign a value to the *language* variable in a single step:

```
Dim language = TextBox1.Text
```

The compiler will create a String variable, because the statement assigns a string to the variable. We'll come back to the topic of declaring and initializing variables in Chapter 2, ''Variables and Data Types.''

Then the program compares the value of the *language* variable to the literal *Visual Basic*, and depending on the outcome of the comparison, it displays one of two messages. The `MsgBox()` function displays the specified message in a small window with the OK button, as shown in Figure 1.8. Users can view the message and then click the OK button to close the message box.

Even if you're not familiar with the syntax of the language, you should be able to understand what this code does. Visual Basic is the simplest of the languages supported by Visual Studio 2008, and we will discuss the various aspects of the language in detail in the following chapters. In the meantime, you should try to understand the process of developing a Windows application: how to build the visible interface of the application and how to program the events to which you want your application to react.

The code of our first application isn't very robust. If the user doesn't enter the string with the exact spelling shown in the listing, the comparison will fail. We can convert the string to uppercase and then compare it with *VISUAL BASIC* to eliminate differences in case. To convert a string to uppercase, use the `ToUpper` method of the String class. The following expression returns the string stored in the *`language`* variable, converted to uppercase:

```
language.ToUpper
```

We should also take into consideration the fact that the user may enter *VB* or *VB 2008*, and so on. In the following section, we'll further improve our application. You never know what users may throw at your application, so whenever possible you should try to limit their responses to the number of available choices. In our case, we can display the names of certain languages (the ones we're interested in) and force the user to select one of them.

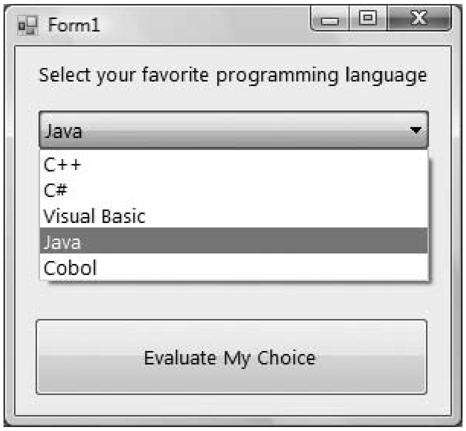
One way to display a limited number of choices is to use a ComboBox control. In the following section, we'll revise our sample application so that users won't have to enter the name of the language. We'll force them to select their favorite language from a list so that we won't have to validate the string supplied by the user.

Making the Application More User-Friendly

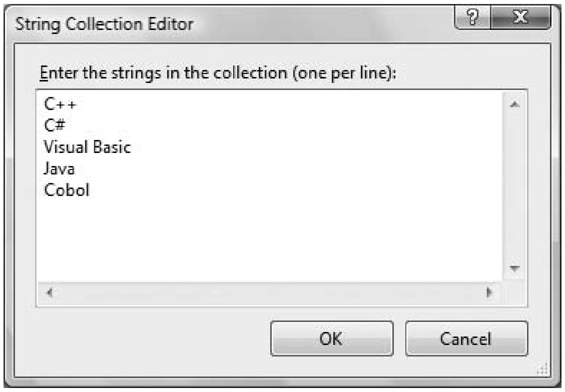
Start a new project: the WindowsApplication2 project. Do not select the Create Directory For Solution check box; we'll save the project from within the IDE. As soon as the project is created, open the File menu and choose Save All to save the project. When the Save Project dialog box appears, click the Browse button to select the folder where the project will be saved. In the Project Location dialog box that appears, select an existing folder or create a new folder such as **MyProjects** or **VB.NET Samples**.

Open the Toolbox and double-click the icon of the ComboBox tool. A ComboBox control will be placed on your form. Now place a Button control on the form and position it so that your form looks like the one shown in Figure 1.10. Then set the button's `Text` property to **Evaluate My Choice**.

FIGURE 1.10
Displaying options in a ComboBox control



We must now populate the ComboBox control with the valid choices. Select the ComboBox control on the form by clicking it with the mouse and locate its `Items` property in the Properties window. The setting of this property is Collection, which means that the `Items` property doesn't have a single value; it's a collection of items (strings, in this case). Click the ellipsis button and you'll see the String Collection Editor dialog box, as shown in Figure 1.11.



C++
C#
Visual Basic
Java
Cobol

FIGURE 1.11
Click the ellipsis button next to the `Items` property of a ComboBox to see the String Collection Editor dialog box.

The main pane in the String Collection Editor dialog box is a TextBox, in which you can enter the items you want to appear in the ComboBox control at runtime. Enter the following strings, one per row and in the order shown here:

C++

C#

Visual Basic

Java

Cobol

Click the OK button to close the dialog box. The items will not appear on the control at design time, but you will see them when you run the project. Before running the project, set one more property. Locate the ComboBox control's `Text` property and set it to **Select your favorite programming language**. This is not an item of the list; it's the string that will initially appear on the control.

You can run the project now and see how the ComboBox control behaves. Press F5 and wait a few seconds. The project will be compiled, and you'll see its form on your Desktop, on top of the Visual Studio window. I'm sure you know how the ComboBox control behaves in a typical Windows application, and our sample application is no exception. You can select an item on the control, either with the mouse or with the keyboard. Click the button with the arrow to expand the list and then select an item with the mouse. Or press the down or up arrow keys to scroll through the list of items. The control isn't expanded, but each time you click an arrow button, the next or previous item in the list appears on the control. Press the Tab key to move the focus to the Button control and press the spacebar to emulate a `Click` event (or simply click the Button control).

We haven't told the application what to do when the button is clicked, so let's go back and add some code to the project. Stop the application by clicking the Stop button on the toolbar (the solid black square) or by choosing Debug ➢ Stop Debugging from the main menu. When the form appears in design mode, double-click the button, and the code window will open, displaying an empty `Click` event handler. Insert the statements shown in Listing 1.2 between the `Private Sub` and `End Sub` statements.

Listing 1.2: The Revised *Click* Event Handler

```
Private Sub Button1_Click(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles Button1.Click
   Dim language As String
   language = ComboBox1.Text
   If language = "Visual Basic" Then
      MsgBox("We have a winner!")
   Else
      MsgBox(language & "is not a bad language.")
   End If
End Sub
```

When the form is first displayed, a string that doesn't correspond to a language is displayed in the ComboBox control. We can preselect one of the items from within our code when the form is first loaded. When a form is loaded, the `Load` event of the Form object is raised. Double-click somewhere on the form and the editor will open the form's `Load` event handler:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles MyBase.Load
End Sub
```

Enter the following code to select the item Visual Basic when the form is loaded:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles MyBase.Load
   ComboBox1.SelectedIndex = 2
End Sub
```

`SelectedIndex` is a property of the ComboBox control that determines the selected item. You can set it to an integer value from within your code to select an item on the control, and you can also use it to retrieve the index of the selected item in the list. Instead of comparing strings, we can compare the `SelectedIndex` property to the value that corresponds to the index of the item Visual Basic, with a statement such as the following:

```
If ComboBox1.SelectedIndex = 2 Then
    MsgBox("We have a winner!")
Else
    MsgBox(ComboBox1.Text & "is not a bad language.")
End If
```

The `Text` property of the ComboBox control returns the text on the control, and we use it to print the selected language's name. Of course, if you insert or remove items from the list, you must edit the code accordingly. If you run the application and test it thoroughly, you'll realize that there's a problem with the ComboBox control. Users can type a new string in the control, which will be interpreted as a language. By default, the ComboBox control allows users to type in something, in addition to selecting an item from the list. To change the control's behavior, select it on the form and locate its `DisplayStyle` property in the Properties window. Expand the list of possible settings for the control and change the property's value from DropDown to DropDown-List. Run the application again and test it; our sample application has become bulletproof. It's a simple application, but you'll see more techniques for building robust applications in Chapter 4, ''GUI Design and Event-Driven Programming.''

The controls on the Toolbox are more than nice pictures we place on our forms. They encapsulate a lot of functionality and expose properties that allow us to adjust their appearance and their functionality. Most properties are usually set at design time, but quite frequently we change the properties of various controls from within our code.

Now that you're somewhat familiar with the process of building Windows applications, and before you look into any additional examples, I will quickly present the components of the Visual Studio IDE.

Understanding the IDE Components

The IDE of Visual Studio 2008 contains numerous components, and it will take you a while to explore them. It's practically impossible to explain in a single chapter what each tool, window, and menu command does. We'll discuss specific tools as we go along and as the topics get more and more advanced. In this section, I will go through the basic items of the IDE — the ones we'll use in the following few chapters to build simple Windows applications.

The IDE Menu

The IDE menu provides the following commands, which lead to submenus. Notice that most menus can also be displayed as toolbars. Also, not all options are available at all times. The options that cannot possibly apply to the current state of the IDE are either invisible or disabled. The Edit menu is a typical example. It's quite short when you're designing the form and quite lengthy when you edit code. The Data menu disappears altogether when you switch to the code editor — you can't use the options of this menu while editing code. If you open an XML document in the IDE, the XML command will be added to the main menu of Visual Studio.

File Menu

The File menu contains commands for opening and saving projects or project items, as well as commands for adding new or existing items to the current project. For the time being, use the New ➢ Project command to create a new project, Open ➢ Project/Solution to open an existing project or solution, Save All to save all components of the current project, and the Recent Projects submenu to open one of the recent projects.

Edit Menu

The Edit menu contains the usual editing commands. Among these commands are the Advanced command and the IntelliSense command. Both commands lead to submenus, which are discussed next. Note that these two items are visible only when you're editing your code, and are invisible while you're designing a form.

Edit ➤ Advanced Submenu

The more-interesting options of the Edit ➤ Advanced submenu are the following:

View White Space Space characters (necessary to indent lines of code and make it easy to read) are replaced by periods.

Word Wrap When a code line's length exceeds the length of the code window, the line is automatically wrapped.

Comment Selection/Uncomment Selection *Comments* are lines you insert between your code's statements to document your application. Every line that begins with a single quote is a comment; it is part of the code, but the compiler ignores it. Sometimes, we want to disable a few lines from our code but not delete them (because we want to be able to restore them later). A simple technique to disable a line of code is to *comment it out* (insert the comment symbol in front of the line). This command allows you to comment (or uncomment) large segments of code in a single move.

Edit ➤ IntelliSense Submenu

The Edit ➤ IntelliSense menu item leads to a submenu with five options, which are described next. IntelliSense is a feature of the editor (and of other Microsoft applications) that displays as much information as possible, whenever possible. When you type the name of a control and the following period, IntelliSense displays a list of the control's properties and methods, so that you can select the desired one, rather than guessing its name. When you type the name of a function and the opening parenthesis, IntelliSense will display the syntax of the function — its arguments. The IntelliSense submenu includes the following options:

List Members When this option is on, the editor lists all the members (properties, methods, events, and argument list) in a drop-down list. This list will appear when you enter the name of an object or control followed by a period. Then you can select the desired member from the list with the mouse or with the keyboard. Let's say your form contains a control named *`TextBox1`* and you're writing code for this form. When you enter the name of the control followed by a period (**`TextBox1.`**), a list with the members of the TextBox control will appear (as seen in Figure 1.12).

In addition, a description of the selected member is displayed in a ToolTip box, as you can see in the same figure. Select the `Text` property and then enter the equal sign, followed by a string in quotes, as follows:

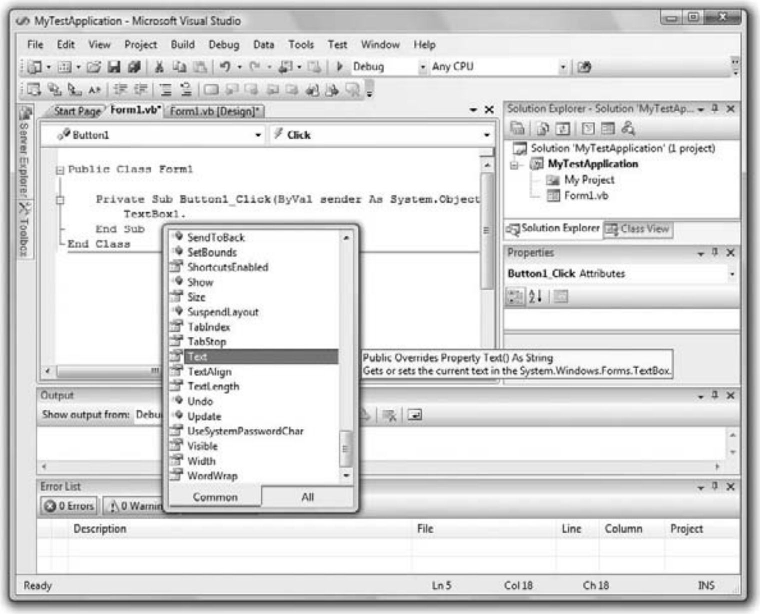
```
TextBox1.Text = "Your User Name"
```

If you select a property that can accept a limited number of settings, you will see the names of the appropriate constants in a drop-down list. If you enter the following statement, you will see the constants you can assign to the property (see Figure 1.13):

```
TextBox1.TextAlign =
```

Again, you can select the desired value with the mouse. The drop-down list with the members of a control or object (the Members list) remains open until you type a terminator key (the Esc or End key) or select a member by pressing the space bar or the Enter key.

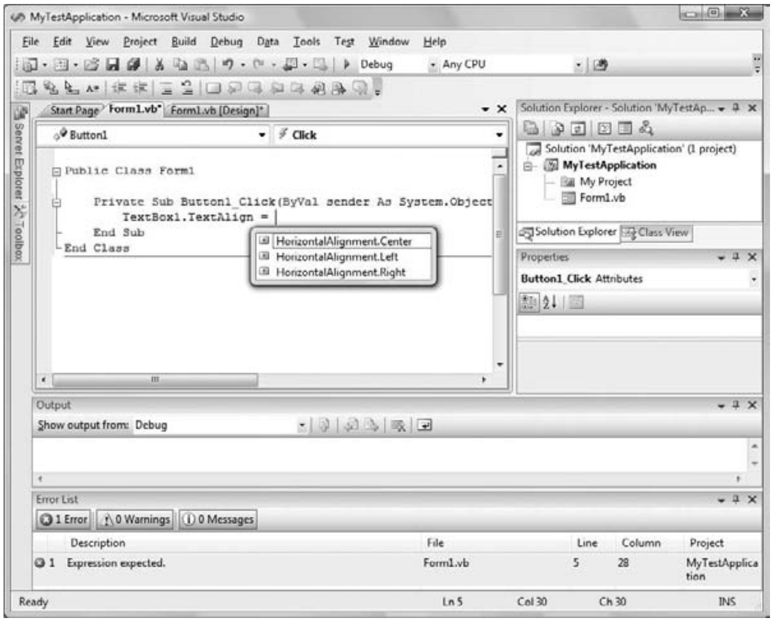
FIGURE 1.12
Viewing the members of a control in the IntelliSense drop-down list



```
Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object
        TextBox1.
    End Sub
End Class
```

Public Overrides Property Text() As String
Gets or sets the current text in the System.Windows.Forms.TextBox.

FIGURE 1.13
Viewing the possible settings of a property in the IntelliSense drop-down list



```
Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object
        TextBox1.TextAlign =
    End Sub
End Class
```

	Description	File	Line	Column	Project
1	Expression expected.	Form1.vb	5	28	MyTestApplication

Parameter Info While editing code, you can move the pointer over a variable, method, or property and see its declaration in a yellow pop-up box. You can also jump to the variable's definition or the body of a procedure by choosing Go To Definition from the context menu that will appear if you right-click the variable or method name in the code window.

Quick Info This is another IntelliSense feature that displays information about commands and functions. When you type the opening parenthesis following the name of a function, for example, the function's arguments will be displayed in a ToolTip box (a yellow horizontal box). The first argument appears in bold font; after entering a value for this argument, the next one is shown in bold. If an argument accepts a fixed number of settings, these values will appear in a drop-down list, as explained previously.

Complete Word The Complete Word feature enables you to complete the current word by pressing Ctrl+spacebar. For example, if you type *TextB* and then press Ctrl+spacebar, you will see a list of words that you're most likely to type (TextBox, TextBox1, and so on).

Insert Snippet This command opens the Insert Snippet window at the current location in the code editor window. Code snippets, which are an interesting feature of Visual Studio 2008, are discussed in the section ''Using Code Snippets'' later in this chapter.

Edit ➤ Outlining Submenu

A practical application contains a substantial amount of code in a large number of event handlers and custom procedures (subroutines and functions). To simplify the management of the code window, the Outlining submenu contains commands that collapse and expand the various procedures.

Let's say you're finished editing the `Click` event handlers of several buttons on the form. You can reduce these event handlers to a single line that shows the names of the procedures and a plus sign in front of them. You can expand a procedure's listing at any time by clicking the plus sign in front of its name. When you do so, a minus sign appears in front of the procedure's name, and you can click it to collapse the body of the procedure again. The Outlining submenu contains commands to handle the outlining of the various procedures, or turn off outlining and view the complete listings of all procedures. You will use these commands as you write applications with substantial amounts of code:

Toggle Outlining Expansion This option lets you change the outline mode of the current procedure. If the procedure's definition is collapsed, the code is expanded, and vice versa.

Toggle All Outlining This option is similar to the Toggle Outlining Expansion option, but it toggles the outline mode of the current document. A form is reduced to a single statement. A file with multiple classes is reduced to one line per class.

Stop Outlining This option turns off outlining and adds a new command to the Outlining submenu, Start Automatic Outlining, which you can select to turn on automatic outlining again.

Collapse To Definitions This option reduces the listing to a list of procedure headers.

View Menu

This menu contains commands to display any toolbar or window of the IDE. You have already seen the Toolbars menu (in the ''Starting a New Project'' section). The Other Windows command leads to a submenu with the names of some standard windows, including the Output and Command windows. The Output window is the console of the application. The compiler's messages, for example, are displayed in the Output window. The Command window allows you to enter and execute statements. When you debug an application, you can stop it and enter VB statements in the Command window.

Project Menu

This menu contains commands for adding items to the current project (an item can be a form, a file, a component, or even another project). The last option in this menu is the Project Properties command, which opens the project's Properties Pages. The Add Reference and Add Web Reference commands allow you to add references to .NET components and web components, respectively.

Build Menu

The Build menu contains commands for building (compiling) your project. The two basic commands in this menu are Build and Rebuild All. The Build command compiles (builds the executable) of the entire solution, but it doesn't compile any components of the project that haven't changed since the last build. The Rebuild All command does the same, but it clears any existing files and builds the solution from scratch.

Debug Menu

This menu contains commands to start or end an application, as well as the basic debugging tools. The basic commands of this menu are discussed briefly in Chapter 4 and in Appendix B.

Data Menu

This menu contains commands you will use with projects that access data. You'll see how to use this short menu's commands in the discussion of the visual database tools in Chapters 21 and 22 of the book.

Format Menu

The Format menu, which is visible only while you design a Windows or web form, contains commands for aligning the controls on the form. The commands of this menu are discussed in Chapter 4. The Format menu is invisible when you work in the code editor — its commands apply to the visible elements of the interface.

Tools Menu

This menu contains a list of useful tools, such as the Macros command, which leads to a submenu with commands for creating macros. Just as you can create macros in a Microsoft Office application to simplify many tasks, you can create macros to automate many of the repetitive tasks you perform in the IDE. The last command in this menu, the Options command, leads to the Options dialog box, in which you can fully customize the environment. The Choose Toolbox Items command opens a dialog box that enables you to add more controls to the Toolbox. In Chapter 12, ''Building Custom Windows Controls,'' you'll learn how to design custom controls and add them to the Toolbox.

Window Menu

This is the typical Window menu of any Windows application. In addition to the list of open windows, it also contains the Hide command, which hides all toolboxes, leaving the entire window of the IDE devoted to the code editor or the Form Designer. The toolboxes don't disappear completely; they're all retracted, and you can see their tabs on the left and right edges of the IDE window. To expand a toolbox, just hover the mouse pointer over the corresponding tab.

Help Menu

This menu contains the various help options. The Dynamic Help command opens the Dynamic Help window, which is populated with topics that apply to the current operation. The Index command opens the Index window, in which you can enter a topic and get help on the specific topic.

Toolbox Window

The Toolbox window contains all the controls you can use to build your application's interface. This window is usually retracted, and you must move the pointer over it to view the Toolbox. The controls in the Toolbox are organized in various tabs, so take a look at them to become familiar with the controls and their functions.

In the first few chapters, we'll work with the controls in the Common Controls and Menus & Toolbars tabs. The Common Controls tab contains the icons of the most common Windows controls. The Data tab contains the icons of the objects you will use to build data-driven applications (they're explored later in this book). The Dialogs tab contains controls for implementing the common dialog controls, which are so common in Windows interfaces; they're discussed in Chapter 8, ''More Windows Controls.''

Solution Explorer Window

The Solution Explorer window contains a list of the items in the current solution. A *solution* can contain multiple projects, and each project can contain multiple items. The Solution Explorer displays a hierarchical list of all the components, organized by project. You can right-click any component of the project and choose Properties in the context menu to see the selected component's properties in the Properties window. If you select a project, you will see the Project Properties dialog box. You will find more information on project properties in the following chapter.

If the solution contains multiple projects, you can right-click the project you want to become the startup form and select Set As StartUp Project. You can also add items to a project with the Add Item command of the context menu, or remove a component from the project with the Exclude From Project command. This command removes the selected component from the project, but doesn't affect the component's file on the disk. The Delete command removes the selected component from the project and also deletes the component's file from the disk.

Properties Window

This window (also known as the Properties Browser) displays all the properties of the selected component and its settings. Every time you place a control on a form, you switch to this window to adjust the appearance of the control. You have already seen how to manipulate the properties of a control through the Properties window.

Many properties are set to a single value, such as a number or a string. If the possible settings of a property are relatively few, they're displayed as meaningful constants in a drop-down list. Other properties are set through a more elaborate interface. Color properties, for example, are set from within a Color dialog box that's displayed right in the Properties window. Font properties are set through the usual Font dialog box. Collections are set in a Collection Editor dialog box, in which you can enter one string for each item of the collection, as you did for the items of the ComboBox control earlier in this chapter.

If the Properties window is hidden, or if you have closed it, you can either choose View ➤ Properties Window, or right-click a control on the form and choose Properties. Or you can simply press F4 to bring up this window. There will be times when a control might totally overlap another control, and you won't be able to select the hidden control and view its properties. In this case, you can select the desired control in the ComboBox at the top of the Properties window. This box contains the names of all the controls on the form, and you can select a control on the form by selecting its name on this box.

Output Window

The Output window is where many of the tools, including the compiler, send their output. Every time you start an application, a series of messages is displayed in the Output window. These messages are generated by the compiler, and you need not understand them at this point. If the Output window is not visible, choose View ➤ Other Windows ➤ Output from the menu.

Command and Immediate Windows

While testing a program, you can interrupt its execution by inserting a so-called *breakpoint*. When the breakpoint is reached, the program's execution is suspended, and you can execute a statement in the Immediate window. Any statement that can appear in your VB code can also be executed in the Immediate window. To evaluate an expression, enter a question mark followed by the expression you want to evaluate, as in the following samples, where `result` is a variable in the program you interrupted:

```
? Math.Log(35)
? "The answer is " & result.ToString
```

You can also send output to this window from within your code with the `Debug.Write` and `Debug.WriteLine` methods. Actually, this is a widely used debugging technique — to print the values of certain variables before entering a problematic area of the code. There are more elaborate tools to help you debug your application, and you'll find a discussion in Appendix B, but printing a few values to the Immediate window is a time-honored practice in programming with VB.

In many of the examples of this book, especially in the first few chapters, I use the `Debug.WriteLine` statement to print something to the Immediate window. To demonstrate the use of the `DateDiff()` function, for example, I'll use a statement like the following:

```
Debug.WriteLine(DateDiff(DateInterval.Day, #3/9/2007#, #5/15/2008#))
```

When this statement is executed, the value 433 will appear in the Immediate window. This statement demonstrates the syntax of the `DateDiff()` function, which returns the difference between the two dates in days. Sending some output to the Immediate window to test a function or display the results of intermediate calculations is a common practice.

To get an idea of the functionality of the Immediate window, switch back to your first sample application and insert the `Stop` statement after the `End If` statement in the button's `Click` event handler. Run the application, select a language, and click the button on the form. After displaying a message box, the application will reach the `Stop` statement and its execution will be suspended.

You'll see the Immediate window at the bottom of the IDE. If it's not visible, open the Debug menu and choose Windows ➤ Immediate. In the Immediate window, enter the following statement:

```
? ComboBox1.Items.Count
```

Then press Enter to execute it. Notice that IntelliSense is present while you're typing in the Immediate window. The expression prints the number of items in the ComboBox control. (Don't worry about the numerous properties of the control and the way I present them here; they're discussed in detail in Chapter 6, ''Basic Windows Controls.'') As soon as you press Enter, the value 5 will be printed on the following line.

You can also manipulate the controls on the form from within the Immediate window. Enter the following statement and press Enter to execute it:

```
ComboBox1.SelectedIndex = 4
```

The fifth item on the control will be selected (the indexing of the items begins with 0). However, you can't see the effects of your changes, because the application isn't running. Press F5 to resume the execution of the application and you will see that the item Cobol is now selected in the ComboBox control.

The Immediate window is available only while the application's execution is suspended. To continue experimenting with it, click the button on the form to evaluate your choice. When the `Stop` statement is executed again, you'll be switched to the Immediate window.

Unlike the Immediate window, the Command window is available at design time. The Command window allows you to access all the commands of Visual Studio by typing their names in this window. If you enter the string **`Edit`** followed by a period, you will see a list of all commands of the Edit menu, including the ones that are not visible at the time, and you can invoke any of these commands and pass arguments to them. For example, if you enter **`Edit.Find "Margin"`** in the Command window and then press Enter, the first instance of the string *Margin* will be located in the open code window. To start the application, you can type `Debug.Start`. You can add a new project to the current solution with the AddProj command, and so on. Most developers hardly ever use this window in designing or debugging applications.

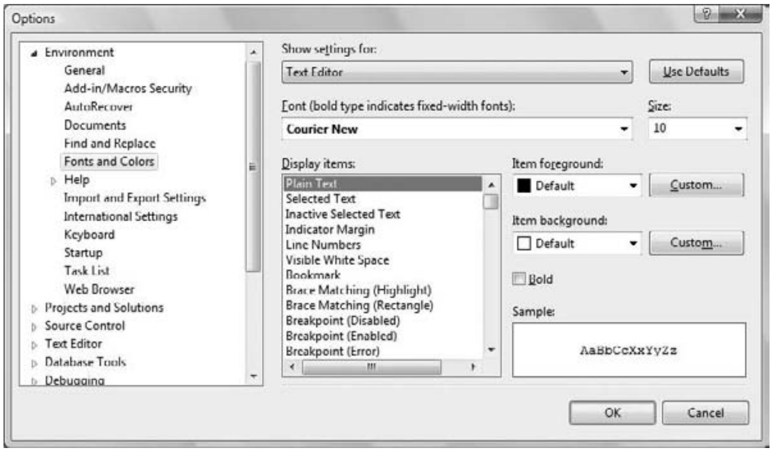
Error List Window

This window is populated by the compiler with error messages, if the code can't be successfully compiled. You can double-click an error message in this window, and the IDE will take you to the line with the statement in error — which you should fix. Change the `MsgBox()` function name to **`MssgBox()`**. As soon as you leave the line with the error, the name of the function will be underlined with a wiggly red line and the following error description will appear in the Error List window:

```
Name 'MssgBox' is not declared
```

Setting Environment Options

The Visual Studio IDE is highly customizable. I will not discuss all the customization options here, but I will show you how to change the default settings of the IDE. Open the Tools menu and select Options (the last item in the menu). The Options dialog box appears, in which you can set all the options regarding the environment. Figure 1.14 shows the options for the fonts of the various items of the IDE. Here you can set the font for the Text Editor, dialog boxes, toolboxes, and so on. Select an item in the tree in the left pane list and then set the font for this item in the box below.



Options

Environment
- General
- Add-in/Macros Security
- AutoRecover
- Documents
- Find and Replace
- Fonts and Colors
- Help
- Import and Export Settings
- International Settings
- Keyboard
- Startup
- Task List
- Web Browser

Projects and Solutions
Source Control
Text Editor
Database Tools
Debugging

Show settings for: Text Editor

Font (bold type indicates fixed-width fonts): Courier New

Size: 10

Display items: Plain Text, Selected Text, Inactive Selected Text, Indicator Margin, Line Numbers, Visible White Space, Bookmark, Brace Matching (Highlight), Brace Matching (Rectangle), Breakpoint (Disabled), Breakpoint (Enabled), Breakpoint (Error)

Item foreground: Default

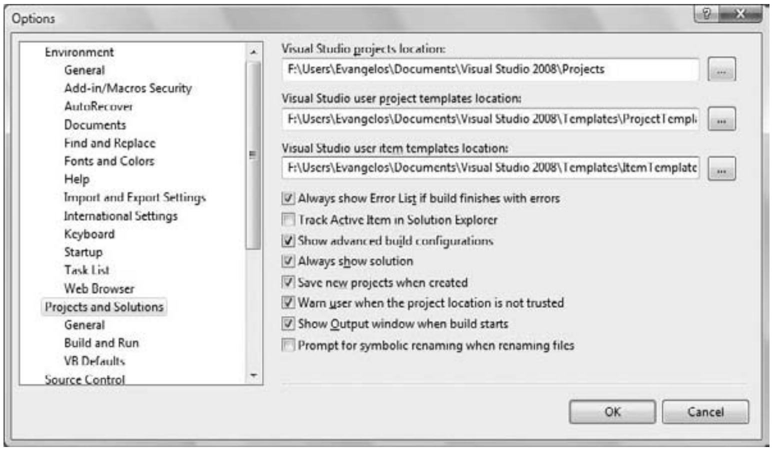
Item background: Default

Bold

Sample: AaBbCcXxYyZz

FIGURE 1.14
The Fonts And Colors options

Figure 1.15 shows the Projects And Solutions options. The top box indicates the default location for new projects. The Save New Projects When Created check box determines whether the editor will create a new folder for the project when it's created. If you uncheck this box, then Visual Studio will create a folder in the Temp folder. Projects in the Temp folder will be removed when you run the Disk Cleanup utility to claim more space on your hard drives.



Options

Environment
- General
- Add-in/Macros Security
- AutoRecover
- Documents
- Find and Replace
- Fonts and Colors
- Help
- Import and Export Settings
- International Settings
- Keyboard
- Startup
- Task List
- Web Browser

Projects and Solutions
- General
- Build and Run
- VB Defaults

Source Control

Visual Studio projects location: F:\Users\Evangelos\Documents\Visual Studio 2008\Projects

Visual Studio user project templates location: F:\Users\Evangelos\Documents\Visual Studio 2008\Templates\ProjectTempl

Visual Studio user item templates location: F:\Users\Evangelos\Documents\Visual Studio 2008\Templates\ItemTemplate

- [✓] Always show Error List if build finishes with errors
- [ ] Track Active Item in Solution Explorer
- [✓] Show advanced build configurations
- [✓] Always show solution
- [✓] Save new projects when created
- [✓] Warn user when the project location is not trusted
- [✓] Show Output window when build starts
- [ ] Prompt for symbolic renaming when renaming files

FIGURE 1.15
The Projects And Solutions options

By default, Visual Studio saves the changes to the current project every time you press F5. You can change this behavior by setting the Before Building option in the Build And Run page, under the Project And Solutions branch. If you change this setting, you must save your project from time to time with the File ➢ Save All command.

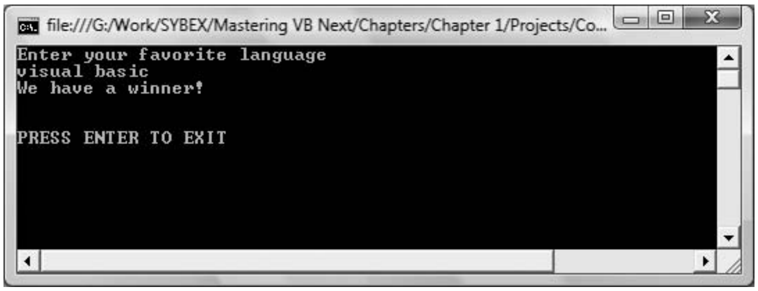
Most of the tabs in the Options dialog box are straightforward, and you should take a look at them. If you don't like some of the default aspects of the IDE, this is the place to change them. If you switch to the Basic item under the Text Editor branch of the tree in the left pane of the Options dialog box, you will find the Line Numbers option. Select this check box to display numbers in front of each line in the code window. The Options dialog box contains a lot of options for customizing your work environment, and it's worth exploring on your own.

Building a Console Application

Apart from Windows applications, you can use Visual Studio 2008 to build applications that run in a command prompt window. The command prompt window isn't really a DOS window, even though it looks like one. It's a text window, and the only way to interact with an application is to enter lines of text and read the output generated by the application, which is displayed in this text window, one line at a time. This type of application is called a *console application*, and I'm going to demonstrate console applications with a single example. We will not return to this type of application later in the book because it's not what you're supposed to do as a Windows developer.

The console application you'll build in this section, ConsoleApplication1, prompts the user to enter the name of her favorite language. It then prints the appropriate message on a new line, as shown in Figure 1.16.

FIGURE 1.16
A console application uses the command prompt window to interact with the user.



file:///G:/Work/SYBEX/Mastering VB Next/Chapters/Chapter 1/Projects/Co...

```
Enter your favorite language
visual basic
We have a winner!


PRESS ENTER TO EXIT
```

Start a new project. In the New Project dialog box, select the template Console Application. You can also change its default name from ConsoleApplication1 to a more descriptive name. For this example, don't change the application's name.

A console application doesn't have a user interface, so the first thing you'll see is the code editor's window with the following statements:

```
Module Module1

    Sub Main()

    End Sub

End Module
```

Unlike a Windows application, which is a class, a console application is a module. `Main()` is the name of a subroutine that's executed automatically when you run a console application. The code you want to execute must be placed between the statements `Sub Main()` and `End Sub`. Insert the statements shown in Listing 1.3 in the application's `Main()` subroutine.

Listing 1.3: Console Application

```
Module Module1
   Sub Main()
      Console.WriteLine("Enter your favorite language")
      Dim language As String
      language = Console.ReadLine()
      language = language.ToUpper
      If language = "VISUAL BASIC" Or _
              language = "VB" Or
              language = "VB.NET" Then
         Console.WriteLine("We have a winner!")
      Else
         Console.WriteLine(language & "is not a bad language.")
      End If
      Console.WriteLine()
      Console.WriteLine()
      Console.WriteLine("PRESS ENTER TO EXIT")
      Console.ReadLine()
   End Sub
End Module
```

This code is quite similar to the code of the equivalent Windows applications we developed earlier, except that it uses the `Console.WriteLine` statement to send its output to the command prompt window instead of a message box.

A console application doesn't react to events because it has no visible interface. However, it's easy to add some basic elements of the Windows interface to a console application. If you change the `Console.WriteLine` method call into the `MsgBox()` function, the message will be displayed in a message box.

The reason to build a console application is to test a specific feature of the language without having to build a user interface. Many of the examples in the documentation are console applications; they demonstrate the topic at hand and nothing more. If you want to test the `DateDiff()` function, for example, you can create a new console application and enter the lines of Listing 1.4 in its `Main()` subroutine.

Listing 1.4: Testing the *DateDiff()* Function with a Console Application

```
Sub Main()
   Console.WriteLine(DateDiff(DateInterval.Day, #3/9/2000#, #5/15/2008#))
   Console.WriteLine("PRESS ENTER TO EXIT")
   Console.ReadLine()
End Sub
```

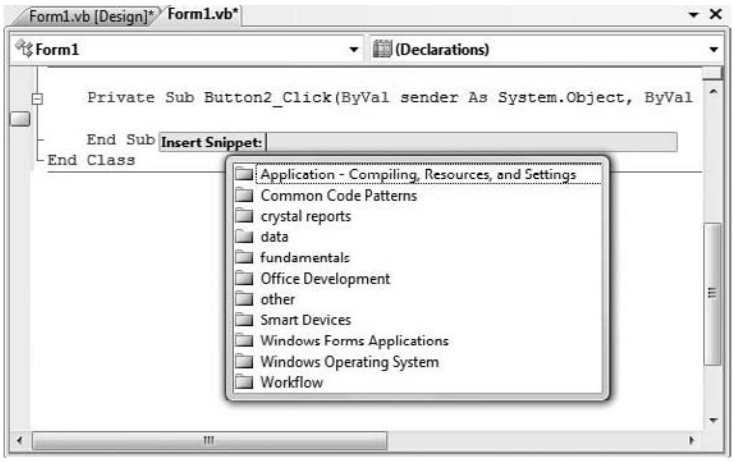
The last two lines will be the same in every console application you write. Without them, the command prompt window will close as soon as the `End Sub` statement is reached, and you won't have a chance to see the result. The `Console.ReadLine` method waits until the user presses the Enter key.

Console applications are convenient for testing short code segments, but Windows programming is synonymous with designing graphical user interfaces, so you won't find any more console applications in this book.

Using Code Snippets

Visual Basic 2008 comes with a lot of predefined code snippets for selected actions, and you can insert these snippets in your code as needed. Let's say you want to insert the statements for writing some text to a file, but you have no idea how to access files. Create an empty line in the listing (press the Enter key a couple of times at the end of a code line). Then open the Edit menu and choose IntelliSense ➢ Insert Snippet (or right-click somewhere in the code window and choose Insert Snippet from the context menu).

You will see on the screen a list of the snippets, organized in folders according to their function, as shown in Figure 1.17. Select the *fundamentals* folder, which will display another list of options: *collections and arrays*, *datatypes*, *filesystem*, and *math*. Double-click the *filesystem* item to see a list of common file-related tasks, as shown in Figure 1.18. Locate the item Write Text To A File in the list and double-click it to insert the appropriate snippet at the current location in the code window.



```
    Private Sub Button2_Click(ByVal sender As System.Object, ByVal

    End Sub
End Class
```

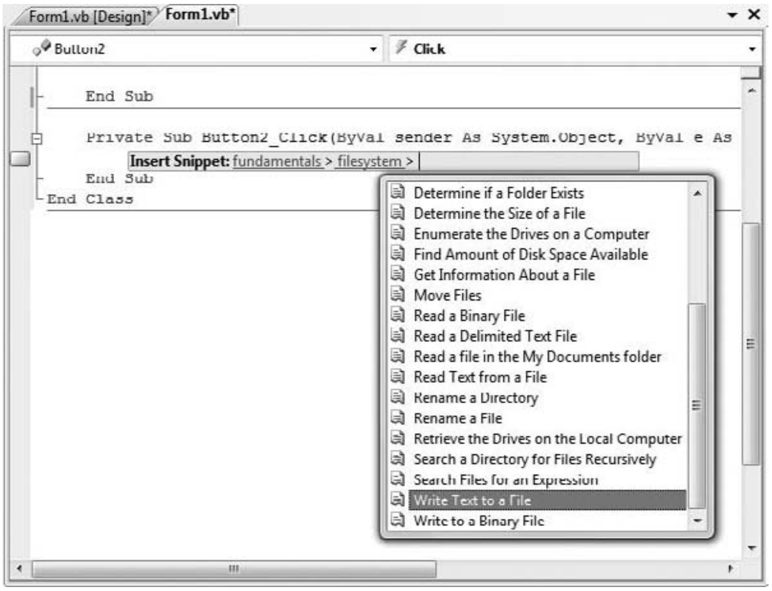
FIGURE 1.17
The code snippets organized according to their function

The following snippet will be inserted in your code:

```
My.Computer.FileSystem.WriteAllText("C:\test.txt", "Text", True)
```

To write some text to a file, you need to call the `WriteAllText` method of the My.Computer .FileSystem object. You can replace the strings shown in the snippet with actual values. The first string is the filename, the second string is the text to be written to the file, and the last argument of the method determines whether the text will be appended to the file (if False) or will overwrite any existing text (if True).

The snippet shows you the basic statements for performing a common task, and you can edit the code inserted by Visual Studio as needed. A real-world application would probably prompt the user for a filename via the File common dialog box and then use the filename specified by the user in the dialog box, instead of a hard-coded file name.


Form1.vb [Design]*
Form1.vb*
Button2
Click

```
    End Sub

    Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As
        Insert Snippet: fundamentals > filesystem >
    End Sub
End Class
```

Determine if a Folder Exists
Determine the Size of a File
Enumerate the Drives on a Computer
Find Amount of Disk Space Available
Get Information About a File
Move Files
Read a Binary File
Read a Delimited Text File
Read a file in the My Documents folder
Read Text from a File
Rename a Directory
Rename a File
Retrieve the Drives on the Local Computer
Search a Directory for Files Recursively
Search Files for an Expression
Write Text to a File
Write to a Binary File

FIGURE 1.18
Selecting a code snippet to insert in your code

As you program, you should always try to find out whether there's a snippet for the task at hand. Sometimes you can use a snippet without even knowing how it works. Although snippets can simplify your life, they won't help you understand the Framework, which is discussed in detail throughout this book.

Using the My Object

You have probably noticed that the code snippets of Visual Studio use an entity called My, which is a peculiar object that was introduced with VB 2005 to simplify many programming tasks. As you saw in the preceding code snippet, the My object allows you to write some text to a file with a single statement, the `WriteAllText` method. If you're familiar with earlier versions of Visual Basic, you know that you must first open a file, and then write some text to it, and finally close the file. The My object allows you to perform all these operations with a single statement, as you saw in the preceding example.

Another example is the `Play` method, which you can use to play back a WAV file from within your code:

```
My.Computer.Audio.Play ("C:\Sounds\CountDown.wav")
```

You can also use the following expression to play back a system sound:

```
My.Computer.Audio.PlaySystemSound(System.Media.SystemSounds.Exclamation)
```

The method that plays back the sound is the `Play` method, and the method that writes text to a file is the `WriteAllText` method. However, you can't call them directly through the My object; they're not methods of the My object. If they were, you'd have to dig hard to find out the method you need. The My object exposes six components, which contain their own components. Here's a description of the basic components of the My object and the functionality you should expect to find in each component:

My.Application The Application component provides information about the current application. The `CommandLineArgs` property of My.Application returns a collection of strings, which are the arguments passed to the application when it was started. Typical Windows applications aren't called with command-line arguments, but it's possible to start an application and pass a filename as an argument to the application (the document to be opened by the application, for example). The `Info` property is an object that exposes properties such as `DirectoryPath` (the application's default folder), `ProductName`, `Version`, and so on.

Computer This component of the My object exposes a lot of functionality via a number of properties, many of which are objects. The My.Computer.Audio component lets you play back sounds. The My.Computer.Clipboard component lets you access the Clipboard. To find out whether the Clipboard contains a specific type of data, use the `ContainsText`, `ContainsImage`, `ContainsData`, and `ContainsAudio` methods. To retrieve the contents of the Clipboard, use the `GetText`, `GetImage`, `GetData`, and `GetAudioStream` methods. Assuming that you have a form with a TextBox control and a PictureBox control, you can retrieve text or image data from the Clipboard and display it on the appropriate control with the following statements:

```
If My.Computer.Clipboard.ContainsImage Then
    PictureBox1.Image = My.Computer.Clipboard.GetImage
End If
If My.Computer.Clipboard.ContainsText Then
    TextBox2.Text = My.Computer.Clipboard.GetText
End If
```

You may have noticed that using the My object in your code requires that you write long statements. You can shorten them substantially via the `With` statement, as shown next:

```
With My.Computer.Clipboard
    If .ContainsImage Then
        PictureBox1.Image = .GetImage
    End If
    If .ContainsText Then
        TextBox2.Text = .GetText
    End If
End With
```

When you're executing multiple statements on the same object, you can specify the object in a `With` statement and call its methods in the block of the `With` statement by specifying the method name prefixed with a period. The `With` statement is followed by the name of the object to which all following methods apply, and is terminated with the `End With` statement.

Another property of the My.Computer component is the FileSystem object that exposes all the methods you need to access files and folders. If you enter the expression `My.Computer.FileSystem` followed by a period, you will see all the methods exposed by the FileSystem component. Among them, you will find `DeleteFile`, `DeleteDirectory`, `RenameFile`, `RenameDirectory`, `WriteAllText`, `ReadAllText`, and many more. Select a method and then type the opening parenthesis. You will see the syntax of the method in a ToolTip. The syntax of the `CopyFile` method is as follows:

```
My.Computer.FileSystem.CopyFile( _
         sourceFileName As String, destinationFileName As String)
```

Just specify the path of the file you want to copy and the new file's name, and you're finished. This statement will copy the specified file to the specified location.

You will notice that the ToolTip box with the syntax of the `CopyFile` method has multiple versions, which are listed at the left side of the box along with arrow up and arrow down icons. Click these two buttons to see the next and previous versions of the method. The second version of the `CopyFile` method is as follows:

```
My.Computer.FileSystem.CopyFile( _
         sourceFileName As String, destinationFileName As String, _
         overwrite As Boolean)
```

The *`overwrite`* argument specifies whether the method should overwrite the destination file if it exists.

The third version of the method accepts a different third argument that determines whether the usual copy animation will be displayed as the file is being copied.

The various versions of the same method differ in the number and/or type of their arguments, and they're called overloaded forms of the method. Instead of using multiple method names for the same basic operation, the overloaded forms of a method allow you to call the same method name and adjust its behavior by specifying different arguments.

Forms This component lets you access the forms of the current application. You can also access the application's forms by name, so the Forms component isn't the most useful one.

Settings This component lets you access the application settings. These settings apply to the entire application and are stored in an XML configuration file. The settings are created from within Visual Studio, and you use the Settings component to read them.

User This component returns information about the current user. The most important property of the User component is the `CurrentPrincipal` property, which is an object that represents the credentials of the current user.

WebServices The WebServices component represents the web services referenced by the current application.

The My object gives beginners unprecedented programming power and allows you to perform tasks that would require substantial code if implemented with earlier versions of the language, not to mention the research it would take to locate the appropriate methods in the Framework. You can explore the My object on your own and use it as needed. My is not a substitute for learning the language and the Framework. It can help you initially, but you can't go far without learning the methods of the Framework for handling files or any other feature.

Let's say you want to locate all the files of a specific type in a folder, including its subfolders. Scanning a folder and its subfolders to any depth is quite a task (you'll find the code in Chapter 15, ''Accessing Folders and Files''). You can do the same with a single statement by using the My object:

```
Dim files As ReadOnlyCollection(Of String)
files = My.Computer.FileSystem.GetFiles("D:\Data", True, "*.txt")
```

The `GetFiles` method populates the *`files`* collection with the pathnames of the text files in the folder `D:\Data` and its subfolders. However, it won't help you if you want to process each file in place. Moreover, this `GetFiles` method is synchronous: If the folder contains many subfolders with many files, it will block the interface until it retrieves all the files. In Chapter 15, you'll see the code that retrieves filenames and adds them to a control as it goes along.

If you're already familiar with VB, you may think that the My object is an aid for the absolute beginner or the nonprogrammer. This isn't true. VB is about productivity, and the My object can help you be more productive with your daily tasks, regardless of your knowledge of the language or programming skills. If you can use My to save a few (or a few dozen) statements, do it. There's no penalty for using the My object, because the compiler replaces the methods of the My object with the equivalent method calls to the Framework.

The Bottom Line

Navigate the integrated development environment of Visual Studio. To simplify the process of application development, Visual Studio provides an environment that's common to all languages, known as an integrated development environment (IDE). The purpose of the IDE is to enable the developer to do as much as possible with visual tools, before writing code. The IDE provides tools for designing, executing, and debugging your applications. It's your second desktop, and you'll be spending most of your productive hours in this environment.

Master It Describe the basic components of the Visual Studio IDE.

Understand the basics of a Windows application. A Windows application consists of a *visual interface* and *code*. The visual interface is what users see at runtime: a form with controls with which the user can interact — by entering strings, checking or clearing check boxes, clicking buttons, and so on. The visual interface of the application is designed with visual tools. The visual elements incorporate a lot of functionality, but you need to write some code to react to user actions.

Master It Describe the process of building a simple Windows application.

Chapter 2

Variables and Data Types

This chapter and the next discuss the fundamentals of any programming language: variables and data types. A *variable* stores data, which are processed with statements. A program is a list of statements that manipulate variables. To write even simple applications, you need a basic understanding of some fundamental topics, such as the data types (the kind of data you can store in a variable), the scope and lifetime of variables, and how to write procedures and pass arguments to them. In this chapter, we'll explore the basic data types of Visual Basic, and in the following one, you'll learn about procedures and flow-control statements.

If you're new to Visual Basic, you may find some material in this chapter less than exciting. It covers basic concepts and definitions — in general, tedious, but necessary, material. Think of this chapter as a prerequisite for the following ones. If you need information on core features of the language as you go through the examples in the rest of the book, you'll probably find it here.

In this chapter, you'll learn how to do the following:

- Declare and use variables
- Use the native data types
- Create custom data types
- Use arrays

Variables

In Visual Basic, as in any other programming language, variables store values during a program's execution. A variable has a name and a value. The variable *`UserName`*, for example, can have the value `Joe`, and the variable *`Discount`* can have the value `0.35`. *`UserName`* and *`Discount`* are variable names, and `Joe` and `0.35` are their values. `Joe` is a *string* (that is, text or an alphanumeric value), and `0.35` is a numeric value. When a variable's value is a string, it must be enclosed in double quotes. In your code, you can refer to the value of a variable by the variable's name.

In addition to a name and a value, variables have a *data type*, which determines what kind of values we can store to a variable. VB 2008 supports several data types (and they're discussed in detail later in this chapter). It's actually the Common Language Runtime (CLR) that supports the data types, and they're common to all languages, not just to Visual Basic. The data type of a variable is specified when the variable is declared, and you should always declare variables before using them. To declare a variable, enter the `Dim` statement, followed by the variable's name, the `As` keyword, and the variable's type:

```
Dim Amount As Decimal
```

Decimal is a numeric data type; it can store both integer and noninteger values. For example, the following statements calculate and display the discount for the amount of $24,500:

```
Dim Amount As Decimal
Dim Discount As Decimal
Dim DiscountedAmount As Decimal
Amount = 24500
Discount = 0.35
DiscountedAmount = Amount * (1 - Discount)
MsgBox("Your price is $" & DiscountedAmount.ToString)
```

If you enter these statements in a button's `Click` event handler to test them, the compiler may underline the statement that assigns the value `0.35` to the *`Discount`* variable and generate an error message. To view the error message, hover the pointer over the underlined segment of the statement in error. This will happen if the Strict option is on. (I discuss the Strict option, along with two more options of the compiler, later in this chapter.) By default, the Strict option is off and the statement should generate an error.

The compiler treats any numeric value with a fractional part as a Double value and detects that you're attempting to assign a Double value to a Decimal variable. To convert the numeric value to the Decimal type, use the following notation:

```
Discount = 0.35D
```

As you will see later, the *D* character at the end of a numeric value indicates that the value should be treated as a Decimal value, and there are a few more *type characters* (see Table 2.2 later in this chapter). I've used the Decimal data type here because it's commonly used in financial calculations. The message that this expression displays depends on the values of the *`Discount`* and *`Amount`* variables. If you decide to offer a better discount, all you have to do is change the value of the *`Discount`* variable. If you didn't use the *`Discount`* variable, you'd have to make many changes throughout your code. In other words, if you coded the line that calculated the discounted amount as follows, you'd have to look for every line in your code that calculates discounts and change the discount from 0.35 to another value:

```
DiscountedAmount = 24500 * (1 - 0.35)
```

By changing the value of the *`Discount`* variable in a single place in your code, the entire program is up-to-date.

Declaring Variables

In most programming languages, variables must be declared in advance. Historically, the reason for doing this has been to help the compiler generate the most efficient code. If the compiler knows all the variables and their types ahead of time, it can produce the most compact and efficient, or optimized, code. For example, when you tell the compiler that the variable *`Discount`* will hold a number, the compiler sets aside a certain number of bytes for the *`Discount`* variable to use.

One of the most popular, yet intensely criticized, features of BASIC was that it didn't force the programmer to declare all variables. As you will see, there are more compelling reasons than speed and efficiency for declaring variables. For example, when a compiler knows the types of variables in advance, it can catch many errors at design or compile time — errors that otherwise would surface at runtime. When you declare a variable as *Date*, the compiler won't let you assign an integer value to it.

When programming in VB 2008, you should declare your variables because this is the default mode, and Microsoft recommends this practice strongly. If you attempt to use an undeclared variable in your code, VB 2008 will throw an exception. It will actually catch the error as soon as you complete the line that uses the undeclared variable, underlining it with a wiggly line. It is possible to change the default behavior and use undeclared variables the way most people did with earlier versions of VB (you'll see how this is done in the section ''The Strict, Explicit, and Infer Options,'' later in this chapter), but all the examples in this book use explicitly declared variables. In any case, you're strongly encouraged to declare your variables.

To declare a variable, use the `Dim` statement followed by the variable's name, the `As` keyword, and its type, as follows:

```
Dim meters As Integer
Dim greetings As String
```

The first variable, *`meters`*, will store integers, such as 3 or 1,002; the second variable, *`greetings`*, will store text. You can declare multiple variables of the same or different type in the same line, as follows:

```
Dim Qty As Integer, Amount As Decimal, CardNum As String
```

If you want to declare multiple variables of the same type, you need not repeat the type. Just separate all the variables of the same type with commas and set the type of the last variable:

```
Dim Length, Width, Height As Integer, Volume, Area As Double
```

This statement declares three Integer variables and two Double variables. Double variables hold fractional values (or *floating-point values*, as they're usually called) that are similar to the Single data type, except that they can represent noninteger values with greater accuracy.

You can use other keywords in declaring variables, such as `Private`, `Public`, and `Static`. These keywords are called *access modifiers* because they determine which sections of your code can access the specific variables and which sections can't. You'll learn about these keywords in later sections of this chapter. In the meantime, bear in mind that all variables declared with the `Dim` statement exist in the module in which they were declared. If the variable *`Count`* is declared in a subroutine (an event handler, for example), it exists only in that subroutine. You can't access it from outside the subroutine. Actually, you can have a *`Count`* variable in multiple procedures. Each variable is stored locally, and they don't interfere with one another.

Variable-Naming Conventions

When declaring variables, you should be aware of a few naming conventions. A variable's name

- Must begin with a letter, followed by more letters or digits.
- Can't contain embedded periods or other special punctuation symbols. The only special character that can appear in a variable's name is the underscore character.
- Mustn't exceed 255 characters.

- Must be unique within its scope. This means that you can't have two identically named variables in the same subroutine, but you can have a variable named `counter` in many different subroutines.

Variable names in VB 2008 are case-insensitive: *`myAge`*, *`myage`*, and *`MYAGE`* all refer to the same variable in your code. Actually, as you enter variable names, the editor converts their casing so that they match their declaration.

Variable Initialization

VB 2008 allows you to initialize variables in the same line that declares them. The following statement declares an Integer variable and initializes it to 3,045:

```
Dim distance As Integer = 3045
```

This statement is equivalent to the following two:

```
Dim distance As Integer
distance = 3045
```

It is also possible to declare and initialize multiple variables, of the same or different type, on the same line:

```
Dim quantity As Integer = 1, discount As Single = 0.25
```

Type Inference

As I mentioned earlier, one of the trademark features of BASIC, including earlier versions of Visual Basic, was the ability to use variables without declaring them. It has never been a recommended practice, yet VB developers loved it. This feature is coming back to the language, only in a safer manner. VB 2008 allows you to declare variables by assigning values to them. The compiler will infer the type of the variable from its value and will create a variable of the specific type behind the scenes. The following statement creates an Integer variable:

```
Dim count = 2999
```

To request the variable's type, use the `GetType` method. This method returns a Type object, which represents the variable's type. The name of the type is given by the `ToString` property. The following statement will print the highlighted string in the Immediate window:

```
Debug.WriteLine(count.GetType.ToString)
System.Int32
```

The *`count`* variable is of the Integer type. If you attempt to assign a value of a different type to this variable later in your code, such as a date, the editor will underline the value and generate the warning *Value of type 'Date' cannot be converted to Integer*. The compiler has inferred the type of the value assigned initially to the variable and created a variable of the same type. That's why subsequent statements can't change the variable's type. You can turn off type inference by inserting the following statement at the top of the module:

```
Option Infer Off
```

Alternatively, you can turn on or off this option in the project's Properties pages. If the Infer option is off, the compiler will handle variables declared without a specific type depending on the Strict option. If the Strict option is off, the compiler will create an Object variable, which can store any value, even values of different types in the course of the application. If the Strict option is on, the compiler will reject the declaration; it will underline the variable's name with a wiggly line and generate the following warning: *Option Strict On requires all variable declarations to have an As clause*.

For more information on the various variable declaration–related options of the compiler, see the section ''The Strict, Explicit, and Infer Options,'' later in this chapter. In the following sections, you'll explore the various data types of Visual Basic, and I will use explicit declarations, which is the recommended best practice for creating and using variables in your code.

Types of Variables

Visual Basic recognizes the following five categories of variables:

- Numeric
- String
- Boolean
- Date
- Object

The two major variable categories are numeric and string. *Numeric variables* store numbers, and *string variables* store text. *Object variables* can store any type of data. Why bother to specify the type if one type suits all? On the surface, using object variables might seem like a good idea, but they have their disadvantages. Integer variables are optimized for storing integers, and date variables are optimized for storing dates. Before VB can use an object variable, it must determine its type and perform the necessary conversions. If the variable is declared with a specific type, these conversions are not necessary.

We begin our discussion of variable types with numeric variables. Text is stored in string variables, but numbers can be stored in many formats, depending on the size of the number and its precision. That's why there are many types of numeric variables. The String and Date data types are much richer in terms of the functionality they expose, and are discussed in more detail in Chapter 13, ''Handling Strings, Characters, and Dates.''

Numeric Variables

You'd expect that programming languages would use the same data type for numbers. After all, a number is a number. But this couldn't be further from the truth. All programming languages provide a variety of numeric data types, including the following:

- Integers (there are several integer data types)
- Decimals
- Single, or floating-point numbers with limited precision
- Double, or floating-point numbers with extreme precision

Decimal, Single, and Double are the three basic data types for storing floating-point numbers (numbers with a fractional part). The Double data type can represent these numbers more accurately than the Single type and is used almost exclusively in scientific calculations.

The Integer data types store whole numbers. The data type of your variable can make a difference in the results of the calculations. The proper variable types are determined by the nature of the values they represent, and the choice of data type is frequently a trade-off between precision and speed of execution (less-precise data types are manipulated faster). Visual Basic supports the numeric data types shown in Table 2.1. In the Data Type column, I show the name of each data type and the corresponding keyword in parentheses.

Integer Variables

There are three types of variables for storing integers, and they differ only in the range of numbers each can represent. As you understand, the more bytes a type takes, the larger values it can hold. The type of Integer variable you'll use depends on the task at hand. You should choose the type that can represent the largest values you anticipate will come up in your calculations. You can go for the Long type, to be safe, but Long variables are four times as large as Short variables, and it takes the computer longer to process them.

The statements in Listing 2.1 will help you understand when to use the various Integer data types. Each numeric data type exposes the `MinValue` and `MaxValue` properties, which return the minimum and maximum values, respectively, that can be represented by the corresponding data type. Values of the Short (`Int16`) type can be stored in Integer (`Int32`) and Long (`Int64`) variables, but the reverse is not true. If you attempt to store a Long value to an Integer variable, an error will be generated and the compiler will underline the offending line with a wiggly line. I have included comments after each statement to explain the errors produced by some of the statements.

LISTING 2.1: Experimenting with the Ranges of Numeric Variables

```
Dim shortInt As Int16
Dim Int As Int32
Dim longInt As Int64
Debug.WriteLine(Int16.MinValue)
Debug.WriteLine(Int16.MaxValue)
Debug.WriteLine(Int32.MinValue)
Debug.WriteLine(Int32.MaxValue)
Debug.WriteLine(Int64.MinValue)
Debug.WriteLine(Int64.MaxValue)
shortInt = Int16.MaxValue + 1
'   ERROR, exceeds the maximum value of the Short data type
Int = Int16.MaxValue + 1
'   OK, is within the range of the Integer data type
Int = Int32.MaxValue + 1
'   ERROR, exceeds the maximum value of the Integer data type
Int = Int32.MinValue - 1
'   ERROR, exceeds the minimum value of the Integer data type
longInt = Int32.MaxValue + 1
'   OK, is within the range of the Long data type
longInt = Int64.MaxValue + 1
'   ERROR, exceeds the range of all Integer data types
```

TABLE 2.1: Visual Basic Numeric Data Types

DATA TYPE	**MEMORY REPRESENTATION**	**STORES**
Byte (`Byte`)	1 byte	Integers in the range 0 to 255.
Signed Byte (`SByte`)	1 byte	Integers in the range −128 to 127.
Short (`Int16`)	2 bytes	Integer values in the range −32,768 to 32,767.
Integer (`Int32`)	4 bytes	Integer values in the range −2,147,483,648 to 2,147,483,647.
Long (`Int64`)	8 bytes	Integer values in the range −9,223,372,036,854,755,808 to 9,223,372,036,854,755,807.
Unsigned Short (`UShort`)	2 bytes	Positive integer values in the range 0 to 65,535.
Unsigned Integer (`UInteger`)	4 bytes	Positive integers in the range 0 to 4,294,967,295.
Unsigned Long (`ULong`)	8 bytes	Positive integers in the range 0 to 18,446,744,073,709,551,615.
Single Precision (`Single`)	4 bytes	Single-precision floating-point numbers. It can represent negative numbers in the range −3.402823E38 to −1.401298E-45 and positive numbers in the range 1.401298E-45 to 3.402823E38. The value 0 can't be represented precisely (it's a very, very small number, but not exactly 0).
Double Precision (`Double`)	8 bytes	Double-precision floating-point numbers. It can represent negative numbers in the range −1.79769313486232E308 to −4.94065645841247E-324 and positive numbers in the range 4.94065645841247E-324 to 1.79769313486232E308.
Decimal (`Decimal`)	16 bytes	Integer and floating-point numbers scaled by a factor in the range from 0 to 28. See the description of the Decimal data type for the range of values you can store in it.

The six `WriteLine` statements will print the minimum and maximum values you can represent with the various Integer data types. The following statement attempts to assign to a Short integer variable a value that exceeds the largest possible value you can represent with the Short data type, and it will generate an error. The editor will underline the incorrect statement, and if you hover the pointer over the statement, you'll see the error description: *Constant expression not representable in type Short*. If you attempt to store the same value to an Integer variable, there will be no problem because this value is well within the range of the Integer data type.

The next two statements attempt to store to an Integer variable two values that are also outside of the range that an integer can represent. The first value exceeds the range of positive values, and the second exceeds the range of negative values. If you attempt to store these values to a Long variable, there will be no problem. If you exceed the range of values that can be represented by the Long data type, you're out of luck. This value can't be represented as an integer, and you must store it in one of the variable types discussed in the next sections.

Single- and Double-Precision Numbers

The names *Single* and *Double* come from single-precision and double-precision numbers. Double-precision numbers are stored internally with greater accuracy than single-precision numbers. In scientific calculations, you need all the precision you can get; in those cases, you should use the Double data type.

The result of the operation 1 / 3 is 0.333333. . . (an infinite number of digits *3*). You could fill 256 MB of RAM with *3* digits, and the result would still be truncated. Here's a simple example that demonstrates the effects of truncation:

In a button's `Click` event handler, declare two variables as follows:

```
Dim a As Single, b As Double
```

Then enter the following statements:

```
a = 1 / 3
Debug.WriteLine(a)
```

Run the application, and you should get the following result in the Output window:

```
.3333333
```

There are seven digits to the right of the decimal point. Break the application by pressing Ctrl+Break and append the following lines to the end of the previous code segment:

```
a = a * 100000
Debug.WriteLine(a)
```

This time, the following value will be printed in the Output window:

```
33333.34
```

The result is not as accurate as you might have expected initially — it isn't even rounded properly. If you divide *a* by 100,000, the result will be

```
0.3333334
```

This number is different from the number we started with (0.3333333). The initial value was rounded when we multiplied it by 100,000 and stored it in a Single variable. This is an important point in numeric calculations, and it's called *error propagation*. In long sequences of numeric calculations, errors propagate. Even if you can tolerate the error introduced by the Single data type in a single operation, the cumulative errors might be significant.

Let's perform the same operations with double-precision numbers, this time using the variable *b*. Add these lines to the button's `Click` event handler:

```
b = 1 / 3
Debug.WriteLine(b)
b = b * 100000
Debug.WriteLine(b)
```

This time, the following numbers are displayed in the Output window:

```
0.333333333333333
33333.3333333333
```

The results produced by the double-precision variables are more accurate.

Why are such errors introduced in our calculations? The reason is that computers store numbers internally with two digits: zero and one. This is very convenient for computers because electronics understand two states: on and off. As a matter of fact, all the statements are translated into bits (zeros and ones) before the computer can understand and execute them. The binary numbering system used by computers is not much different from the decimal system we humans use; computers just use fewer digits. We humans use 10 different digits to represent any number, whole or fractional, because we have 10 fingers (in effect, computers count with just two fingers). Just as with the decimal numbering system, in which some numbers can't be precisely represented, there are also numbers that can't be represented precisely in the binary system.

Let me give you a more illuminating example. Create a single-precision variable, *a*, and a double-precision variable, *b*, and assign the same value to them:

```
Dim a As Single, b As Double
a = 0.03007
b = 0.03007
```

Then print their difference:

```
Debug.WriteLine(a-b)
```

If you execute these lines, the result won't be zero! It will be −6.03199004634014E-10. This is a very small number that can also be written as 0.000000000603199004634014. Because different numeric types are stored differently in memory, they don't quite match. What this means to you is that all variables in a calculation should be of the same type.

Eventually, computers will understand mathematical notation and will not convert all numeric expressions into values, as they do today. If you multiply the expression 1/3 by 3, the result should be 1. Computers, however, must convert the expression 1/3 into a value before they can multiply it by 3. Because 1/3 can't be represented precisely, the result of the (1/3) × 3 will not be exactly 1. If the variables *a* and *b* are declared as Single or Double, the following statements will print 1:

```
a = 3
b = 1 / a
Debug.WriteLine(a * b)
```

If the two variables are declared as Decimal, however, the result will be a number very close to 1, but not exactly 1 (it will be 0.9999999999999999999999999999 — there are 28 digits after the decimal point).

The Decimal Data Type

Variables of the Decimal type are stored internally as integers in 16 bytes and are scaled by a power of 10. The scaling power determines the number of decimal digits to the right of the floating point, and it's an integer value from 0 to 28. When the scaling power is 0, the value is multiplied by 10^0, or 1, and it's represented without decimal digits. When the scaling power is 28, the value is divided by 10^{28} (which is 1 followed by 28 zeros — an enormous value), and it's represented with 28 decimal digits.

The largest possible value you can represent with a Decimal value is an integer: 79,228,162, 514,264,337,593,543,950,335. The smallest number you can represent with a Decimal variable is the negative of the same value. These values use a scaling factor of 0. When the scaling factor is 28, the largest value you can represent with a Decimal variable is quite small, actually. It's 7.9228162514264337593543950335 (and the smallest value is the same with a minus sign). This is a very small numeric value (not quite 8), but it's represented with extreme accuracy. The number zero can't be represented precisely with a Decimal variable scaled by a factor of 28. The smallest positive value you can represent with the same scaling factor is 0.00. . .01 (there are 27 zeros between the decimal period and the digit 1) — an extremely small value, but still not quite zero. The more accuracy you want to achieve with a Decimal variable, the smaller the range of available values you have at your disposal — just as with everything else in life.

When using decimal numbers, the compiler keeps track of the decimal digits (the digits following the decimal point) and treats all values as integers. The value 235.85 is represented as the integer 23585, but the compiler knows that it must scale down the value by 100 when it finishes using it. Scaling down by 100 (that is, 10^2) corresponds to shifting the decimal point by two places. First, the compiler multiplies this value by 100 to make it an integer. Then, it divides it by 100 to restore the original value. Let's say that you want to multiply the following values:

```
328.558 * 12.4051
```

First, you must turn them into integers. You must remember that the first number has three decimal digits, and the second number has four decimal digits. The result of the multiplication will have seven decimal digits. So you can multiply the following integer values:

```
328558 * 124051
```

and then treat the last seven digits of the result as decimals. Use the Windows Calculator (in the Scientific view) to calculate the previous product. The result is 40,757,948,458. The actual value after taking into consideration the decimal digits is 4,075.7948458. This is how the compiler manipulates the Decimal data type. Insert the following lines in a button's `Click` event handler and execute the program:

```
Dim a As Decimal = 328.558D
Dim b As Decimal = 12.4051D
Dim c As Decimal
c = a * b
Debug.WriteLine(c.ToString)
```

The *D* character at the end of the two numeric values specifies that the numbers should be converted into Decimal values. By default, every value with a fractional part is treated as a Double value. Assigning a Double value to a Decimal variable will produce an error if the Strict option is on, so we must specify explicitly that the two values should be converted to the Decimal type. The *D* character at the end of the value is called a *type character*. Table 2.2 lists all of them.

TABLE 2.2: Type Characters

TYPE CHARACTER	**DESCRIPTION**	Example
C	Converts value to a Char type	`Dim ch As String = ''A''c`
D or @	Converts value to a Decimal type	`Dim price As Decimal = 12.99D`
R or #	Converts value to a Double type	`Dim pi As Double = 3.14 R`
I or %	Converts value to an Integer type	`Dim count As Integer = 99I`
L or &	Converts value to a Long type	`Dim distance As Long = 1999L`
S	Converts value to a Short type	`Dim age As Short = 1 S`
F or !	Converts value to a Single type	`Dim velocity As Single = 74.99F`

If you perform the same calculations with Single variables, the result will be truncated (and rounded) to three decimal digits: 4,075.795. Notice that the Decimal data type didn't introduce any rounding errors. It's capable of representing the result with the exact number of decimal digits. This is the real advantage of Decimals, which makes them ideal for financial applications. For scientific calculations, you must still use Doubles. Decimal numbers are the best choice for calculations that require a specific precision (such as four or eight decimal digits).

Infinity and Other Oddities

The Framework can represent two very special values, which may not be numeric values themselves but are produced by numeric calculations: NaN (not a number) and Infinity. If your calculations produce NaN or Infinity, you should confirm the data and repeat the calculations, or give up. For all practical purposes, neither NaN nor Infinity can be used in everyday business calculations.

Not a Number (NaN)

NaN is not new. Packages such as Wolfram Mathematica and Microsoft Excel have been using it for years. The value NaN indicates that the result of an operation can't be defined: It's not a regular number, not zero, and not infinity. NaN is more of a mathematical concept rather than a value you can use in your calculations. The `Log()` function, for example, calculates the logarithm of positive values. By definition, you can't calculate the logarithm of a negative value. If the argument you pass to the `Log()` function is a negative value, the function will return the value NaN to indicate that the calculations produced an invalid result. You may find it annoying that a numeric function returns a non-numeric value, but it's better than throwing an exception. Even if you don't detect this condition immediately, your calculations will continue and they will all produce NaN values.

Some calculations produce undefined results, such as infinity. Mathematically, the result of dividing any number by zero is infinity. Unfortunately, computers can't represent infinity, so they produce an error when you request a division by zero. VB 2008 will report a special value, which isn't a number: the Infinity value. If you call the `ToString` method of this value, however, it will return the string `Infinity`. Let's generate an `Infinity` value. Start by declaring a Double variable, *`dblVar`*:

```
Dim dblVar As Double = 999
```

Then divide this value by zero:

```
Dim infVar as Double
infVar = dblVar / 0
```

and display the variable's value:

```
MsgBox(infVar)
```

The string `Infinity` will appear in a message box. This string is just a description; it tells you that the result is not a valid number (it's a very large number that exceeds the range of numeric values that can be represented with any data type), but it *shouldn't* be used in other calculations. However, you *can* use the Infinity value in arithmetic operations. Certain operations with infinity make sense; others don't. If you add a number to infinity, the result is still infinity (any number, even an arbitrarily large one, can still be increased). If you divide a value by infinity, you'll get the zero value, which also makes sense. If you divide one Infinity value by another Infinity value, you'll get the second odd value, NaN.

Another calculation that will yield a non-number is the division of a very large number by a very small number. If the result exceeds the largest value that can be represented with the Double data type, the result is `Infinity`. Declare three variables as follows:

```
Dim largeVar As Double = 1E299
Dim smallVar As Double = 1E-299
Dim result As Double
```

The notation `1E299` means 10 raised to the power of 299, which is an extremely large number. Likewise, `1E-299` means 10 raised to the power of −299, which is equivalent to dividing 10 by a number as large as `1E299`.

Then divide the large variable by the small variable and display the result:

```
result = largeVar / smallVar
MsgBox(result)
```

The result will be Infinity. If you reverse the operands (that is, you divide the very small by the very large variable), the result will be zero. It's not exactly zero, but the Double data type can't accurately represent numeric values that are very, very close to zero.

You can also produce an Infinity value by multiplying a very large (or very small) number by itself many times. But clearly, the most absurd method of generating an Infinity value is to assign the Double.PositiveInfinity or Double.NegativeInfinity value to a variable!

The result of the division 0 / 0, for example, is not a numeric value. If you attempt to enter the statement `0 / 0` in your code, however, VB will catch it even as you type, and you'll get the error message *Division by zero occurs in evaluating this expression*.

To divide zero by zero, set up two variables as follows:

```
Dim var1, var2 As Double
Dim result As Double
var1 = 0
var2 = 0
result = var1 / var2
MsgBox(result)
```

If you execute these statements, the result will be `NaN`. Any calculations that involve the *`result`* variable will yield `NaN` as a result. The following statements will produce a NaN value:

```
result = result + result
result = 10 / result
result = result + 1E299
MsgBox(result)
```

If you make *`var2`* a very small number, such as 1E-299, the result will be zero. If you make *`var1`* a very small number, the result will be Infinity.

For most practical purposes, Infinity is handled just like NaN. They're both numbers that shouldn't occur in business applications (unless you're projecting the national deficit in the next 50 years), and when they do, it means that you must double-check your code or your data. They are much more likely to surface in scientific calculations, and they must be handled with the statements described in the next section.

Testing for Infinity and NaN

To find out whether the result of an operation is a NaN or Infinity, use the `IsNaN` and `IsInfinity` methods of the Single and Double data types. The Integer data type doesn't support these methods, even if it's possible to generate Infinity and NaN results with integers. If the `IsInfinity` method returns True, you can further examine the sign of the Infinity value with the `IsNegativeInfinity` and `IsPositiveInfinity` methods.

In most situations, you'll display a warning and terminate the calculations. The statements of Listing 2.2 do just that. Place these statements in a button's `Click` event handler and run the application.

Listing 2.2: Handling NaN and Infinity Values

```
Dim var1, var2 As Double
Dim result As Double
var1 = 0
var2 = 0
result = var1 / var2
If Double.IsInfinity(result) Then
    If Double.IsPositiveInfinity(result) Then
        MsgBox("Encountered a very large number. Can't continue")
    Else
        MsgBox("Encountered a very small number. Can't continue")
    End If
Else
    If Double.IsNaN(result) Then
        MsgBox("Unexpected error in calculations")
    Else
        MsgBox("The result is : " & result.ToString)
    End If
End If
```

This listing will generate a NaN value. Set the value of the `var1` variable to 1 to generate a positive Infinity value, or to −1 to generate a negative Infinity value. As you can see, the `IsInfinity`, `IsPositiveInfinity`, `IsNegativeInfinity`, and `IsNaN` methods require that the variable be passed as an argument.

If you change the values of the `var1` and `var2` variables to the following values and execute the application, you'll get the message *Encountered a very large number*:

```
var1 = 1E+299
var2 = 1E-299
```

If you reverse the values, you'll get the message *Encountered a very small number*. In any case, the program will terminate gracefully and let you know the type of problem that prevents the completion of the calculations.

Byte Variables

None of the previous numeric types is stored in a single byte. In some situations, however, data are stored as bytes, and you must be able to access individual bytes. The Byte data type holds an integer in the range of 0 to 255. Bytes are frequently used to access binary files, image and sound files, and so on. Note that you no longer use bytes to access individual characters. Unicode characters are stored in two bytes.

To declare a variable as a Byte, use the following statement:

```
Dim n As Byte
```

The variable `n` can be used in numeric calculations too, but you must be careful not to assign the result to another Byte variable if its value might exceed the range of the Byte type. If the variables `A` and `B` are initialized as follows:

```
Dim A As Byte, B As Byte
A = 233
B = 50
```

the following statement will produce an overflow exception:

```
Debug.WriteLine(A + B)
```

The same will happen if you attempt to assign this value to a Byte variable with the following statement:

```
B = A + B
```

The result (283) can't be stored in a single byte. Visual Basic generates the correct answer, but it can't store it into a Byte variable.

Boolean Operations with Bytes

The operators that won't cause overflows are the Boolean operators `AND`, `OR`, `NOT`, and `XOR`, which are frequently used with Byte variables. These aren't logical operators that return True or False; they combine the matching bits in the two operands and return another byte. If you combine the numbers 199 and 200 with the `AND` operator, the result is 192. The two values in binary format are 11000111 and 11001000. If you perform a bitwise AND operation on these two values, the result is 11000000, which is the decimal value 192.

In addition to the Byte data type, VB 2008 provides a Signed Byte data type, `SByte`, which can represent signed values in the range from −128 to 127. The bytes starting with the *1* bit represent negative values. The range of positive values is less by one than the range of values of negative values, because the value 0 is considered a positive value (its first bit is *0*).

Boolean Variables

The Boolean data type stores True/False values. Boolean variables are, in essence, integers that take the value −1 (for True) and 0 (for False). Actually, any nonzero value is considered True. Boolean variables are declared as

```
Dim failure As Boolean
```

and they are initialized to False. Boolean variables are used in testing conditions, such as the following:

```
Dim failure As Boolean = False
' other statements ...
If failure Then MsgBox("Couldn't complete the operation")
```

They are also combined with the logical operators `And`, `Or`, `Not`, and `Xor`. The `Not` operator toggles the value of a Boolean variable. The following statement is a toggle:

```
running = Not running
```

If the variable `running` is True, it's reset to False, and vice versa. This statement is a shorter way of coding the following:

```
Dim running As Boolean
If running = True Then
   running = False
Else
   running = True
End If
```

Boolean operators operate on Boolean variables and return another Boolean as their result. The following statements will display a message if one (or both) of the variables `ReadOnly` and `Hidden` are True (presumably these variables represent the corresponding attributes of a file):

```
If ReadOnly Or Hidden Then
   MsgBox("Couldn't open the file")
Else
   { statements to open and process file}
End If
```

The condition of the `If` statement combines the two Boolean values with the `Or` operator. If one or both of them are True, the parenthesized expression is True. This value is negated with the `Not` operator, and the `If` clause is executed only if the result of the negation is True. If `ReadOnly` is True and `Hidden` is False, the expression is evaluated as

```
If Not (True Or False)
```

`(True Or False)` is True, which reduces the expression to

```
If Not True
```

which, in turn, is False.

String Variables

The String data type stores only text, and string variables are declared as follows:

```
Dim someText As String
```

You can assign any text to the variable `someText`. You can store nearly 2GB of text in a string variable (that's 2 billion characters, and is much more text than you care to read on a computer screen). The following assignments are all valid:

```
Dim aString As String
aString = "Now is the time for all good men to come " & _
            " to the aid of their country"
aString = ""
aString = "There are approximately 25,000 words in this chapter"
aString = "25,000"
```

The second assignment creates an empty string, and the last one creates a string that just happens to contain numeric digits, which are also characters. The difference between these two variables is that they hold different values:

```
Dim aNumber As Integer = 25000
Dim aString As String = "25,000"
```

The *`aString`* variable holds the characters 2, 5, comma, 0, 0, and 0; and *`aNumber`* holds a single numeric value. However, you can use the variable *`aString`* in numeric calculations, and the variable *`aNumber`* in string operations. VB will perform the necessary conversions as long as the Strict option is off.

The String data type and its text manipulation methods are discussed in detail in Chapter 13.

Character Variables

Character variables store a single Unicode character in two bytes. In effect, characters are Unsigned Short integers (`UInt16`); you can use the `CChar()` function to convert integers to characters and use the `CInt()` function to convert characters to their equivalent integer values.

To declare a Character variable, use the `Char` keyword:

```
Dim char1, char2 As Char
```

You can initialize a Character variable by assigning either a character or a string to it. In the latter case, only the first character of the string is assigned to the variable. The following statements will print the characters *a* and *A* to the Output window:

```
Dim char1 As Char = "a", char2 As Char = "ABC"
Debug.WriteLine(char1)
Debug.WriteLine(char2)
```

These statements will work only if the Strict option is off. If it's on, the values assigned to the `char1` and `char2` variables will be marked in error. To fix the error that prevents the compilation of the code, change the `Dim` statement as follows:

```
Dim char1 As Char = "a"c, char2 As Char = "A"c
```

When the Strict option is on, you can't assign a string to a Char variable and expect that only the first character of the string will be used.

The Integer values that correspond to the English characters are the ANSI (American National Standards Institute) codes of the equivalent characters. The following statement will print the value 65:

```
Debug.WriteLine(Convert.ToInt32("a"))
```

If you convert the Greek character alpha (α) to an integer, its value is 945. The Unicode value of the famous character π is 960.

Character variables are used in conjunction with strings. You'll rarely save real data as characters. However, you might have to process the individual characters in a string, one at a time. The Char data type exposes a number of interesting methods for manipulating characters, and they're presented in detail in Chapter 13. Let's say the string variable `password` holds a user's new password, and you require that passwords contain at least one special symbol. The code segment of Listing 2.3 scans the password and rejects it if it contains letters and digits only.

Listing 2.3: Processing Individual Characters

```
Dim password As String, ch As Char
Dim i As Integer
Dim valid As Boolean = False
While Not valid
   password = InputBox("Please enter your password")
   For i = 0 To password.Length - 1
      ch = password.Chars(i)
      If Not Char.IsLetterOrDigit(ch) Then
         valid = True
         Exit For
      End If
   Next
   If valid Then
      MsgBox("You new password will be activated immediately!")
   Else
      MsgBox("Your password must contain at least one special symbol!")
   End If
End While
```

If you are not familiar with the `If...Then`, `For...Next`, or `While...End While` structures, you can read their descriptions in the following chapter.

The code prompts the user with an input box to enter a password. The *`valid`* variable is Boolean and it's initialized to False. (You don't have to initialize a Boolean variable to False because this is its default initial value, but it does make the code easier to read.) It's set to True from within the body of the loop, only if the password contains a character that is not a letter or a digit. We set it to False initially, so the `While...End While` loop will be executed at least once. This loop will keep prompting the user until a valid password is entered.

The `For...Next` loop scans the string variable *`password`*, one letter at a time. At each iteration, the next letter is copied into the *`ch`* variable. The `Chars` property of the String data type is an array that holds the individual characters in the string (another example of the functionality built into the data types).

Then the program examines the current character. The `IsLetterOrDigit` method of the Char data type returns True if a character is either a letter or a digit. If the current character is a symbol, the program sets the *`valid`* variable to True so that the outer loop won't be executed again, and it exits the `For...Next` loop. Finally, it prints the appropriate message, and either prompts for another password or quits.

The Char class and its methods are discussed in more detail in Chapter 13.

Date Variables

Date and time values are stored internally in a special format, but you don't need to know the exact format. They are double-precision numbers: the integer part represents the date, and the fractional part represents the time. A variable declared as Date with a statement like the following can store both date and time values:

```
Dim expiration As Date
```

The following are all valid assignments:

```
expiration = #01/01/2008#
expiration = #8/27/2008 6:29:11 PM#
expiration = "July 2, 2008"
expiration = Today()
```

By the way, the `Today()` function returns the current date and time, while the `Now()` function returns the current date. You can also retrieve the current date by calling the `Today` property of the Date data type: `Date.Today`.

The pound sign tells Visual Basic to store a date value to the *`expiration`* variable, just as the quotes tell Visual Basic that the value is a string. You can store a date as a string to a Date variable, but it will be converted to the appropriate format. If the Strict option is on, you can't specify dates by using the Long date format (as in the third statement of this example).

The date format is determined by the Regional Settings (found in the Control Panel). In the United States, the format is *mm/dd/yy*. (In other countries, the format is *dd/mm/yy*.) If you assign an invalid date to a date variable, such as 23/04/2002, the statement will be underlined and an error message will appear in the Task List window. The description of the error is *Date constant is not valid*.

The Date data type is extremely flexible; Visual Basic knows how to handle date and time values, so you won't have to write complicated code to perform the necessary conversions. To manipulate dates and times, use the members of the Date type, which are discussed in detail in Chapter 13, or the date and time functions of VB 6, which are still supported by VB 2008.

You can also perform arithmetic operations with date values. VB recognizes your intention to subtract dates and it properly evaluates their difference. The result is a TimeSpan object, which represents a time interval. If you execute the following statements, the value 638.08:49:51.4970000 will appear in the Output window:

```
Dim d1, d2 As Date
d1 = Now
d2 = #1/1/2004#Debug.WriteLine(d1 - d2)
```

The value of the TimeSpan object represents an interval of 638 days, 8 hours, 49 minutes, and 51.497 seconds.

Data Type Identifiers

Finally, you can omit the `As` clause of the `Dim` statement, yet create typed variables, with the variable declaration characters, or *data type identifiers.* These characters are special symbols that you append to the variable name to denote the variable's type. To create a string variable, you can use this statement:

```
Dim myText$
```

The dollar sign signifies a string variable. Notice that the name of the variable includes the dollar sign — it's *`myText$`*, not *`myText`*. To create a variable of a particular type, use one of the data declaration characters shown in Table 2.3. (Not all data types have their own identifiers.)

Using type identifiers doesn't help to produce the cleanest and easiest-to-read code. They're relics from really old versions of BASIC, and if you haven't used them in the past, there's no really good reason to start using them now.

The Strict, Explicit, and Infer Options

The Visual Basic compiler provides three options that determine how it handles variables:

- The *Explicit option* indicates whether you will declare all variables.
- The *Strict option* indicates whether all variables will be of a specific type.
- The *Infer option* indicates whether the compiler should determine the type of a variable from its value.

These options have a profound effect on the way you declare and use variables, and you should understand what they do. By exploring these settings, you will also understand a little better how

TABLE 2.3: Data Type Definition Characters

SYMBOL	DATA TYPE	EXAMPLE
`$`	String	`A$`, `messageText$`
`%`	Integer (`Int32`)	`counter%`, `var%`
`&`	Long (`Int64`)	`population&`, `colorValue&`
`!`	Single	`distance!`
`#`	Double	`ExactDistance`
`@`	Decimal	`Balance@`

the compiler handles variables. It's recommended that you turn on all three of them, but old VB developers may not follow this advice.

VB 2008 doesn't *require* that you declare your variables, but the default behavior is to throw an exception if you attempt to use a variable that hasn't been previously declared. If an undeclared variable's name appears in your code, the editor will underline the variable's name with a wiggly line, indicating that it caught an error. The description of the error will appear in the Task List below the code window. If you rest the pointer over the segment of the statement in question, you will see the description of the error in a ToolTip box.

To change the default behavior, you must insert the following statement at the beginning of the file:

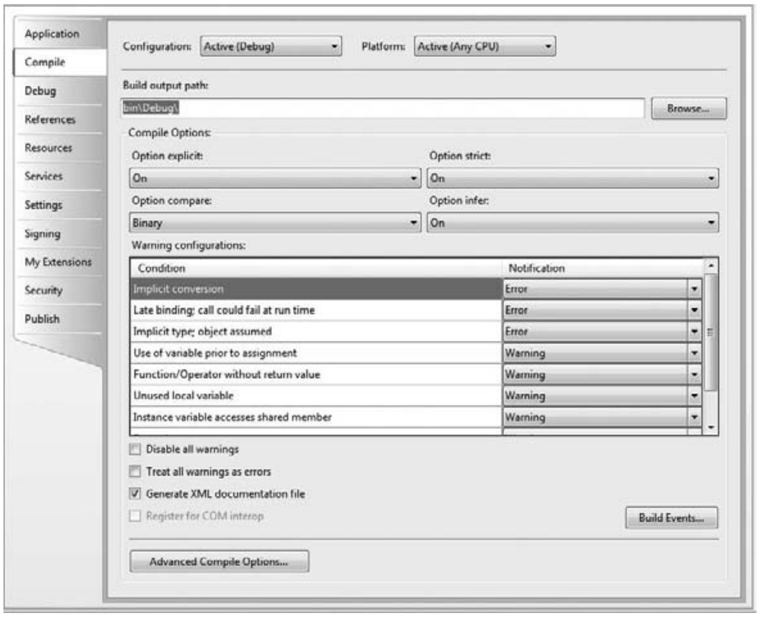
```
Option Explicit Off
```

The `Option Explicit` statement must appear at the very beginning of the file. This setting affects the code in the current module, not in all files of your project or solution. You can turn on the Strict (as well as the Explicit) option for an entire solution. Open the solution's properties dialog box (right-click the solution's name in Solution Explorer and select Properties), select the Compile tab, and set the Strict and Explicit options accordingly, as shown in Figure 2.1.

You can also set default values for the Explicit option (as well as for Strict and Infer) for all projects through the Options dialog box of the IDE. To open this dialog box, choose the Options command from the Tools menu. When the dialog box appears, select the VB Defaults tab under Projects And Solutions, as shown in Figure 2.2. Here you can set the default values for all four options. You can still change the default values for specific projects through the project's Properties pages.

The way undeclared variables are handled by VB 2008 is determined by the Explicit and Strict options, which can be either on or off. The Explicit option requires that all variables used in the code are declared before they're used. The Strict option requires that variables are declared with a specific type. In other words, the Strict option disallows the use of generic variables that can store any data type.

The default value of the Explicit statement is On. This is also the recommended value, and you should not make a habit of changing this setting. In the section ''Why Declare Variables?'' later in this chapter, you will see an example of the pitfalls you'll avoid by declaring your variables. By setting the Explicit option to Off, you're telling VB that you intend to use variables without declaring them. As a consequence, VB can't make any assumption about the variable's type, so it uses a generic type of variable that can hold any type of information. These variables are called Object variables, and they're equivalent to the old variants.



Configuration: Active (Debug) Platform: Active (Any CPU)

Build output path: bin\Debug\

Compile Options:

Option explicit: On

Option strict: On

Option compare: Binary

Option infer: On

Warning configurations:

Condition	Notification
Implicit conversion	Error
Late binding; call could fail at run time	Error
Implicit type; object assumed	Error
Use of variable prior to assignment	Warning
Function/Operator without return value	Warning
Unused local variable	Warning
Instance variable accesses shared member	Warning

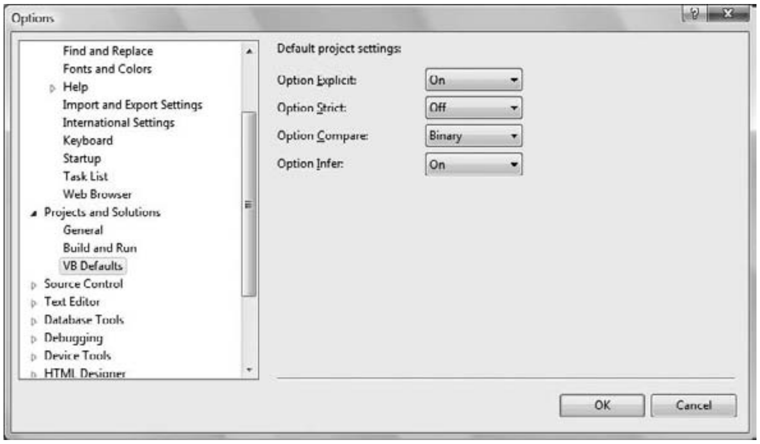
☐ Disable all warnings

☐ Treat all warnings as errors

☑ Generate XML documentation file

☐ Register for COM interop

FIGURE 2.1
Setting the variable-related options on the project's Properties pages



Default project settings:

Option Explicit: On

Option Strict: Off

Option Compare: Binary

Option Infer: On

FIGURE 2.2
Setting the variable-related options in the Visual Studio Options dialog box

While the option Explicit is set to Off, every time Visual Basic runs into an undeclared variable name, it creates a new variable on the spot and uses it. The new variable's type is Object, the generic data type that can accommodate all other data types. Using a new variable in your code is equivalent to declaring it without type. Visual Basic adjusts its type according to the value you assign to it. Create two variables, *var1* and *var2*, by referencing them in your code with statements like the following ones:

```
var1 = "Thank you for using Fabulous Software"
var2 = 49.99
```

The *var1* variable is a string variable, and *var2* is a numeric one. You can verify this with the GetType method, which returns a variable's type. The following statements print the highlighted types shown below each statement:

```
Debug.WriteLine "Variable var1 is " & var1.GetType().ToString
Variable var1 is System.String
Debug.WriteLine "Variable var2 is " & var2.GetType().ToString
Variable var2 is System.Double
```

Later in the same program, you can reverse the assignments:

```
var1 = 49.99
var2 = "Thank you for using Fabulous Software"
```

If you execute the preceding statements again, you'll see that the types of the variables have changed. The *var1* variable is now a Double, and *var2* is a String. The type of a generic variable is determined by the variable's contents and it can change in the course of the application. Of course, changing a variable's type at runtime doesn't come without a performance penalty (a small one, but nevertheless some additional statements must be executed).

Another related option is the Strict option, which is off by default. The Strict option tells the compiler whether the variables should be *strictly typed*. A strictly typed variable must be declared with a specific type and it can accept values of the same type only. With the Strict option set to Off, you can use a string variable that holds a number in a numeric calculation:

```
Dim a As String = "25000"
Debug.WriteLine a / 2
```

The last statement will print the value 12500 in the Immediate window. Likewise, you can use numeric variables in string calculations:

```
Dim a As Double = 31.03
a = a + "1"
```

If you turn the Strict option on by inserting the following statement at the beginning of the file, you won't be able to mix and match variable types:

```
Option Strict On
```

If you attempt to execute any of the last two code segments while the Strict option is on, the compiler will underline a segment of the statement to indicate an error. If you rest the pointer over the underlined segment of the code, the following error message will appear in a tip box:

```
Option strict disallows implicit conversions from String to Double
```

(or whatever type of conversion is implied by the statement).

When the Strict option is set to On, the compiler doesn't disallow all implicit conversions between data types. For example, it will allow you to assign the value of an integer to a Long, but not the opposite. The Long value might exceed the range of values that can be represented by an Integer variable. You will find more information on implicit conversions in the section titled ''Widening and Narrowing Conversions,'' later in this chapter.

Object Variables

Variants — variables without a fixed data type — were the bread and butter of VB programmers up to version 6. Variants are the opposite of strictly typed variables: They can store all types of values, from a single character to an object. If you're starting with VB 2008, you should use strictly typed variables. However, variants are a major part of the history of VB, and most applications out there (the ones you may be called to maintain) use them. I will discuss variants briefly in this section and show you what was so good (and bad) about them.

Variants, or object variables, were the most flexible data types because they could accommodate all other types. A variable declared as Object (or a variable that hasn't been declared at all) is handled by Visual Basic according to the variable's current contents. If you assign an integer value to an object variable, Visual Basic treats it as an integer. If you assign a string to an object variable, Visual Basic treats it as a string. Variants can also hold different data types in the course of the same program. Visual Basic performs the necessary conversions for you.

To declare a variant, you can turn off the Strict option and use the `Dim` statement without specifying a type, as follows:

```
Dim myVar
```

If you don't want to turn off the Strict option (which isn't recommended, anyway), you can declare the variable with the Object data type:

```
Dim myVar As Object
```

Every time your code references a new variable, Visual Basic will create an object variable. For example, if the variable *`validKey`* hasn't been declared, when Visual Basic runs into the following line, it will create a new object variable and assign the value `002-6abbgd` to it:

```
validKey = "002-6abbgd"
```

You can use object variables in both numeric and string calculations. Suppose that the variable *`modemSpeed`* has been declared as Object with one of the following statements:

```
Dim modemSpeed            ' with Option Strict = Off
Dim modemSpeed As Object  ' with Option Strict = On
```

and later in your code you assign the following value to it:

```
modemSpeed = "28.8"
```

The *modemSpeed* variable is a string variable that you can use in statements such as the following:

```
MsgBox "We suggest a " & modemSpeed & " modem."
```

This statement displays the following message:

```
"We suggest a 28.8 modem."
```

You can also treat the *modemSpeed* variable as a numeric value with the following statement:

```
Debug.WriteLine "A " & modemSpeed & " modem can transfer " & _
             modemSpeed * 1024 / 8 & " bytes per second."
```

This statement displays the following message:

```
"A 28.8 modem can transfer 3686.4 bytes per second."
```

The first instance of the *modemSpeed* variable in the preceding statement is treated as a string because this is the variant's type according to the assignment statement (we assigned a string to it). The second instance, however, is treated as a number (a single-precision number). Visual Basic converts it to a numeric value because it's used in a numeric calculation.

Another example of this behavior of variants can be seen in the following statements:

```
Dim I As Integer, S As String
I = 10
S = "11"
Debug.WriteLine(I + S)
Debug.WriteLine(I & S)
```

The first `WriteLine` statement will display the numeric value 21, whereas the second statement will print the string 1011. The plus operator (+) tells VB to add two values. In doing so, VB must convert the two strings into numeric values and then add them. The concatenation operator (&) tells VB to concatenate the two strings.

Visual Basic knows how to handle object variables in a way that makes sense. The result may not be what you had in mind, but it certainly is dictated by common sense. If you really want to concatenate the strings 10 and 11, you should use the & operator, which would tell Visual Basic exactly what to do. Quite impressive, but for many programmers, this is a strange behavior that can lead to subtle errors — and they avoid it. It's up to you to decide whether to use variants and how far you will go with them. Sure, you can perform tricks with variants, but you shouldn't overuse them to the point that others can't read your code.

Variables as Objects

Variables in VB 2008 are more than just names or placeholders for values. They're intelligent entities that can not only store but also process their values. I don't mean to scare you, but I think you should be told: VB 2008 variables are objects. And here's why: A variable that holds dates is declared as such with the following statement:

```
Dim expiration As Date
```

Then you can assign a date value to the *expiration* variable with a statement like this:

```
expiration = #1/1/2003#
```

So far, nothing out of the ordinary; this is how you use variables with any other language. In addition to holding a date, however, the *expiration* variable can manipulate dates. The following expression will return a new date that's three years ahead of the date stored in the expiration variable:

```
expiration.AddYears(3)
```

The new date can be assigned to another date variable:

```
Dim newExpiration As Date
newExpiration = expiration.AddYears(3)
```

AddYears is a method that knows how to add a number of years to a *Date* variable. There are similarly named methods for adding months, days, and so on. In addition to methods, the Date type exposes properties, such as the Month and Day properties, which return the date's month and day number, respectively. The keywords following the period after the variable's name are called *methods* and *properties*, just like the properties and methods of the controls you place on a form to create your application's visual interface. The methods and properties (or the *members*) of a variable expose the functionality that's built into the class representing the variable itself. Without this built-in functionality, you'd have to write some serious code to extract the month from a date variable, to add a number of days to a given date, to figure out whether a character is a letter, a digit, or a punctuation symbol, and so on. Much of the functionality that you'll need in an application that manipulates dates, numbers, or text has already been built into the variables themselves.

Don't let the terminology scare you. Think of variables as placeholders for values and access their functionality with expressions like the ones shown earlier. Start using variables to store values and, if you need to process them, enter a variable's name followed by a period to see a list of the members it exposes. In most cases, you'll be able to figure out what these members do by just reading their names. I'll come back to the concept of variables as objects, but I wanted to hit it right off the bat. A more detailed discussion of the notion of variables as objects can be found in Chapter 11, ''Working with Objects,'' which discusses objects in detail.

Programming languages can treat simple variables much more efficiently than objects. An integer takes two bytes in memory, and the compiler will generate very efficient code to manipulate an integer variable (add it to another numeric value, compare it to another integer, and so on). If you declare an integer variable and use it in your code as such, Visual Studio doesn't create an object to represent this value. It creates a new variable for storing integers, like good old

BASIC. After you call one of the variable's methods, the compiler emits code to create the actual object. This process is called *boxing* and it introduces a small delay, which is truly insignificant compared to the convenience of manipulating a variable through its methods.

As you've seen by now, variables are objects. This shouldn't come as a surprise, but it's an odd concept for programmers with no experience in object-oriented programming. We haven't covered objects and classes formally yet, but you have a good idea of what an object is. It's an entity that exposes some functionality by means of properties and methods. The TextBox control is an object and it exposes the `Text` property, which allows you to read or set the text on the control. Any name followed by a period and another name signifies an object. The ''other name'' is a property or method of the object.

Converting Variable Types

In many situations, you will need to convert variables from one type into another. Table 2.4 shows the methods of the Convert class that perform data-type conversions.

TABLE 2.4: The Data-Type Conversion Methods of the Convert Class

METHOD	**CONVERTS ITS ARGUMENT TO**
`ToBoolean`	Boolean
`ToByte`	Byte
`ToChar`	Unicode character
`ToDateTime`	Date
`ToDecimal`	Decimal
`ToDouble`	Double
`ToInt16`	Short Integer (2-byte integer, `Int16`)
`ToInt32`	Integer (4-byte integer, `Int32`)
`ToInt64`	Long (8-byte integer, `Int64`)
`ToSByte`	Signed Byte
`CShort`	Short (2-byte integer, `Int16`)
`ToSingle`	Single
`ToString`	String
`ToUInt16`	Unsigned Integer (2-byte integer, `Int16`)
`ToUInt32`	Unsigned Integer (4-byte integer, `Int32`)
`ToUInt64`	Unsigned Long (8-byte integer, `Int64`)

In addition to the methods of the Convert class, you can still use the data-conversion functions of VB (`CInt()` to convert a numeric value to an Integer, `CDbl()` to convert a numeric value to a Double, `CSng()` to convert a numeric value to a Single, and so on), which you can look up in the documentation. If you're writing new applications in VB 2008, use the new Convert class to convert between data types.

To convert the variable initialized as the following

```
Dim A As Integer
```

to a Double, use the `ToDouble` method of the Convert class:

```
Dim B As Double
B = Convert.ToDouble(A)
```

Suppose that you have declared two integers, as follows:

```
Dim A As Integer, B As Integer
A = 23
B = 7
```

The result of the operation `A / B` will be a Double value. The following statement

```
Debug.Write(A / B)
```

displays the value 3.28571428571429. The result is a Double value, which provides the greatest possible accuracy. If you attempt to assign the result to a variable that hasn't been declared as Double, and the Strict option is on, then VB 2008 will generate an error message. No other data type can accept this value without loss of accuracy. To store the result to a Single variable, you must convert it explicitly with a statement like the following:

```
Convert.ToSingle(A / B)
```

You can also use the `DirectCast()` function to convert a variable or expression from one type to another. The `DirectCast()` function is identical to the `CType()` function. Let's say the variable `A` has been declared as String and holds the value 34.56. The following statement converts the value of the `A` variable to a Decimal value and uses it in a calculation:

```
Dim A As String = "34.56"
Dim B As Double
B = DirectCast(A, Double) / 1.14
```

The conversion is necessary only if the Strict option is on, but it's a good practice to perform your conversions explicitly. The following section explains what might happen if your code relies on implicit conversions.

Widening and Narrowing Conversions

In some situations, VB 2008 will convert data types automatically, but not always. Let's say you have declared and initialized two variables, an Integer and a Double, with the following statements:

```
Dim count As Integer = 99
Dim pi As Double = 3.1415926535897931
```

If the Strict option is off and you assign the variable `pi` to the `count` variable, the `count` variable's new value will be 3. (The Double value was rounded to an Integer value, according to the variable's type.) Although this may be what you want, in most cases it's an oversight that will lead to incorrect results.

If the Strict option is on and you attempt to perform the same assignment, the compiler will generate an error message to the effect that you can't convert a Double to an Integer. The exact message is *Option Strict disallows implicit conversions from Double to Integer*.

When the Strict option is on, VB 2008 will perform conversions that do not result in loss of accuracy (precision) or magnitude. These conversions are called *widening conversions*. When you assign an Integer value to a Double variable, no accuracy or magnitude is lost. This is a widening conversion, because it goes from a narrower to a wider type.

On the other hand, when you assign a Double value to an Integer variable, some accuracy is lost (the decimal digits must be truncated). This is a *narrowing conversion*, because we go from a data type that can represent a wider range of values to a data type that can represent a narrower range of values.

Because you, the programmer, are in control, you might want to give up the accuracy — presumably, it's no longer needed. Table 2.5 summarizes the widening conversions that VB 2008 will perform for you automatically.

Table 2.5: VB 2008 Widening Conversions

Original Data Type	Wider Data Type
Any type	Object
Byte	Short, Integer, Long, Decimal, Single, Double
Short	Integer, Long, Decimal, Single, Double
Integer	Long, Decimal, Single, Double
Long	Decimal, Single, Double
Decimal	Single, Double
Single	Double
Double	None
Char	String

If the Strict option is on, the compiler will point out all the statements that may cause runtime errors, and you can reevaluate your choice of variable types. You can also turn on the Strict option temporarily to see the compiler's warnings, and then turn it off again.

Formatting Numbers

So far, you've seen how to use the basic data types of the CLR. All data types expose a `ToString` method, which returns the variable's value (a number or date) as a string, so that it can be used with other strings in your code. The `ToString` method formats numbers and dates in many ways and it's probably one of the most commonly needed methods. You can call the `ToString` method without any arguments, as we have done so far, to convert any value to a string. The `ToString` method, however, accepts an optional argument, which determines how the value will be formatted as a string. For example, you can format a number as currency by prefixing it with the appropriate sign (for example, the dollar symbol) and displaying it to two decimal digits, and you can display dates in many formats. Some reports require that negative amounts are enclosed in parentheses. The `ToString` method allows you to display numbers and dates in any way you wish.

Notice that `ToString` is a method, not a property. It returns a value that you can assign to a string variable or pass as arguments to a function such as `MsgBox()`, but the original value is not affected. The `ToString` method can also format a value if called with an optional argument:

```
ToString(formatString)
```

The *`formatString`* argument is a *format specifier* (a string that specifies the exact format to be applied to the variable). This argument can be a specific character that corresponds to a predetermined format (a *standard format string*, as it's called) or a string of characters that have special meaning in formatting numeric values (a *picture format string*). Use standard format strings for the most common formatting options, and use picture strings to specify unusual formatting requirements. To format the value 9959.95 as a dollar amount, you can use the following standard currency:

```
Dim Amnt As Single = 9959.95
Dim strAmnt As String
strAmnt = Amnt.ToString("C")
```

Or use the following picture numeric format string:

```
strAmnt = Amnt.ToString("$#,###.00")
```

Both statements will format the value as $9,959.95. The `"C"` argument in the first example means currency and formats the numeric value as currency. If you're using a non-U.S. version of Windows, the currency symbol will change accordingly. Use the Regional And Language Options tool in the Control Panel to temporarily change the current culture to a European one, and the amount will be formatted with the Euro sign.

The picture format string is made up of literals and characters that have special meaning in formatting. The dollar sign has no special meaning and will appear as is. The # symbol is a digit placeholder; all # symbols will be replaced by numeric digits, starting from the right. If the number has fewer digits than specified in the string, the extra symbols to the left will be ignored.

The comma tells the Format function to insert a comma between thousands. The period is the decimal point, which is followed by two more digit placeholders. Unlike the # sign, the 0 is a special placeholder: If there are not enough digits in the number for all the zeros you've specified, a 0 will appear in the place of the missing digits. If the original value had been 9959.9, for example, the last statement would have formatted it as $9,959.90. If you used the # placeholder instead, the string returned by the Format method would have a single decimal digit.

Standard Numeric Format Strings

The ToString method of the numeric data types recognizes the standard numeric format strings shown in Table 2.6.

Table 2.6: Standard Numeric Format Strings

Format Character	Description	Example
C or c	Currency	(12345.67).ToString("C") returns $12,345.67
D or d	Decimal	(123456789).ToString("D") returns 123456789. It works with integer values only.
E or e	Scientific format	(12345.67).ToString("E") returns 1.234567E + 004
F or f	Fixed-point format	(12345.67).ToString("F") returns 12345.67
G or g	General format	Returns a value either in fixed-point or scientific format
N or n	Number format	(12345.67).ToString("N") returns 12,345.67
P or p	Percentage	(0.12345).ToString("N") returns 12,35%
R or r	Round-trip	(1 / 3).ToString("R")returns 0.33333333333333331 (where the G specifier would return a value with fewer decimal digits: 0.333333333333333
X or x	Hexadecimal format	250.ToString("X") returns FA

The format character can be followed by an integer. If present, the integer value specifies the number of decimal places that are displayed. The default accuracy is two decimal digits.

The C format string causes the ToString method to return a string representing the number as a currency value. An integer following the C determines the number of decimal digits that are displayed. If no number is provided, two digits are shown after the decimal separator. Assuming that the variable value has been declared as Decimal and its value is 5596, then the expression value.ToString("C") will return the string $5,596.00. If the value of the variable were 5596.4499, then the expression value.ToString("C3") would return the string $5,596.450.

Notice that not all format strings apply to all data types. For example, only integer values can be converted to hexadecimal format, and the D format string works with integer values only.

There are format strings and digits for dates too, and they're discussed in Chapter 13, where I will present the Date data type and related topics in detail.

Picture Numeric Format Strings

If the format characters listed in Table 2.6 are not adequate for the control you need over the appearance of numeric values, you can provide your own picture format strings. Picture format strings contain special characters that allow you to format your values exactly as you like. Table 2.7 lists the picture formatting characters.

Table 2.7: Picture Numeric Format Strings

Format Character	Description	Effect
`0`	Display zero placeholder	Results in a nonsignificant zero if a number has fewer digits than there are zeros in the format
`#`	Display digit placeholder	Replaces the symbol with only significant digits
`.`	Decimal point	Displays a period (.) character
`,`	Group separator	Separates number groups — for example, 1,000
`%`	Percent notation	Displays a % character
`E + 0, E−0, e + 0, e−0`	Exponent notation	Formats the output of exponent notation
`\`	Literal character	Used with traditional formatting sequences like such as \n (newline)
`''''`	Literal string	Displays any string within quotes or apostrophes literally
`;`	Section separator	Specifies different output if the numeric value to be formatted is positive, negative, or zero

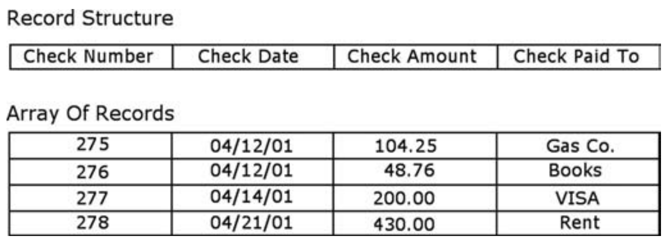
The following statements will print the highlighted values:

```
Dim Amount As Decimal = 42492.45
Debug.WriteLine(Amount.ToString("$#,###.00"))
$42,492.45
Amount = 0.2678
Debug.WriteLine(Amount.ToString("0.000"))
0.268
Amount = -24.95
Debug.WriteLine(Amount.ToString("$#,###.00;($#,###.00)"))
($24.95)
```

User-Defined Data Types

In the previous sections, we used variables to store individual values. As a matter of fact, most programs store sets of data of different types. For example, a program for balancing your checkbook must store several pieces of information for each check: the check's number, amount, date, and so on. All these pieces of information are necessary to process the checks, and ideally, they should be stored together.

You can create custom data types that are made up of multiple values using *structures*. A VB structure allows you to combine multiple values of the basic data types and handle them as a whole. For example, each check in a checkbook-balancing application is stored in a separate structure (or record), as shown in Figure 2.3. When you recall a given check, you need all the information stored in the structure.



Record Structure

Check Number	Check Date	Check Amount	Check Paid To

Array Of Records

275	04/12/01	104.25	Gas Co.
276	04/12/01	48.76	Books
277	04/14/01	200.00	VISA
278	04/21/01	430.00	Rent

FIGURE 2.3
Pictorial representation of a structure

To define a structure in VB 2008, use the `Structure` statement, which has the following syntax:

```
Structure structureName
   Dim variable1 As varType
   Dim variable2 As varType
   ...
   Dim variablen As varType
End Structure
```

`varType` can be any of the data types supported by the CLR. The `Dim` statement can be replaced by the `Private` or `Public` access modifiers. For structures, `Dim` is equivalent to `Public`.

After this declaration, you have in essence created a new data type that you can use in your application. *`structureName`* can be used anywhere you'd use any of the base types (Integers, Doubles, and so on). You can declare variables of this type and manipulate them as you manipulate all other variables (with a little extra typing). The declaration for the `CheckRecord` structure shown in Figure 2.3 is as follows:

```
Structure CheckRecord
   Dim CheckNumber As Integer
   Dim CheckDate As Date
   Dim CheckAmount As Single
   Dim CheckPaidTo As String
End Structure
```

This declaration must appear outside any procedure; you can't declare a `Structure` in a subroutine or function. Once declared, The `CheckRecord` structure becomes a new data type for your application.

To declare variables of this new type, use a statement such as this one:

```
Dim check1 As CheckRecord, check2 As CheckRecord
```

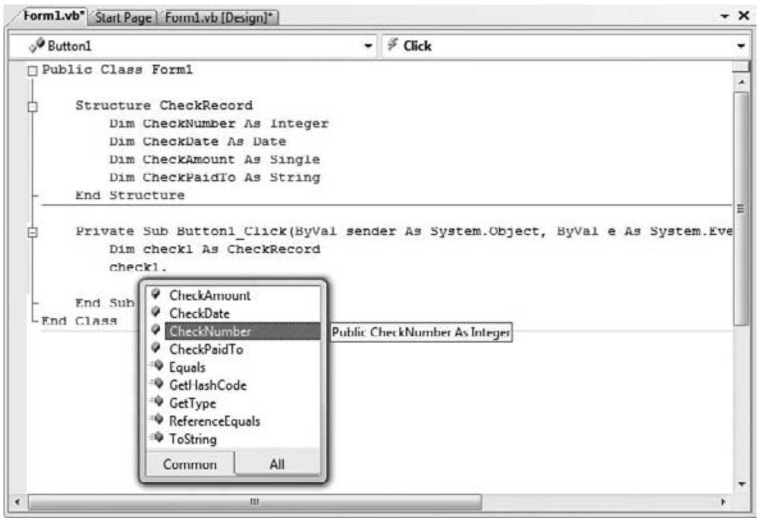
To assign a value to one of these variables, you must separately assign a value to each one of its components (they are called *fields*), which can be accessed by combining the name of the variable and the name of a field, separated by a period, as follows:

```
check1.CheckNumber = 275
```

Actually, as soon as you type the period following the variable's name, a list of all members to the `CheckRecord` structure will appear, as shown in Figure 2.4. Notice that the structure supports a few members on its own. You didn't write any code for the `Equals`, `GetType`, and `ToString` members, but they're standard members of any Structure object, and you can use them in your code. Both the `GetType` and `ToString` methods will return a string like `ProjectName.FormName` + `CheckRecord`. You can provide your own implementation of the `ToString` method, which will return a more meaningful string:

```
Public Overrides Function ToString() As String
    Return "CHECK # " & CheckNumber & " FOR " & _
           CheckAmount.ToString("C")
End Function
```

FIGURE 2.4
Variables of custom types expose their members as properties.



```
Public Class Form1

    Structure CheckRecord
        Dim CheckNumber As Integer
        Dim CheckDate As Date
        Dim CheckAmount As Single
        Dim CheckPaidTo As String
    End Structure

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.Eve
        Dim check1 As CheckRecord
        check1.

    End Sub
End Class
```

As you understand, structures are a lot like objects that expose their fields as properties and then expose a few members of their own. The following statements initialize a `CheckRecord` variable:

```
check2.CheckNumber = 275
check2.CheckDate = #09/12/2008#
check2.CheckAmount = 104.25
check2.CheckPaidTo = "Gas Co."
```

You can also create *arrays of structures* with a declaration such as the following (arrays are discussed later in this chapter):

```
Dim Checks(100) As CheckRecord
```

Each element in this array is a `CheckRecord` structure and it holds all the fields of a given check. To access the fields of the third element of the array, use the following notation:

```
Checks(2).CheckNumber = 275
Checks(2).CheckDate = #09/12/2008#
Checks(2).CheckAmount = 104.25
Checks(2).CheckPaidTo = "Gas Co."
```

THE NOTHING VALUE

The Nothing value is used with object variables and indicates a variable that has not been initialized. If you want to disassociate an object variable from the object it represents, set it to Nothing. The following statements create an object variable that references a brush, uses it, and then releases it:

```
Dim brush As SolidBrush
brush = New SolidBrush(Color.Blue)
{ use brush object to draw with}
brush = Nothing
```

The first statement declares the *`brush`* variable. At this point, the *`brush`* variable is Nothing. The second statement initializes the *`brush`* variable with the appropriate constructor (the brush is initialized to a specific color). After the execution of the second statement, the *`brush`* variable actually represents an object you can draw with in blue. After using it to draw something, you can release it by setting it to Nothing.

If you want to find out whether an object variable has been initialized, use the `Is` or `IsNot` operators, as shown in the following example:

```
Dim myPen As Pen
{ more statements here}
If myPen Is Nothing Then
   myPen  = New Pen(Color.Red)
End If
```

The variable *`myPen`* is initialized with the `New` constructor only if it hasn't been initialized already. If you want to release the *`myPen`* variable later in your code, you can set it to Nothing with the assignment operator. When you compare an object to Nothing, however, you can't use the equals operator; you must use the `Is` and `IsNot` operators.

Examining Variable Types

Besides setting the types of variables and the functions for converting between types, Visual Basic provides the `GetType` method, which returns a string with the variable's type

(`Int32`, `Decimal`, and so on). Any variable exposes these methods automatically, and you can call them like this:

```
Dim var As Double
Debug.WriteLine "The variable's type is " & var.GetType.ToString
```

There's also a `GetType` operator, which accepts as an argument a type and returns a Type object for the specific data type. The `GetType` method and `GetType` operator are used mostly in `If` structures, like the following one:

```
If var.GetType() Is GetType(Double) Then
   { code to handle a Double value}
End If
```

Notice that the code doesn't reference data type names directly. Instead, it uses the value returned by the `GetType` operator to retrieve the type of the class System.Double and then compares this value to the variable's type with the `Is` (or the `IsNot`) keyword.

Is It a Number, String, or Date?

Another set of Visual Basic functions returns variables' data types, but not the exact type. They return a True/False value indicating whether a variable holds a numeric value, a date or an array. The following functions are used to validate user input, as well as data stored in files, before you process them.

IsNumeric() Returns True if its argument is a number (Short, Integer, Long, Single, Double, Decimal). Use this function to determine whether a variable holds a numeric value before passing it to a procedure that expects a numeric value or before processing it as a number. The following statements keep prompting the user with an InputBox for a numeric value. The user must enter a numeric value or click the Cancel button to exit. As long as the user enters non-numeric values, the Input box keeps popping up and prompting for a numeric value:

```
Dim strAge as String = ""
Dim Age As Integer
While Not IsNumeric(strAge)
    strAge = InputBox("Please enter your age")
End While
Age  =  Convert.ToInt16(strAge)
```

The variable *`strAge`* is initialized to a non-numeric value so that the `While...End While` loop will be executed at least once.

IsDate() Returns True if its argument is a valid date (or time). The following expressions return True because they all represent valid dates:

```
IsDate(#10/12/2010#)
IsDate("10/12/2010")
IsDate("October 12, 2010")
```

If the date expression includes the day name, as in the following expression, the `IsDate()` function will return False:

```
IsDate("Sat. October 12, 2010")     ' FALSE
```

IsArray() Returns True if its argument is an array.

Why Declare Variables?

Visual Basic never enforced variable declaration (and it still doesn't), which was a good thing for the beginner programmer. When you want to slap together a ''quick-and-dirty'' program, the last thing you need is someone telling you to decide which variables you're going to use and to declare them before using them. This convenience, however, is a blessing in disguise because most programmers accustomed to the free format of Visual Basic also carry their habits of quick-and-dirty coding to large projects. When writing large applications, you will sooner or later discover that variable declaration is a necessity. It will help you write clean, strongly typed code and simplify debugging. Variable declaration eliminates the source of the most common and totally unnecessary bugs.

Let's examine the side effects of using undeclared variables in your application. To be able to get by without declaring your variables, you must set the Explicit option to Off. Let's assume that you're using the following statements to convert Euros to U.S. dollars:

```
Euro2USD = 1.462
USDollars = amount * Euro2USD
```

The first time your code refers to the *`Euro2USD`* variable name, Visual Basic creates a new variable and then uses it as if it were declared.

Suppose that the variable *`Euro2USD`* appears in many places in your application. If in one of these places you type Euro2UDS, and the program doesn't enforce variable declaration, the compiler will create a new variable, assign it the value zero, and then use it. Any amount converted with the `Euro2UDS` variable will be zero! If the application enforces variable declaration, the compiler will complain (the `Euro2UDS` variable hasn't been declared), and you will catch the error right in the editor, as you type.

A Variable's Scope

In addition to its type, a variable also has a scope. The *scope* (or *visibility*) of a variable is the section of the application that can see and manipulate the variable. If a variable is declared within a procedure, only the code in the specific procedure has access to that variable; this variable doesn't exist for the rest of the application. When the variable's scope is limited to a procedure, it's called *local*.

Suppose that you're coding the `Click` event of a button to calculate the sum of all even numbers in the range 0 to 100. One possible implementation is shown in Listing 2.4.

LISTING 2.4: Summing Even Numbers

```
Private Sub Button1_Click(ByVal sender As Object, _
                ByVal e As System.EventArguments) _
```

```
                 Handles Button1.Click
    Dim i As Integer
    Dim Sum As Integer
    For i = 0 to 100 Step 2
       Sum = Sum + i
    Next
   MsgBox "The sum is " & Sum.ToString
End Sub
```

The variables *i* and *Sum* are local to the `Button1_Click()` procedure. If you attempt to set the value of the *Sum* variable from within another procedure, Visual Basic will complain that the variable hasn't been declared. (Or, if you have turned off the Explicit option, it will create another `Sum` variable, initialize it to zero, and then use it. But this won't affect the variable *Sum* in the `Button1_Click()` subroutine.) The *Sum* variable is said to have *procedure-level scope:* It's visible within the procedure and invisible outside the procedure.

Sometimes, however, you'll need to use a variable with a broader scope; a variable that's available to all procedures within the same file. This variable, which must be declared outside any procedure, is said to have a *module-level scope.* In principle, you could declare all variables outside the procedures that use them, but this would lead to problems. Every procedure in the file would have access to any variable, and you would need to be extremely careful not to change the value of a variable without good reason. Variables that are needed by a single procedure (such as loop counters) should be declared in that procedure.

Another type of scope is the *block-level scope.* Variables introduced in a block of code, such as an `If` statement or a loop, are local to the block but invisible outside the block. Let's revise the previous code segment so that it calculates the sum of squares. To carry out the calculation, we first compute the square of each value and then sum the squares. The square of each value is stored to a variable that won't be used outside the loop, so we can define the *sqrValue* variable in the loop's block and make it local to this specific loop, as shown in Listing 2.5.

Listing 2.5: A Variable Scoped in Its Own Block

```
Private Sub Button1_Click(ByVal sender As Object, _
               ByVal e As System.EventArguments) _
               Handles Button1.Click
   Dim i, Sum As Integer
   For i = 0 to 100 Step 2
      Dim sqrValue As Integer
      sqrValue = i * i
      Sum = Sum + sqrValue
   Next
   MsgBox "The sum of the squares is " & Sum
End Sub
```

The *sqrValue* variable is not visible outside the block of the `For...Next` loop. If you attempt to use it before the `For` statement or after the `Next` statement, VB will throw an exception.

The *`sqrValue`* variable maintains its value between iterations. The block-level variable is not initialized at each iteration, even though there's a `Dim` statement in the loop.

Finally, in some situations, the entire application must access a certain variable. In this case, the variable must be declared as Public. Public variables have a *global* scope: They are visible from any part of the application. To declare a public variable, use the `Public` statement in place of the `Dim` statement. Moreover, you can't declare public variables in a procedure. If you have multiple forms in your application and you want the code in one form to see a certain variable in another form, you can use the `Public` modifier.

The `Public` keyword makes the variable available not only to the entire project, but also to all projects that reference the current project. If you want your variables to be public within a project (in other words, available to all procedures in any module in the project) but invisible to referencing projects, use the `Friend` keyword in the declaration of the module. Variables you want to use throughout your project, but to not become available to other projects that reference this one, should be declared as `Friend`.

So, why do we need so many types of scope? You'll develop a better understanding of scope and which type of scope to use for each variable as you get involved in larger projects. In general, you should try to limit the scope of your variables as much as possible. If all variables were declared within procedures, you could use the same name for storing a temporary value in each procedure and be sure that one procedure's variables wouldn't interfere with those of another procedure, even if you use the same name.

A Variable's Lifetime

In addition to type and scope, variables have a *lifetime*, which is the period for which they retain their value. Variables declared as `Public` exist for the lifetime of the application. Local variables, declared within procedures with the `Dim` or `Private` statement, live as long as the procedure. When the procedure finishes, the local variables cease to exist, and the allocated memory is returned to the system. Of course, the same procedure can be called again. In this case, the local variables are re-created and initialized again. If a procedure calls another, its local variables retain their values while the called procedure is running.

You also can force a local variable to preserve its value between procedure calls by using the `Static` keyword. Suppose that the user of your application can enter numeric values at any time. One of the tasks performed by the application is to track the average of the numeric values. Instead of adding all the values each time the user adds a new value and dividing by the count, you can keep a running total with the function `RunningAvg()`, which is shown in Listing 2.6.

LISTING 2.6: Calculations with Global Variables

```
Function RunningAvg(ByVal newValue As Double) As Double
   CurrentTotal = CurrentTotal + newValue
   TotalItems = TotalItems + 1
   RunningAvg = CurrentTotal / TotalItems
End Function
```

You must declare the variables *`CurrentTotal`* and *`TotalItems`* outside the function so that their values are preserved between calls. Alternatively, you can declare them in the function with the `Static` keyword, as shown in Listing 2.7.

Listing 2.7: Calculations with Local Static Variables

```
Function RunningAvg(ByVal newValue As Double) As Double
   Static CurrentTotal As Double
   Static TotalItems As Integer
   CurrentTotal = CurrentTotal + newValue
   TotalItems = TotalItems + 1
   RunningAvg = CurrentTotal / TotalItems
End Function
```

The advantage of using static variables is that they help you minimize the number of total variables in the application. All you need is the running average, which the `RunningAvg()` function provides without making its variables visible to the rest of the application. Therefore, you don't risk changing the variables' values from within other procedures.

Variables declared in a module outside any procedure take effect when the form is loaded and cease to exist when the form is unloaded. If the form is loaded again, its variables are initialized as if it's being loaded for the first time.

Variables are initialized when they're declared, according to their type. Numeric variables are initialized to zero, string variables are initialized to a blank string, and object variables are initialized to Nothing.

Constants

Some variables don't change value during the execution of a program. These variables are *constants* that appear many times in your code. For instance, if your program does math calculations, the value of pi (3.14159. . .) might appear many times. Instead of typing the value 3.14159 over and over again, you can define a constant, name it `pi`, and use the name of the constant in your code. The statement

```
circumference = 2 * pi * radius
```

is much easier to understand than the equivalent

```
circumference = 2 * 3.14159 * radius
```

You could declare `pi` as a variable, but constants are preferred for two reasons:

Constants don't change value. This is a safety feature. After a constant has been declared, you can't change its value in subsequent statements, so you can be sure that the value specified in the constant's declaration will take effect in the entire program.

Constants are processed faster than variables. When the program is running, the values of constants don't have to be looked up. The compiler substitutes constant names with their values, and the program executes faster.

The manner in which you declare constants is similar to the manner in which you declare variables, except that you use the `Const` keyword and in addition to supplying the constant's name, you must also supply a value, as follows:

```
Const constantname As type = value
```

Constants also have a scope and can be `Public` or `Private`. The constant `pi`, for instance, is usually declared in a module as `Public` so that every procedure can access it:

```
Public Const pi As Double = 3.14159265358979
```

The name of the constant follows the same rules as variable names. The constant's value is a literal value or a simple expression composed of numeric or string constants and operators. You can't use functions in declaring constants. By the way, the specific value I used for this example need not be stored in a constant. Use the `pi` member of the Math class instead (`Math.pi`).

Constants can be strings, too, like these:

```
Const ExpDate = #31/12/1997#
Const ValidKey = "A567dfe"
```

Arrays

A standard structure for storing data in any programming language is the array. Whereas individual variables can hold single entities, such as one number, one date, or one string, *arrays* can hold sets of data of the same type (a set of numbers, a series of dates, and so on). An array has a name, as does a variable, and the values stored in it can be accessed by an index.

For example, you could use the variable *`Salary`* to store a person's salary:

```
Salary = 34000
```

But what if you wanted to store the salaries of 16 employees? You could either declare 16 variables — *`Salary1`*, *`Salary2`*, and so on up to *`Salary16`* — or declare an array with 16 elements. An array is similar to a variable: It has a name and multiple values. Each value is identified by an index (an integer value) that follows the array's name in parentheses. Each different value is an *element* of the array. If the array *`Salaries`* holds the salaries of 16 employees, the element *`Salaries(0)`* holds the salary of the first employee, the element *`Salaries(1)`* holds the salary of the second employee, and so on up to the element *`Salaries(15)`*.

Declaring Arrays

Unlike simple variables, arrays must be declared with the `Dim` (or `Public`) statement followed by the name of the array and the index of the last element in the array in parentheses — for example:

```
Dim Salary(15) As Integer
```

`Salary` is the name of an array that holds 16 values (the salaries of the 16 employees) with indices ranging from 0 to 15. `Salary(0)` is the first person's salary, `Salary(1)` the second person's salary, and so on. All you have to do is remember who corresponds to each salary, but even this data can be handled by another array. To do this, you'd declare another array of 16 elements:

```
Dim Names(15) As String
```

Then assign values to the elements of both arrays:

```
Names(0) = "Joe Doe"
Salary(0) = 34000
Names(1) = "Beth York"
Salary(1) = 62000
...
Names(15) = "Peter Smack"
Salary(15) = 10300
```

This structure is more compact and more convenient than having to hard-code the names of employees and their salaries in variables.

All elements in an array have the same data type. Of course, when the data type is Object, the individual elements can contain different kinds of data (objects, strings, numbers, and so on).

Arrays, like variables, are not limited to the basic data types. You can declare arrays that hold any type of data, including objects. The following array holds colors, which can be used later in the code as arguments to the various functions that draw shapes:

```
Dim colors(2) As Color
colors(0) = Color.BurlyWood
colors(1) = Color.AliceBlue
colors(2) = Color.Sienna
```

The Color class represents colors, and among the properties it exposes are the names of the colors it recognizes.

A better technique for storing names and salaries is to create a structure and then declare an array of this type. The following structure holds names and salaries:

```
Structure Employee
   Dim Name As String
   Dim Salary As Decimal
End Structure
```

Insert this declaration in a form's code file, outside any procedure. Then create an array of the Employee type:

```
Dim Emps(15) As Employee
```

Each element in the Emps array exposes two fields, and you can assign values to them by using statements such as the following:

```
Emps(2).Name = "Beth York"
Emps(2).Salary = 62000
```

The advantage of using an array of structures instead of multiple arrays is that the related information will always be located under the same index. The code is more compact, and you need not maintain multiple arrays.

Initializing Arrays

Just as you can initialize variables in the same line in which you declare them, you can initialize arrays, too, with the following constructor (an *array initializer,* as it's called):

```
Dim arrayname() As type = {entry0, entry1, ... entryN}
```

Here's an example that initializes an array of strings:

```
Dim Names() As String = {"Joe Doe", "Peter Smack"}
```

This statement is equivalent to the following statements, which declare an array with two elements and then set their values:

```
Dim Names(1) As String
Names(0) = "Joe Doe"
Names(1) = "Peter Smack"
```

The number of elements in the curly brackets following the array's declaration determines the dimensions of the array, and you can't add new elements to the array without resizing it. If you need to resize the array in your code dynamically, you must use the ReDim statement, as described in the section called ''Dynamic Arrays,'' later in this chapter. However, you can change the *value* of the existing elements at will, as you would with any other array.

Array Limits

The first element of an array has index 0. The number that appears in parentheses in the Dim statement is one fewer than the array's total capacity and is the array's upper limit (or upper bound). The index of the last element of an array (its upper bound) is given by the method GetUpperBound, which accepts as an argument the dimension of the array and returns the upper bound for this dimension. The arrays we examined so far are one-dimensional and the argument to be passed to the GetUpperBound method is the value 0. The total number of elements in the array is given by the method GetLength, which also accepts a dimension as an argument. The upper bound of the following array is 19, and the capacity of the array is 20 elements:

```
Dim Names(19) As Integer
```

The first element is *Names(0)*, and the last is *Names(19)*. If you execute the following statements, the highlighted values will appear in the Output window:

```
Debug.WriteLine(Names.GetLowerBound(0))
0
Debug.WriteLine(Names.GetUpperBound(0))
19
```

To assign a value to the first and last element of the *Names* array, use the following statements:

```
Names(0) = "First entry"
Names(19) = "Last entry"
```

If you want to iterate through the array's elements, use a loop like the following one:

```
Dim i As Integer, myArray(19) As Integer
For i = 0 To myArray.GetUpperBound(0)
 myArray(i) = i * 1000
Next
```

The actual number of elements in an array is given by the expression `myArray.GetUpperBound(0)` + 1. You can also use the array's `Length` property to retrieve the count of elements. The following statement will print the number of elements in the array *myArray* in the Output window:

```
Debug.WriteLine(myArray.Length)
```

Still confused with the zero-indexing scheme, the count of elements, and the index of the last element in the array? You can make the array a little larger than it needs to be and ignore the first element. Just make sure that you never use the zero element in your code — don't store a value in the element *Array(0)*, and you can then ignore this element. To get 20 elements, declare an array with 21 elements as `Dim MyArray(20) As type` and then ignore the first element.

Multidimensional Arrays

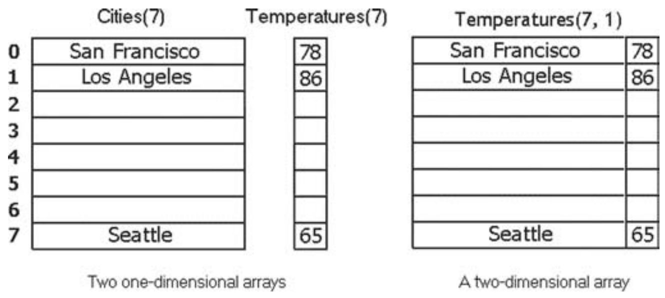
One-dimensional arrays, such as those presented so far, are good for storing long sequences of one-dimensional data (such as names or temperatures). But how would you store a list of cities *and* their average temperatures in an array? Or names and scores; years and profits; or data with more than two dimensions, such as products, prices, and units in stock? In some situations, you will want to store sequences of multidimensional data. You can store the same data more conveniently in an array of as many dimensions as needed.

Figure 2.5 shows two one-dimensional arrays — one of them with city names, the other with temperatures. The name of the third city would be `City(2)`, and its temperature would be `Temperature(2)`.

A two-dimensional array has two indices: The first identifies the row (the order of the city in the array), and the second identifies the column (city or temperature). To access the name and temperature of the third city in the two-dimensional array, use the following indices:

```
Temperatures(2, 0)     ' is the third city's name
Temperatures(2, 1)     ' is the third city's average temperature
```

FIGURE 2.5
Two one-dimensional arrays and the equivalent two-dimensional array



The benefit of using multidimensional arrays is that they're conceptually easier to manage. Suppose that you're writing a game and want to track the positions of certain pieces on a board. Each square on the board is identified by two numbers: its horizontal and vertical coordinates. The obvious structure for tracking the board's squares is a two-dimensional array, in which the first index corresponds to the row number, and the second corresponds to the column number. The array could be declared as follows:

```
Dim Board(9, 9) As Integer
```

When a piece is moved from the square in the first row and first column to the square in the third row and fifth column, you assign the value 0 to the element that corresponds to the initial position:

```
Board(0, 0) = 0
```

And you assign 1 to the square to which it was moved to indicate the new state of the board:

```
Board(2, 4) = 1
```

To find out whether a piece is on the top-left square, you'd use the following statement:

```
If Board(0, 0) = 1 Then
   { piece found}
Else
   { empty square}
End If
```

This notation can be extended to more than two dimensions. The following statement creates an array with 1,000 elements (10 by 10 by 10):

```
Dim Matrix(9, 9, 9)
```

You can think of a three-dimensional array as a cube made up of overlaid two-dimensional arrays, such as the one shown in Figure 2.6.

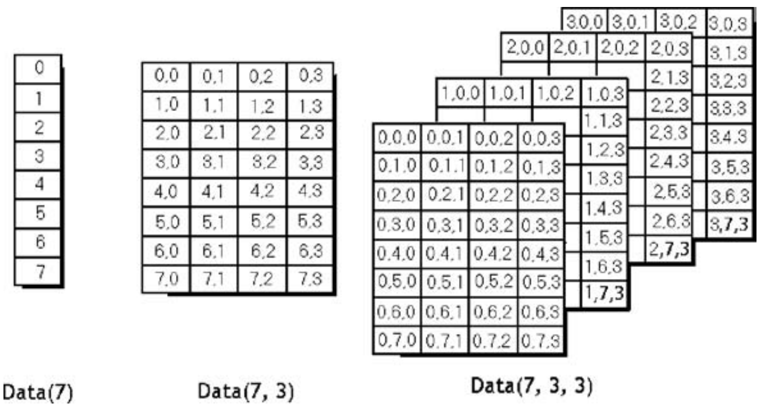


FIGURE 2.6
Pictorial representations of one-, two-, and three-dimensional arrays

It is possible to initialize a multidimensional array with a single statement, just as you do with a one-dimensional array. You must insert enough commas in the parentheses following the array name to indicate the array's rank. The following statements initialize a two-dimensional array and then print a couple of its elements:

```
Dim a(,) As Integer = {{10, 20, 30}, {11, 21, 31}, {12, 22, 32}}
Console.WriteLine(a(0, 1))     ' will print 20
Console.WriteLine(a(2, 2))     ' will print 32
```

You should break the line that initializes the dimensions of the array into multiple lines to make your code easier to read. Just insert the line continuation character at the end of each continued line:

```
Dim a(,) As Integer = {{10, 20, 30}, _
                       {11, 21, 31}, _
                       {12, 22, 32}}
```

If the array has more than one dimension, you can find out the number of dimensions with the `Array.Rank` property. Let's say you have declared an array for storing names and salaries by using the following statements:

```
Dim Employees(1,99) As Employee
```

To find out the number of dimensions, use the following statement:

```
Employees.Rank
```

When using the Length property to find out the number of elements in a multidimensional array, you will get back the total number of elements in the array (2 × 100 for our example). To find out the number of elements in a specific dimension, use the GetLength method, passing as an argument a specific dimension. The following expressions will return the number of elements in the two dimensions of the array:

```
Debug.WriteLine(Employees.GetLength(0))
2
Debug.WriteLine(Employees.GetLength(1))
100
```

Because the index of the first array element is zero, the index of the last element is the length of the array minus 1. Let's say you have declared an array with the following statement to store player statistics for 15 players, and there are five values per player:

```
Dim Statistics(14, 4) As Integer
```

The following statements will return the highlighted values shown beneath them:

```
Debug.WriteLine(Statistics.Rank)
2                           ' dimensions in array
Debug.WriteLine(Statistics.Length)
75                          ' total elements in array
Debug.WriteLine(Statistics.GetLength(0))
15                          ' elements in first dimension
Debug.WriteLine(Statistics.GetLength(1))
5                           ' elements in second dimension
Debug.WriteLine(Statistics.GetUpperBound(0))
14                          ' last index in the first dimension
Debug.WriteLine(Statistics.GetUpperBound(1))
4                           ' last index in the second dimension
```

Multidimensional arrays are becoming obsolete because arrays (and other collections) of custom structures and objects are more flexible and convenient.

Dynamic Arrays

Sometimes you may not know how large to make an array. Instead of making it large enough to hold the (anticipated) maximum number of data (which means that, on the average, part of the array may be empty), you can declare a *dynamic array*. The size of a dynamic array can vary during the course of the program. Or you might need an array until the user has entered a bunch of data, and the application has processed it and displayed the results. Why keep all the data in memory when it is no longer needed? With a dynamic array, you can discard the data and return the resources it occupied to the system.

To create a dynamic array, declare it as usual with the Dim statement (or Public or Private), but don't specify its dimensions:

```
Dim DynArray() As Integer
```

Later in the program, when you know how many elements you want to store in the array, use the `ReDim` statement to redimension the array, this time to its actual size. In the following example, *`UserCount`* is a user-entered value:

```
ReDim DynArray(UserCount)
```

The `ReDim` statement can appear only in a procedure. Unlike the `Dim` statement, `ReDim` is executable — it forces the application to carry out an action at runtime. `Dim` statements aren't executable, and they can appear outside procedures.

A dynamic array also can be redimensioned to multiple dimensions. Declare it with the `Dim` statement outside any procedure, as follows:

```
Dim Matrix() As Double
```

Then use the `ReDim` statement in a procedure to declare a three-dimensional array:

```
ReDim Matrix(9, 9, 9)
```

Note that the `ReDim` statement can't change the type of the array — that's why the `As` clause is missing from the `ReDim` statement. Moreover, subsequent `ReDim` statements can change the bounds of the array *`Matrix`* but not the number of its dimensions. For example, you can't use the statement `ReDim Matrix(99, 99)` later in your code.

THE *PRESERVE* KEYWORD

Each time you execute the `ReDim` statement, all the values currently stored in the array are lost. Visual Basic resets the values of the elements as if the array were just declared (it resets numeric elements to zero and String elements to empty strings.) You can, however, change the size of the array without losing its data. The `ReDim` statement recognizes the `Preserve` keyword, which forces it to resize the array without discarding the existing data. For example, you can enlarge an array by one element without losing the values of the existing elements:

```
ReDim Preserve DynamicArray(DynArray.GetUpperBound(0) + 1)
```

If the array *`DynamicArray`* held 12 elements, this statement would add one element to the array: the element *`DynamicArray(12)`*. The values of the elements with indices 0 through 11 wouldn't change.

The Bottom Line

Declare and use variables. Programs use *variables* to store information during their execution, and different types of information are stored in variables of different *types*. Dates, for example, are stored in variables of the Date type, while text is stored in variables of the String type. The various data types expose a lot of functionality that's specific to a data type; the methods provided by each data type are listed in the IntelliSense box.

Master It How would you declare and initialize a few variables?

Master It Explain briefly the Explicit, Strict, and Infer options.

Use the native data types. The CLR recognized the following data types, which you can use in your code to declare variables: Strings, Numeric types, Date and time types, Boolean data type.

All other variables, or variables that are declared without a type, are Object variables and can store any data type, or any object.

Master It How will the compiler treat the following statement?

```
Dim amount = 32
```

Create custom data types. Practical applications need to store and manipulate multiple data items, not just integers and strings. To maintain information about people, we need to store each person's name, date of birth, address, and so on. Products have a name, a description, a price, and other related items. To represent such entities in our code, we use structures, which hold many pieces of information about a specific entity together.

Master It Create a structure for storing products and populate it with data.

Use arrays. Arrays are structures for storing sets of data, as opposed to single-valued variables.

Master It How would you declare an array for storing 12 names and another one for storing 100 names and Social Security numbers?

Chapter 3

Programming Fundamentals

The one thing you should have learned about programming in Visual Basic so far is that an *application* is made up of small, self-contained segments. The code you write isn't a monolithic listing; it's made up of small segments called *procedures*, and you work on one procedure at a time.

The two types of procedures supported by Visual Basic are the topics we'll explore in this chapter: subroutines and functions — the building blocks of your applications. We'll discuss them in detail: how to call them with arguments and how to retrieve the results returned by the functions. You'll learn how to use the built-in functions that come with the language, as well as how to write your own subroutines and functions.

The statements that make up the core of the language are actually very few. The flexibility of any programming language is based on its capacity to alter the sequence in which the statements are executed through a set of so-called flow-control statements. These are the statements that literally make decisions and react differently depending on the data, user actions, or external conditions. Among other topics, in this chapter you'll learn how to do the following:

- Use Visual Basic's flow-control statements
- Write subroutines and functions
- Pass arguments to subroutines and functions

Flow-Control Statements

What makes programming languages so flexible and capable of handling every situation and programming challenge with a relatively small set of commands is their capability to examine external or internal conditions and act accordingly. Programs aren't monolithic sets of commands that carry out the same calculations every time they are executed; this is what calculators (and extremely simple programs) do. Instead, they adjust their behavior depending on the data supplied; on external conditions, such as a mouse click or the existence of a peripheral; even on abnormal conditions generated by the program itself.

In effect, the statements discussed in the first half of this chapter are what programming is all about. Without the capability to control the flow of the program, computers would just be bulky calculators. You have seen how to use the `If` statement to alter the flow of execution in previous chapters, and I assume you're somewhat familiar with these kinds of statements. In this section, you'll find a formal discussion of flow-control statements. These statements are grouped into two major categories: decision statements and looping statements.

Decision Statements

Applications need a mechanism to test conditions and take a different course of action depending on the outcome of the test. Visual Basic provides three such *decision*, or *conditional*, statements:

- `If...Then`
- `If...Then...Else`
- `Select Case`

If...Then

The `If...Then` statement tests an expression, which is known as a condition. If the condition is True, the program executes the statement(s) that follow. The `If...Then` statement can have a single-line or a multiple-line syntax. To execute one statement conditionally, use the single-line syntax as follows:

```
If condition Then statement
```

Conditions are logical expressions that evaluate to a True/False value and they usually contain comparison operators — equals (=), different (<>), less than (<), greater than (>), less than or equal to (<=), and so on — and logical operators: `And`, `Or`, `Xor`, and `Not`. Here are a few examples of valid conditions:

```
If (age1 < age2) And (age1 > 12) Then ...
If score1 = score2 Then ...
```

The parentheses are not really needed in the first sample expression, but they make the code a little easier to read. Sometimes parentheses are mandatory, to specify the order in which the expression's parts will be evaluated, just like math formulae may require parentheses to indicate the precedence of calculations. You can also execute multiple statements by separating them with colons:

```
If condition Then statement: statement: statement
```

Here's an example of a single-line `If` statement:

```
expDate = expDate + 1
If expdate.Month > 12 Then expYear = expYear + 1: expMonth = 1
```

You can break this statement into multiple lines by using the multiline syntax of the `If` statement, which delimits the statements to be executed conditionally with the `End If` statement, as shown here:

```
If expDate.Month > 12 Then
   expYear = expYear + 1
   expMonth = 1
End If
```

The `Month` property of the Date type returns the month of the date to which it's applied as a numeric value. Most VB developers prefer the multiple-line syntax of the `If` statement, even if it contains a single statement. The block of statements between the `Then` and `End If` keywords form the body of the conditional statement, and you can have as many statements in the body as needed.

Many control properties are Boolean values, which evaluate to a True/False value. Let's say that your interface contains a CheckBox control and you want to set its caption to On or Off depending on whether it's selected at the time. Here's an `If` statement that changes the caption of the CheckBox:

```
If CheckBox1.Checked Then
     CheckBox1.Text = "ON"
Else
     CheckBox1.Text = "OFF"
End If
```

This statement changes the caption of the CheckBox all right, but when should it be executed? Insert the statement in the CheckBox control's `CheckedChanged` event handler, which is fired every time the control's check mark is turned on or off, whether because of a user action on the interface or from within your code.

The expressions can get quite complicated. The following expression evaluates to True if the *`date1`* variable represents a date earlier than the year 2008 and either one of the *`score1`* and *`score2`* variables exceeds 90:

```
If (date1 < #1/1/2008) And (score1 < 90 Or score2 < 90) Then
     ' statements
End If
```

The parentheses around the last part of the comparison are mandatory, because we want the compiler to perform the following comparison first:

```
score1 < 90 Or score2 < 90
```

If either variable exceeds 90, the preceding expression evaluates to True and the initial condition is reduced to the following:

```
If (date1 < #1/1/2008) And (True) Then
```

The compiler will evaluate the first part of the expression (it will compare two dates) and finally it will combine two Boolean values with the `And` operator: if both values are True, the entire condition is True; otherwise, it's False. If you didn't use parentheses, the compiler would evaluate the three parts of the expression:

```
expression1: date1 < #1/1/2008#
expression2: score1 < 90
expression3: score2 < 90
```

Then it would combine `expression1` with `expression2` using the `And` operator, and finally it would combine the result with *`expression3`* using the `OR` operator. If *`score2`* were less than 90, the entire expression would evaluate to True, regardless of the value of the `date1` variable.

If...Then...Else

A variation of the `If...Then` statement is the `If...Then...Else` statement, which executes one block of statements if the condition is True and another block of statements if the condition is False. The syntax of the `If...Then...Else` statement is as follows:

```
If condition Then
   statementblock1
Else
   statementblock2
End If
```

Visual Basic evaluates the condition; if it's True, VB executes the first block of statements and then jumps to the statement following the `End If` statement. If the condition is False, Visual Basic ignores the first block of statements and executes the block following the `Else` keyword.

A third variation of the `If...Then...Else` statement uses several conditions, with the `ElseIf` keyword:

```
If condition1 Then
   statementblock1
ElseIf condition2 Then
   statementblock2
ElseIf condition3 Then
   statementblock3
Else
   statementblock4
End If
```

You can have any number of `ElseIf` clauses. The conditions are evaluated from the top, and if one of them is True, the corresponding block of statements is executed. The `Else` clause, which is optional, will be executed if none of the previous expressions is True. Listing 3.1 is an example of an `If` statement with `ElseIf` clauses.

LISTING 3.1: Multiple *ElseIf* Statements

```
score = InputBox("Enter score")
If score < 50 Then
   Result = "Failed"
ElseIf score < 75 Then
   Result = "Pass"
ElseIf score < 90 Then
   Result = "Very Good"
Else
   Result = "Excellent"
End If
MsgBox Result
```

MULTIPLE *IF. . .THEN* STRUCTURES VERSUS *ELSEIF*

Notice that after a True condition is found, Visual Basic executes the associated statements and skips the remaining clauses. It continues executing the program with the statement immediately after End If. All following ElseIf clauses are skipped, and the code runs a bit faster. That's why you should prefer the complicated structure with the ElseIf statements used in Listing 3.1 to this equivalent series of simple If statements:

```
If score < 50 Then
   Result = "Failed"
End If
If score < 75 And score >= 50 Then
   Result = "Pass"
End If
If score < 90 And score > =75 Then
   Result = "Very Good"
End If
If score >= 90 Then
   Result = "Excellent"
End If
```

With the multiple If statements, the compiler will generate code that evaluates all the conditions, even if the score is less than 50.

The order of the comparisons is vital when you're using multiple ElseIf statements. Had you written the previous code segment with the first two conditions switched, like the following segment, the results would be quite unexpected:

```
If score < 75 Then
   Result = "Pass"
ElseIf score < 50 Then
   Result = "Failed"
ElseIf score < 90 Then
   Result = "Very Good"
Else
   Result = "Excellent"
End If
```

Let's assume that *score* is 49. The code would compare the score variable to the value 75. Because 49 is less than 75, it would assign the value Pass to the variable *Result*, and then it would skip the remaining clauses. Thus, a student who scored 49 would have passed the test! So be extremely careful and test your code thoroughly if it uses multiple ElseIf clauses. You must either make sure they're listed in the proper order or use upper and lower limits, as in the preceding sidebar.

The *IIf()* Function

Not to be confused with the `If...Then` statement, VB provides the `IIf()` function. This built-in function accepts as an argument an expression and two values, evaluates the expression, and returns the first value if the expression is True, or the second value if the expression is False. The syntax of the `IIf()` function is the following:

```
IIf(expression, TruePart, FalsePart)
```

The *`TruePart`* and *`FalsePart`* arguments are objects. (They can be integers, strings, or any built-in or custom object.) The `IIf()` function is a more compact notation for simple `If` statements. Let's say you want to display one of the strings "Close" or "Far", depending on the value of the *`distance`* variable. Instead of a multiline `If` statement, you can call the `IIf()` function as follows:

```
IIf(distance > 1000, "Far", "Close")
```

Another typical example of the `IIf()` function is in formatting negative values. It's fairly common in business applications to display negative amounts in parentheses. Use the `IIf()` statement to write a short expression that formats negative and positive amounts differently, like the following one:

```
IIf(amount < 0, "(" & _
          Math.Abs(amount).ToString("#,###.00") & ")", _
          amount.ToString("#,###.00"))
```

The `Abs` method of the Math class returns the absolute value of a numeric value, and the string argument of the `ToString` method determines that the amount should have two decimal digits.

Select Case

An alternative to the efficient but difficult-to-read code of the multiple `ElseIf` structure is the `Select Case` structure, which compares the same expression to different values. The advantage of the `Select Case` statement over multiple `If...Then...ElseIf` statements is that it makes the code easier to read and maintain.

The `Select Case` structure evaluates a single expression at the top of the structure. The result of the expression is then compared with several values; if it matches one of them, the corresponding block of statements is executed. Here's the syntax of the `Select Case` statement:

```
Select Case expression
   Case value1
      statementblock1
   Case value2
      statementblock2
      .
      .
      .
   Case Else
      statementblockN
End Select
```

A practical example based on the `Select Case` statement is shown in Listing 3.2.

LISTING 3.2: Using the *Select Case* Statement

```
Dim Message As String
Select Case Now.DayOfWeek
    Case DayOfWeek.Monday
        message = "Have a nice week"
    Case DayOfWeek.Friday
        message = "Have a nice weekend"
    Case Else
        message = "Welcome back!"
End Select
MsgBox(message)
```

In the listing, the expression that's evaluated at the beginning of the statement is the `Now.DayOfWeek` method. This method returns a member of the `DayOfWeek` enumeration, and you can use the names of these members in your code to make it easier to read. The value of this expression is compared with the values that follow each `Case` keyword. If they match, the block of statements up to the next `Case` keyword is executed, and the program skips to the statement following the `End Select` statement. The block of the `Case Else` statement is optional, and is executed if none of the previous cases matches the expression. The first two `Case` statements take care of Fridays and Mondays, and the `Case Else` statement takes care of the other days.

Some `Case` statements can be followed by multiple values, which are separated by commas. Listing 3.3 is a revised version of the previous example.

LISTING 3.3: A Select *Case* Statement with Multiple Cases per Clause

```
Select Case Now.DayOfWeek
    Case DayOfWeek.Monday
        message = "Have a nice week"
    Case DayOfWeek.Tuesday, DayOfWeek.Wednesday, DayOfWeek.Thursday
        message = "Welcome back!"
    Case DayOfWeek.Friday, DayOfWeek.Saturday, DayOfWeek.Sunday
        message = "Have a nice weekend!"
End Select
MsgBox(message)
```

Monday, weekends, and weekdays are handled separately by three `Case` statements. The second `Case` statement handles multiple values (all workdays except for Monday and Friday). Monday is handled by a separate `Case` statement. This structure doesn't contain a `Case Else` statement because all possible values are examined in the `Case` statements; the `DayOfWeek` method can't return another value.

The `Case` statements can get a little more complex. For example, you may want to distinguish a case where the variable is larger (or smaller) than a value. To implement this logic, use the `Is` keyword, as in the following code segment that distinguishes between the first and second half of the month:

```
Select Now.Day
    Case Is < 15
        MsgBox("It's the first half of the month")
    Case Is >= 15
        MsgBox("It's the second half of the month")
End Select
```

Short-Circuiting Expression Evaluation

A common pitfall of evaluating expressions with VB is to attempt to compare a Nothing value to something. An object variable that hasn't been set to a value can't be used in calculations or comparisons. Consider the following statements:

```
Dim B As SolidBrush
B = New SolidBrush(Color.Cyan)
If B.Color = Color.White Then
    MsgBox("Please select another brush color")
End If
```

These statements create a SolidBrush object variable, the *B* variable, and then examine the brush color and prohibit the user from drawing with a white brush. The second statement initializes the brush to the cyan color. (Every shape drawn with this brush will appear in cyan.) If you attempt to use the `B` variable without initializing it, a runtime exception will be thrown: the infamous `NullReferenceException`. In our example, the exception will be thrown when the program gets to the `If` statement, because the *B* variable has no value (it's Nothing), and the code attempts to compare it to something. Nothing values can't be compared to anything. Comment out the second statement by inserting a single quote in front of it and then execute the code to see what will happen. Then restore the statement by removing the comment mark.

Let's fix it by making sure that *B* is not Nothing:

```
If B IsNot Nothing And B.Color = Color.White Then
    MsgBox("Please select another brush color")
End If
```

The `If` statement should compare the `Color` property of the `B` object, only if the `B` object is not Nothing. But this isn't the case. The `AND` operator evaluates all terms in the expression and then combines their results (True or False values) to determine the value of the expression. If they're all True, the result is also True. However, it won't skip the evaluation of some terms as soon as it hits a False value. To avoid unnecessary comparisons, use the `AndAlso` operator. The `AndAlso` operator does what the And operator should have done in the first place: It stops evaluating the remaining terms or the expression because they won't affect the result. If one of its operands is False, the entire expression will evaluate to False. In other words, if B is Nothing, there's no reason to compare its color; the entire expression will evaluate to False, regardless of the brush's color. Here's how we use the AndAlso operator:

```
If B IsNot Nothing AndAlso B.Color = Color.White Then
    MsgBox("Please select another brush color")
End If
```

The AndAlso operator is said to short-circuit the evaluation of the entire expression as soon as it runs into a False value. As soon as one of the parts in an AndAlso operation turns out to be False, the entire expression is False and there's no need to evaluate the remaining terms.

There's an equivalent operator for short-circuiting OR expressions: the OrElse operator. The OrElse operator can speed the evaluation of logical expressions a little, but it's not as important as the AndAlso operator. Another good reason for short-circuiting expression evaluation is to help performance. If the second term of an And expression takes longer to execute (it has to access a remote database, for example), you can use the AndAlso operator to make sure that it's not executed when not needed.

Loop Statements

Loop statements allow you to execute one or more lines of code repetitively. Many tasks consist of operations that must be repeated over and over again, and loop statements are an important part of any programming language. Visual Basic supports the following loop statements:

- For...Next
- Do...Loop
- While...End While

For...Next

Unlike the other two loops, the For...Next loop requires that you know the number of times that the statements in the loop will be executed. The For...Next loop has the following syntax:

```
For counter = start To end [Step increment]
   statements
Next [counter]
```

The keywords in the square brackets are optional. The arguments *counter*, *start*, *end*, and *increment* are all numeric. The loop is executed as many times as required for the counter to reach (or exceed) the end value.

In executing a For...Next loop, Visual Basic does the following:

1. Sets *counter* equal to *start*.
2. Tests to see whether *counter* is greater than end. If so, it exits the loop without executing the statements in the loop's body, not even once. If *increment* is negative, Visual Basic tests to see whether counter is less than end. If it is, it exits the loop.
3. Executes the statements in the block.

4. Increases `counter` by the amount specified with the *`increment`* argument, following the `Step` keyword. If the *`increment`* argument isn't specified, *`counter`* is increased by 1. If *`Step`* is a negative value, *`counter`* is decreased accordingly.
5. Continues with step 3.

The `For...Next` loop in Listing 3.4 scans all the elements of the numeric array *`data`* and calculates their average.

Listing 3.4: Iterating an Array with a *For...Next* Loop

```
Dim i As Integer, total As Double
For i = 0 To data.GetUpperBound(0)
   total = total + data(i)
Next i
Debug.WriteLine (total / Data.Length)
```

The single most important thing to keep in mind when working with `For...Next` loops is that the loop's ending value is set at the beginning of the loop. Changing the value of the `end` variable in the loop's body won't have any effect. For example, the following loop will be executed 10 times, not 100 times:

```
Dim endValue As Integer = 10
Dim i as Integer
For i = 0 To endValue
   endValue = 100
   { more statements }
Next i
```

You can, however, adjust the value of the *`counter`* from within the loop. The following is an example of an endless (or infinite) loop:

```
For i = 0 To 10
   Debug.WriteLine(i)
   i = i - 1
Next i
```

This loop never ends because the loop's *`counter`*, in effect, is never increased. (If you try this, press Ctrl + Break to interrupt the endless loop.)

Do Not Manipulate the Loop's Counter

Manipulating the `counter` of a `For...Next` loop is strongly discouraged. This practice will most likely lead to bugs such as infinite loops, overflows, and so on. If the number of repetitions of a loop isn't known in advance, use a `Do...Loop` or a `While...End While` structure (discussed in the following section). To jump out of a `For...Next` loop prematurely, use the `Next For` statement.

The `increment` argument can be either positive or negative. If `start` is greater than `end`, the value of `increment` must be negative. If not, the loop's body won't be executed, not even once.

VB 2008 allows you to declare the counter in the `For` statement. The counter variable ceases to exist when the program bails out of the loop:

```
For i As Integer = 1 to 10
    Debug.WriteLine(i.ToString)
Next
Debug.WriteLine(i.ToString)
```

The `i` variable is used as the loop's counter and it's not visible outside the loop. The last statement won't even compile; the editor will underline it with a wiggly line and will generate the error message *Name 'i' is not declared*.

DO…LOOP

The `Do...Loop` executes a block of statements for as long as a condition is True, or until a condition becomes True. Visual Basic evaluates an expression (the loop's condition), and if it's True, the statements in the loop's body are executed. The expression is evaluated either at the beginning of the loop (before executing any statements) or at the end of the loop (the block statements are executed at least once). If the expression is False, the program's execution continues with the statement following the loop.

There are two variations of the `Do...Loop` statement; both use the same basic model. A loop can be executed either while the condition is True or until the condition becomes True. These two variations use the keywords `While` and `Until` to specify for how long the statements will be executed. To execute a block of statements while a condition is True, use the following syntax:

```
Do While condition
   statement-block
Loop
```

To execute a block of statements until the condition becomes True, use the following syntax:

```
Do Until condition
   statement-block
Loop
```

When Visual Basic executes these loops, it first evaluates `condition`. If `condition` is False, a `Do...While` loop is skipped (the statements aren't even executed once), but a `Do...Until` loop is executed. When the `Loop` statement is reached, Visual Basic evaluates the expression again; it repeats the statement block of the `Do...While` loop if the expression is True or repeats the statements of the `Do...Until` loop if the expression is False. In short, the `Do...While` loop is executed when the condition is True, and the `Do...Until` loop is executed when the condition is False.

A last variation of the `Do...Loop` statement allows you to evaluate the condition at the end of the loop. Here's the syntax of both loops, with the evaluation of the condition at the end of the loop:

```
Do
   statement-block
Loop While condition
```

```
Do
   statement-block
Loop Until condition
```

As you can guess, the statements in the loop's body are executed at least once, because no testing takes place as the loop is entered.

Here's a typical example of using a `Do...Loop`: Suppose that the variable *`MyText`* holds some text (like the `Text` property of a TextBox control), and you want to count the words in the text. (We'll assume that there are no multiple spaces in the text and that the space character separates successive words.) To locate an instance of a character in a string, use the `IndexOf` method, which is discussed in detail in Chapter 13, ''Handling Strings, Characters, and Dates.'' This method accepts two arguments: the starting location of the search and the character being searched. The following loop repeats for as long as there are spaces in the text. Each time the `IndexOf` method finds another space in the text, it returns the location of the space. When there are no more spaces in the text, the `IndexOf` method returns the value –1, which signals the end of the loop, as shown:

```
Dim MyText As String = _
        "The quick brown fox jumped over the lazy dog"
Dim position, words As Integer
position = 0: words = 0
Do While position >= 0
    position = MyText.IndexOf(" ", position + 1)
    words += 1
Loop
MsgBox("There are " & words & " words in the text")
```

The `Do...Loop` is executed while the `IndexOf` method function returns a positive number, which means that there are more spaces (and therefore words) in the text. The variable *`position`* holds the location of each successive space character in the text. The search for the next space starts at the location of the current space plus 1 (so the program won't keep finding the same space). For each space found, the program increments the value of the *`words`* variable, which holds the total number of words when the loop ends. By the way, there are simpler methods of breaking a string into its constituent words, such as the `Split` method of the String class, which is discussed in Chapter 13. This is just an example of the `Do...While` loop.

You might notice a problem with the previous code segment: It assumes that the text contains at least one word. You should insert an `If` statement that detects zero-length strings and doesn't attempt to count words in them.

You can code the same routine with the `Until` keyword. In this case, you must continue searching for spaces until *`position`* becomes –1. Here's the same code with a different loop:

```
Dim position As Integer = 0
Dim words As Integer = 0
Do Until position = -1
   position = MyText.IndexOf(" ", position + 1)
   words = words + 1
Loop
MsgBox("There are " & words & " words in the text")
```

While...End While

The `While...End While` loop executes a block of statements as long as a condition is True. The loop has the following syntax:

```
While condition
   statement-block
End While
```

If *`condition`* is True, all statements in the bock are executed. When the `End While` statement is reached, control is returned to the `While` statement, which evaluates *`condition`* again. If *`condition`* is still True, the process is repeated. If *`condition`* is False, the program resumes with the statement following `End While`.

The loop in Listing 3.5 prompts the user for numeric data. The user can type a negative value to indicate he's done entering values and terminate the loop. As long as the user enters positive numeric values, the program keeps adding them to the `total` variable.

Listing 3.5: Reading an Unknown Number of Values

```
Dim number, total As Double
number = 0
While number => 0
   total = total + number
   number = InputBox("Please enter another value")
End While
```

I've assigned the value 0 to the *`number`* variable before the loop starts because this value isn't negative and doesn't affect the total.

Sometimes, the condition that determines when the loop will terminate can't be evaluated at the top of the loop. In these cases, we declare a Boolean value and set it to True or False from within the loop's body. Here's the outline of such a loop:

```
Dim repeatLoop As Boolean
repeatLoop = True
While repeatLoop
   { statements }
   If condition Then
      repeatLoop = True
   Else
      repeattLoop = False
   End If
End While
```

You may also see an odd loop statement like the following one:

```
While True
   { statements }
End While
```

It’s also common to express the True condition as follows:

```
While 1 = 1
```

This seemingly endless loop must be terminated from within its own body with an `Exit While` statement, which is called when a condition becomes True or False. The following loop terminates when a condition is met in the loop’s body:

```
While True
   { statements }
   If condition Then Exit While
   { more statements }
End While
```

Nested Control Structures

You can place, or *nest*, control structures inside other control structures (such as an `If...Then` block within a `For...Next` loop). Control structures in Visual Basic can be nested in as many levels as you want. The editor automatically indents the bodies of nested decision and loop structures to make the program easier to read.

When you nest control structures, you must make sure that they open and close within the same structure. In other words, you can’t start a `For...Next` loop in an `If` statement and close the loop after the corresponding `End If`. The following code segment demonstrates how to nest several flow-control statements. (The curly brackets denote that regular statements should appear in their place and will not compile, of course.)

```
For a = 1 To 100
   { statements }
   If a = 99 Then
      { statements }
   End If
   While b < a
      { statements }
      If total <= 0 Then
         { statements }
      End If
   End While
   For c = 1 to a
      { statements }
   Next c
Next a
```

I’m showing the names of the counter variables after the `Next` statements to make the code more readable. To find the matching closing statement (`Next`, `End If`, or `End While`), move down from the opening statement until you hit a line that starts at the same column. This is the matching closing statement. Notice that you don’t have to align the nested structures yourself; the editor reformats the code automatically as you edit. It also inserts the matching closing statement — the `End If` statement is inserted automatically as soon as you enter an `If` statement, for example.

Listing 3.6 shows the structure of a nested `For...Next` loop that scans all the elements of a two-dimensional array.

LISTING 3.6: Iterating through a Two-Dimensional Array

```
Dim Array2D(6, 4) As Integer
Dim iRow, iCol As Integer
For iRow = 0 To Array2D.GetUpperBound(0)
    For iCol = 0 To Array2D.GetUpperBound(1)
        Array2D(iRow, iCol) = iRow * 100 + iCol
        Debug.Write(iRow & ", " & iCol & " = " & _
                    Array2D(iRow, iCol) & "    ")
    Next iCol
    Debug.WriteLine()
Next iRow
```

The outer loop (with the *`iRow`* counter) scans each row of the array. At each iteration, the inner loop scans all the elements in the row specified by the counter of the outer loop (*`iRow`*). After the inner loop completes, the counter of the outer loop is increased by one, and the inner loop is executed again — this time to scan the elements of the next row. The loop's body consists of two statements that assign a value to the current array element and then print it in the Output window. The current element at each iteration is `Array2D(iRow, iCol)`.

You can also nest multiple `If` statements. The code in Listing 3.7 tests a user-supplied value to determine whether it's positive; if so, it determines whether the value exceeds a certain limit.

LISTING 3.7: Simple Nested *If* Statements

```
Dim Income As Decimal
Income = Convert.ToDecimal(InputBox("Enter your income"))
If Income > 0 Then
   If Income > 12000 Then
      MsgBox "You will pay taxes this year"
   Else
      MsgBox "You won't pay any taxes this year"
   End If
Else
   MsgBox "Bummer"
End If
```

The *`Income`* variable is first compared with zero. If it's negative, the `Else` clause of the `If...Then` statement is executed. If it's positive, it's compared with the value 12,000, and depending on the outcome, a different message is displayed. The code segment shown here doesn't perform any extensive validations and assumes that the user won't enter a string when prompted for her income.

The *Exit* Statement

The Exit statement allows you to exit prematurely from a block of statements in a control structure, from a loop, or even from a procedure. Suppose that you have a For...Next loop that calculates the square root of a series of numbers. Because the square root of negative numbers can't be calculated (the Math.Sqrt method will generate a runtime error), you might want to halt the operation if the array contains an invalid value. To exit the loop prematurely, use the Exit For statement as follows:

```
For i = 0 To UBound(nArray)
    If nArray(i) < 0 Then
        MsgBox("Can't complete calculations" & vbCrLf & _
               "Item " & i.ToString & " is negative! "
        Exit For
    End If
    nArray(i) = Math.Sqrt(nArray(i))
Next
```

If a negative element is found in this loop, the program exits the loop and continues with the statement following the Next statement.

There are similar Exit statements for the Do loop (Exit Do), the While loop (Exit While), the Select statement (Exit Select), and for functions and subroutines (Exit Function and Exit Sub). If the previous loop was part of a function, you might want to display an error and exit not only the loop, but also the function itself by using the Exit Function statement.

Writing and Using Procedures

The idea of breaking a large application into smaller, more manageable sections is not new to computing. Few tasks, programming or otherwise, can be managed as a whole. The event handlers are just one example of breaking a large application into smaller tasks.

For example, when you write code for a control's Click event, you concentrate on the event at hand — namely, how the program should react to the Click event. What happens when the control is double-clicked or when another control is clicked is something you will worry about later — in another control's event handler. This divide-and-conquer approach isn't unique to programming events. It permeates the Visual Basic language, and even the longest applications are written by breaking them into small, well-defined, easily managed tasks. Each task is performed by a separate procedure that is written and tested separately from the others. As mentioned earlier, the two types of procedures supported by Visual Basic are subroutines and functions.

Subroutines usually perform actions and they don't return any result. Functions, on the other hand, perform some calculations and return a value. This is the only difference between subroutines and functions. Both subroutines and functions can accept *arguments*, which are values you pass to the procedure when you call it. Usually, the arguments are the values on which the procedure's code acts. Arguments and the related keywords are discussed in detail in the ''Arguments'' section later in this chapter.

Subroutines

A *subroutine* is a block of statements that carries out a well-defined task. The block of statements is placed within a set of Sub...End Sub statements and can be invoked by name.

The following subroutine displays the current date in a message box and can be called by its name, ShowDate():

```
Sub ShowDate()
   MsgBox(Now().ToShortDateString)
End Sub
```

Normally, the task performed by a subroutine is more complicated than this; but even this simple subroutine is a block of code isolated from the rest of the application. The statements in a subroutine are executed, and when the End Sub statement is reached, control returns to the calling program. It's possible to exit a subroutine prematurely by using the Exit Sub statement.

All variables declared within a subroutine are local to that subroutine. When the subroutine exits, all variables declared in it cease to exist.

Most procedures also accept and act upon arguments. The ShowDate() subroutine displays the current date in a message box. If you want to display any other date, you have to implement it differently and add an argument to the subroutine:

```
Sub ShowDate(ByVal birthDate As Date)
   MsgBox(birthDate.ToShortDateString)
End Sub
```

birthDate is a variable that holds the date to be displayed; its type is Date. The ByVal keyword means that the subroutine sees a copy of the variable, not the variable itself. What this means practically is that the subroutine can't change the value of the variable passed by the calling application. To display the current date in a message box, you must call the ShowDate() subroutine as follows from within your program:

```
ShowDate()
```

To display any other date with the second implementation of the subroutine, use a statement like the following:

```
Dim myBirthDate = #2/9/1960#
ShowDate(myBirthDate)
```

Or, you can pass the value to be displayed directly without the use of an intermediate variable:

```
ShowDate(#2/9/1960#)
```

If you later decide to change the format of the date, there's only one place in your code you must edit: the statement that displays the date from within the ShowDate() subroutine.

Functions

A *function* is similar to a subroutine, but a function returns a result. Because they return values, functions — like variables — have types. The value you pass back to the calling program from a function is called the *return value*, and its type must match the type of the function. Functions accept arguments, just like subroutines. The statements that make up a function are placed in a set of Function. . .End Function statements, as shown here:

```
Function NextDay() As Date
    Dim theNextDay As Date
    theNextDay = Now.AddDays(1)
    Return theNextDay
End Function
```

The Function keyword is followed by the function name and the As keyword that specifies its type, similar to a variable declaration. AddDays is a method of the Date type, and it adds a number of days to a Date value. The NextDay() function returns tomorrow's date by adding one day to the current date. NextDay() is a custom function, which calls the built-in AddDays method to complete its calculations.

The result of a function is returned to the calling program with the Return statement, which is followed by the value you want to return from your function. This value, which is usually a variable, must be of the same type as the function. In our example, the Return statement happens to be the last statement in the function, but it could appear anywhere; it could even appear several times in the function's code. The first time a Return statement is executed, the function terminates, and control is returned to the calling program.

You can also return a value to the calling routine by assigning the result to the name of the function. The following is an alternate method of coding the NextDay() function:

```
Function NextDay() As Date
   NextDay = Now.AddDays(1)
End Function
```

Notice that this time I've assigned the result of the calculation to the function's name directly and didn't use a variable. This assignment, however, doesn't terminate the function like the Return statement. It sets up the function's return value, but the function will terminate when the End Function statement is reached, or when an Exit Function statement is encountered.

Similar to variables, a custom function has a name that must be unique in its scope (which is also true for subroutines, of course). If you declare a function in a form, the function name must be unique in the form. If you declare a function as Public or Friend, its name must be unique in the project. Functions have the same scope rules as variables and can be prefixed by many of the same keywords. In effect, you can modify the default scope of a function with the keywords Public, Private, Protected, Friend, and Protected Friend. In addition, functions have types, just like variables, and they're declared with the As keyword.

Suppose that the function CountWords() counts the number of words, and the function CountChars() counts the number of characters in a string. The average length of a word could be calculated as follows:

```
Dim longString As String, avgLen As Double
longString = TextBox1.Text
avgLen = CountChars(longString) / CountWords(longString)
```

The first executable statement gets the text of a TextBox control and assigns it to a variable, which is then used as an argument to the two functions. When the third statement executes, Visual

Basic first calls the functions `CountChars()` and `CountWords()` with the specified arguments, and then divides the results they return.

You can call functions in the same way that you call subroutines, but the result won't be stored anywhere. For example, the function `Convert()` might convert the text in a text box to uppercase and return the number of characters it converts. Normally, you'd call this function as follows:

```
nChars = Convert()
```

If you don't care about the return value — you only want to update the text on a TextBox control — you would call the `Convert()` function with the following statement:

```
Convert()
```

Arguments

Subroutines and functions aren't entirely isolated from the rest of the application. Most procedures accept arguments from the calling program. Recall that an *argument* is a value you pass to the procedure and on which the procedure usually acts. This is how subroutines and functions communicate with the rest of the application.

Subroutines and functions may accept any number of arguments, and you must supply a value for each argument of the procedure when you call it. Some of the arguments may be optional, which means you can omit them; you will see shortly how to handle optional arguments.

The custom function `Min()`, for instance, accepts two numbers and returns the smaller one:

```
Function Min(ByVal a As Single, ByVal b As Single) As Single
   Min = IIf(a < b, a, b)
End Function
```

`IIf()` is a built-in function that evaluates the first argument, which is a logical expression. If the expression is True, the `IIf()` function returns the second argument. If the expression is False, the function returns the third argument.

To call the `Min()` custom function, use a few statements like the following:

```
Dim val1 As Single = 33.001
Dim val2 As Single = 33.0011
Dim smallerVal as Single
smallerVal = Min(val1, val2)
Debug.Write("The smaller value is " & smallerVal)
```

If you execute these statements (place them in a button's `Click` event handler), you will see the following in the Immediate window:

```
The smaller value is 33.001
```

If you attempt to call the same function with two Double values, with a statement like the following, you will see the value 3.33 in the Immediate window:

```
Debug.WriteLine(Min(3.33000000111, 3.33000000222))
```

The compiler converted the two values from Double to Single data type and returned one of them. Which one is it? It doesn't make a difference because when converted to Single, both values are the same.

Interesting things will happen if you attempt to use the `Min()` function with the Strict option turned on. Insert the statement `Option Strict On` at the very beginning of the file, or set Option Strict to On in the Compile tab of the project's Properties pages. The editor will underline the statement that implements the `Min()` function: the `IIf()` function. The `IIf()` function accepts two Object variables as arguments, and returns one of them as its result. The Strict option prevents the compiler from converting an Object to a numeric variable. To use the `IIf()` function with the Strict option, you must change its implementation as follows:

```
Function Min(ByVal a As Object, ByVal b As Object) As Object
   Min = IIf(Val(a) < Val(b), a, b)
End Function
```

It's possible to implement a `Min()` function that can compare arguments of all types (integers, strings, dates, and so on). We'll return to this sample function later in this chapter, in the section ''Overloading Functions.''

Argument-Passing Mechanisms

One of the most important topics in implementing your own procedures is the mechanism used to pass arguments. The examples so far have used the default mechanism: passing arguments by value. The other mechanism is passing them by reference. Although most programmers use the default mechanism, it's important to know the difference between the two mechanisms and when to use each.

By Value versus by Reference

When you pass an argument by value, the procedure sees only a copy of the argument. Even if the procedure changes it, the changes aren't permanent; in other words, the value of the original variable passed to the procedure isn't affected. The benefit of passing arguments by value is that the argument values are isolated from the procedure, and only the code segment in which they are declared can change their values. This is the default argument-passing mechanism in Visual Basic 2008.

In VB 6, the default argument-passing mechanism was by reference, and this is something you should be aware of, especially if you're migrating VB 6 code to VB 2008.

To specify the arguments that will be passed by value, use the `ByVal` keyword in front of the argument's name. If you omit the `ByVal` keyword, the editor will insert it automatically because it's the default option. To declare that the `Degrees()` function's argument is passed by value, use the `ByVal` keyword in the argument's declaration as follows:

```
Function Degrees(ByVal Celsius as Single) As Single
   Return((9 / 5) * Celsius + 32)
End Function
```

To see what the ByVal keyword does, add a line that changes the value of the argument in the function:

```
Function Degrees(ByVal Celsius as Single) As Single
   Return((9 / 5) * Celsius + 32)
   Celsius = 0
End Function
```

Now call the function as follows:

```
CTemp = InputBox("Enter temperature in degrees Celsius")
MsgBox(CTemp.ToString & " degrees Celsius are " & _
       Degrees((CTemp)) & " degrees Fahrenheit")
```

If you enter the value 32, the following message is displayed:

```
32 degrees Celsius are 89.6 degrees Fahrenheit
```

Replace the ByVal keyword with the ByRef keyword in the function's definition and call the function as follows:

```
Celsius = 32.0
FTemp = Degrees(Celsius)
MsgBox(Celsius.ToString & " degrees Celsius are " & FTemp & _
       " degrees Fahrenheit")
```

This time the program displays the following message:

```
0 degrees Celsius are 89.6 degrees Fahrenheit
```

When the *Celsius* argument was passed to the Degrees() function, its value was 32. But the function changed its value, and upon return it was 0. Because the argument was passed by reference, any changes made by the procedure affected the variable permanently. As a result, when the calling program attempted to use it, the variable had a different value than expected.

Returning Multiple Values

If you want to write a function that returns more than a single result, you will most likely pass additional arguments by reference and set their values from within the function's code. The CalculateStatistics() function, shown a little later in this section, calculates the basic statistics of a data set. The values of the data set are stored in an array, which is passed to the function by reference. The CalculateStatistics() function must return two values: the average and standard deviation of the data set. Here's the declaration of the CalculateStatistics() function:

```
Function CalculateStatistics(ByRef Data() As Double, _
            ByRef Avg As Double, ByRef StDev As Double) As Integer
```

The function returns an integer, which is the number of values in the data set. The two important values calculated by the function are returned in the *Avg* and *StDev* arguments:

```
Function CalculateStatistics(ByRef Data() As Double, _
              ByRef Avg As Double, ByRef StDev As Double) As Integer
   Dim i As Integer, sum As Double, sumSqr As Double, points As Integer
   points = Data.Length
   For i = 0 To points - 1
      sum = sum + Data(i)
      sumSqr = sumSqr + Data(i) ^ 2
   Next
   Avg = sum / points
   StDev = System.Math.Sqrt(sumSqr / points - Avg ^ 2)
   Return(points)
End Function
```

To call the `CalculateStatistics()` function from within your code, set up an array of Doubles and declare two variables that will hold the average and standard deviation of the data set:

```
Dim Values(99) As Double
' Statements to populate the data set
Dim average, deviation As Double
Dim points As Integer
points = Stats(Values, average, deviation)
Debug.WriteLine points & " values processed."
Debug.WriteLine "The average is " & average & " and"
Debug.WriteLine "the standard deviation is " & deviation
```

Using `ByRef` arguments is the simplest method for a function to return multiple values. However, the definition of your functions might become cluttered, especially if you want to return more than a few values. Another problem with this technique is that it's not clear whether an argument must be set before calling the function. As you will see shortly, it is possible for a function to return an array or a custom structure with fields for any number of values.

Passing Objects as Arguments

When you pass objects as arguments, they're passed by reference, even if you have specified the `ByVal` keyword. The procedure can access and modify the members of the object passed as an argument, and the new value will be visible in the procedure that made the call.

The following code segment demonstrates this. The object is an ArrayList, which is an enhanced form of an array. The ArrayList is discussed in detail later in the book, but to follow this example all you need to know is that the `Add` method adds new items to the ArrayList, and you can access individual items with an index value, similar to an array's elements. In the `Click` event handler of a Button control, create a new instance of the ArrayList object and call the `PopulateList()` subroutine to populate the list. Even if the ArrayList object is passed to the subroutine by value, the subroutine has access to its items:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles Button1.Click
   Dim aList As New ArrayList()
   PopulateList(aList)
```

```
      Debug.WriteLine(aList(0).ToString)
      Debug.WriteLine(aList(1).ToString)
      Debug.WriteLine(aList(2).ToString)
   End Sub

   Sub PopulateList(ByVal list As ArrayList)
      list.Add("1")
      list.Add("2")
      list.Add("3")
   End Sub
```

The same is true for arrays and all other collections. Even if you specify the `ByVal` keyword, they're passed by reference. A more elegant method of modifying the members of a structure from within a procedure is to implement the procedure as a function returning a structure, as explained in the section ''Functions Returning Structures,'' later in this chapter.

Built-in Functions

VB 2008 provides many functions that implement common or complicated tasks, and you can look them up in the documentation. (You'll find them in the Visual Studio➢ Visual Basic ➢ Reference ➢ Functions branch of the contents tree in the Visual Studio documentation.) There are functions for the common math operations, functions to perform calculations with dates (these are truly complicated operations), financial functions, and many more. When you use the built-in functions, you don't have to know how they work internally — just how to call them and how to retrieve the return value.

The `Pmt()` function, for example, calculates the monthly payments on a loan. All you have to know is the arguments you must pass to the function and how to retrieve the result. The syntax of the `Pmt()` function is the following, where *`MPay`* is the monthly payment, *`Rate`* is the monthly interest rate, and *`NPer`* is the number of payments (the duration of the loan in months). *`PV`* is the loan's present value (the amount you took from the bank):

```
   MPay = Pmt(Rate, NPer, PV, FV, Due)
```

`Due` is an optional argument that specifies when the payments are due (the beginning or the end of the month), and *`FV`* is another optional argument that specifies the future value of an amount. This isn't needed in the case of a loan, but it can help you calculate how much money you should deposit each month to accumulate a target amount over a given time. (The amount returned by the `Pmt()` function is negative because it's a negative cash flow — it's money you owe — so pay attention to the sign of your values.)

To calculate the monthly payment for a $20,000 loan paid off over a period of six years at a fixed interest rate of 7.25%, you call the `Pmt()` function, as shown in Listing 3.8.

Listing 3.8: Using the *Pmt()* Built-in Function

```
   Dim mPay, totalPay As Double
   Dim Duration As Integer = 6 * 12
   Dim Rate As Single = (7.25 / 100) / 12
   Dim Amount As Single = 20000
   mPay = -Pmt(Rate, Duration, Amount)
```

```
totalPay = mPay * Duration
MsgBox("Your monthly payment will be " & mPay.ToString("C") & _
           vbCrLf & "You will pay back a total of " & _
           totalPay.ToString("C"))
```

Notice that the interest (7.25%) is divided by 12 because the function requires the monthly interest. The value returned by the function is the monthly payment for the loan specified with the *`Duration`*, *`Amount`*, and *`Rate`* variables. If you place the preceding lines in the `Click` event handler of a Button, run the project, and then click the button, the following message will appear in a message box:

```
Your monthly payment will be $343.39
You will pay back a total of $24,723.80
```

Let's say you want to accumulate $40,000 over the next 15 years by making monthly deposits of equal amounts. To calculate the monthly deposit amount, you must call the `Pmt()` function, passing 0 as the present value and the target amount as the future value. Replace the statements in the button's `Click` event handler with the following and run the project:

```
Dim mPay As Double
Dim Duration As Integer = 15 * 12
Dim Rate As Single = (4.0 / 100.0) / 12
Dim Amount As Single = -40000.0
mPay = Pmt(Rate, Duration, 0, Amount)
MsgBox("A monthly deposit of " & mPay.ToString("C") & vbCrLf & _
       "every month will yield $40,000 in 15 years")
```

It turns out that if you want to accumulate $40,000 over the next 15 years to send your kid to college, assuming a constant interest rate of 4%, you must deposit $162.54 every month. You'll put out almost $30,000, and the rest will be the interest you earn.

`Pmt()` is one of the simpler financial functions provided by the Framework, but most of us would find it really difficult to write the code for this function. Because financial calculations are quite common in business programming, many of the functions you might need already exist, and all you need to know is how to call them. If you're developing financial applications, you should look up the financial functions in the documentation.

Let's look at another useful built-in function, the `MonthName()` function, which accepts as an argument a month number and returns the name of the month. This function is not as trivial as you might think because it returns the month name or its abbreviation in the language of the current culture. The `MonthName()` function accepts as arguments the month number and a True/False value that determines whether it will return the abbreviation or the full name of the month. The following statements display the name of the current month (both the abbreviation and the full name). Every time you execute these statements, you will see the current month's name in the current language:

```
Dim mName As String
mName = MonthName(Now.Month, True)
MsgBox(mName)   ' prints "Jan"
mName = MonthName(Now.Month, False)
MsgBox(mName)   ' prints "January"
```

A similar function, the `WeekDayName()` function, returns the name of the week for a specific weekday. This function accepts an additional argument that determines the first day of the week. (See the documentation for more information on the syntax of the `WeekDayName()` function.)

The primary role of functions is to extend the functionality of the language. Many functions that perform rather common practical operations have been included in the language, but they aren't nearly enough for the needs of all developers or all types of applications. Besides the built-in functions, you can write custom functions to simplify the development of your custom applications, as explained in the following section.

Custom Functions

Most of the code we write is in the form of custom functions or subroutines that are called from several places in the application. Subroutines are just like functions, except that they don't return a value, so we'll focus on the implementation of custom functions. With the exception of a function's return value, everything else presented in this and the following section applies to subroutines as well.

Let's look at an example of a fairly simple (but not trivial) function that does something really useful. Books are identified by a unique international standard book number (ISBN), and every application that manages books needs a function to verify the ISBN, which is made up of 12 digits followed by a check digit. To calculate the check digit, you multiply each of the 12 digits by a constant; the first digit is multiplied by 1, the second digit is multiplied by 3, the third digit by 1 again, and so on. The sum of these multiplications is then divided by 10, and we take the remainder. The check digit is this remainder subtracted from 10. To calculate the check digit for the ISBN 978078212283, compute the sum of the following products:

```
9 * 1 + 7 * 3 + 8 * 1 + 0 * 3 + 7 * 1 + 8 * 3 +
2 * 1 + 1 * 3 + 2 * 1 + 2 * 3 + 8 * 1 + 3 * 3 = 99
```

The sum is 99; when you divide it by 10, the remainder is 9. The check digit is 10 – 9, or 1, and the book's complete ISBN is 9780782122831. The `ISBNCheckDigit()` function, shown in Listing 3.9, accepts the 12 digits of the ISBN as an argument and returns the appropriate check digit.

Listing 3.9: The *ISBNCheckDigit()* Custom Function

```
Function ISBNCheckDigit(ByVal ISBN As String) As String
    Dim i As Integer, chksum As Integer = 0
    Dim chkDigit As Integer
    Dim factor As Integer = 3
    For i = 0 To 11
        factor = 4 - factor
        chksum += factor * Convert.ToInt16(ISBN.Substring(i, 1))
    Next
    Return (((10 - (chksum Mod 10)) Mod 10)).ToString
End Function
```

The `ISBNCheckDigit()` function returns a string value because ISBNs are handled as strings, not numbers. (Leading zeros are important in an ISBN but are totally meaningless, and omitted, in a numeric value.) The `Substring` method of a String object extracts a number of characters from the string to which it's applied. The first argument is the starting location in the string, and the second is the number of characters to be extracted. The expression `ISBN.Substring(i, 1)` extracts one character at a time from the *`ISBN`* string variable. During the first iteration of the loop, it extracts the first character; during the second iteration, it extracts the second character, and so on.

The extracted character is a numeric digit, which is multiplied by the *`factor`* variable value and the result is added to the *`chkSum`* variable. This variable is the checksum of the ISBN. After it has been calculated, we divide it by 10 and take its remainder (the first `Mod` operator returns the remainder of this division), which we subtract from 10. The second `Mod` operator maps the value 10 to 0. This is the ISBN's check digit and the function's return value.

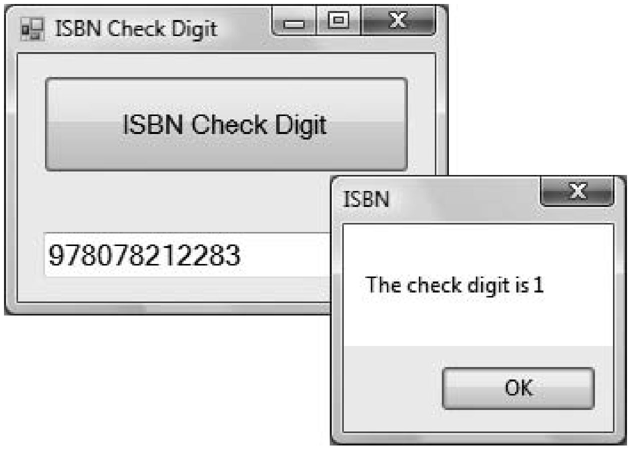
You can use this function in an application that maintains a book database to make sure that all books are entered with a valid ISBN. You can also use it with a web application that allows viewers to request books by their ISBN. The same code will work with two different applications, even when passed to other developers. Developers using your function don't have to know how the check digit is calculated, just how to call the function and retrieve its result.

To test the `ISBNCheckDigit()` function, start a new project, place a button on the form, and enter the following statements in its `Click` event handler (or open the ISBN project in the folder with this chapter's sample projects):

```
Private Sub Button1_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles Button1.Click
  Console.WriteLine("The check Digit is " & _
                 ISBNCheckDigit("978078212283"))
End Sub
```

After inserting the code of the `ISBNCheckDigit()` function and the code that calls the function, your code editor should look like Figure 3.1. You can place a TextBox control on the form and pass the `Text` property of the control to the `ISBNCheckDigit()` function to calculate the check digit.

A similar algorithm is used for calculating the check digit of credit cards: the Luhns algorithm. You can look it up on the Internet and write a custom function for validating credit card numbers.



978078212283

The check digit is 1

FIGURE 3.1
Calling the `ISBNCheckDigit()` function

Passing Arguments and Returning Values

So far you've learned how to write and call procedures with a few simple arguments and how to retrieve the function's return value and use it in your code. This section covers a few advanced topics on argument-passing techniques and how to write functions that return multiple values, or arrays of values and custom data types.

Passing an Unknown Number of Arguments

Generally, all the arguments that a procedure expects are listed in the procedure's definition, and the program that calls the procedure must supply values for all arguments. On occasion, however, you might not know how many arguments will be passed to the procedure. Procedures that calculate averages or, in general, process multiple values can accept from a few to several arguments whose count is not known at design time. VB 2008 supports the `ParamArray` keyword, which allows you to pass a variable number of arguments to a procedure.

Let's look at an example. Suppose that you want to populate a ListBox control with elements. To add an item to the ListBox control, you call the `Add` method of its `Items` collection as follows:

```
ListBox1.Items.Add("new item")
```

This statement adds the string `new item` to the *`ListBox1`* control. If you frequently add multiple items to a `ListBox` control from within your code, you can write a subroutine that performs this task. The following subroutine adds a variable number of arguments to the *`ListBox1`* control:

```
Sub AddNamesToList(ByVal ParamArray NamesArray() As Object)
   Dim x As Object
   For Each x In NamesArray
      ListBox1.Items.Add(x)
   Next x
End Sub
```

This subroutine's argument is an array prefixed with the keyword `ParamArray`, which holds all the parameters passed to the subroutine. If the parameter array holds items of the same type, you can declare the array to be of the specific type (string, integer, and so on). To add items to the list, call the `AddNamesToList()` subroutine as follows:

```
AddNamesToList("Robert", "Manny", "Renee", "Charles", "Madonna")
```

If you want to know the number of arguments actually passed to the procedure, use the `Length` property of the parameter array. The number of arguments passed to the `AddNamesToList()` subroutine is given by the following expression:

```
NamesArray.Length
```

The following loop goes through all the elements of the *`NamesArray`* and adds them to the list:

```
Dim i As Integer
For i = 0 to NamesArray.GetUpperBound(0)
   ListBox1.Items.Add(NamesArray(i))
Next i
```

VB arrays are zero-based (the index of the first item is 0), and the `GetUpperBound` method returns the index of the last item in the array.

A procedure that accepts multiple arguments relies on the order of the arguments. To omit some of the arguments, you must use the corresponding comma. Let's say you want to call such a procedure and specify the first, third, and fourth arguments. The procedure must be called as follows:

```
ProcName(arg1, , arg3, arg4)
```

The arguments to similar procedures are usually of equal stature, and their order doesn't make any difference. A function that calculates the mean or other basic statistics of a set of numbers, or a subroutine that populates a `ListBox` or `ComboBox` control, are prime candidates for implementing this technique. If the procedure accepts a variable number of arguments that aren't equal in stature, you should consider the technique described in the following section. If the function accepts a parameter array, this must the last argument in the list, and none of the other parameters can be optional.

Named Arguments

You learned how to write procedures with optional arguments and how to pass a variable number of arguments to the procedure. The main limitation of the argument-passing mechanism, though, is the *order* of the arguments. By default, Visual Basic matches the values passed to a procedure to the declared arguments by their order (which is why the arguments you've seen so far are called *positional arguments*).

This limitation is lifted by Visual Basic's capability to specify *named arguments*. With named arguments, you can supply arguments in any order because they are recognized by name and not by their order in the list of the procedure's arguments. Suppose you've written a function that expects three arguments: a name, an address, and an email address:

```
Function Contact(Name As String, Address As String, EMail As String)
```

When calling this function, you must supply three strings that correspond to the arguments *`Name`*, *`Address`*, and *`EMail`*, in that order. However, there's a safer way to call this function: Supply the arguments in any order by their names. Instead of calling the `Contact()` function as follows:

```
Contact("Peter Evans", "2020 Palm Ave., Santa Barbara, CA 90000", _
        "PeterEvans@example.com")
```

you can call it this way:

```
Contact(Address:="2020 Palm Ave., Santa Barbara, CA 90000", _
        EMail:="PeterEvans@example.com", Name:="Peter Evans")
```

The `:=` operator assigns values to the named arguments. Because the arguments are passed by name, you can supply them in any order.

To test this technique, enter the following function declaration in a form's code:

```
Function Contact(ByVal Name As String, ByVal Address As String, _
                 ByVal EMail As String) As String
   Debug.WriteLine(Name)
```

```
    Debug.WriteLine(Address)
    Debug.WriteLine(EMail)
    Return ("OK")
End Function
```

Then call the `Contact()` function from within a button's `Click` event with the following statement:

```
Debug.WriteLine( _
        Contact(Address:="2020 Palm Ave., Santa Barbara, CA 90000", _
        Name:="Peter Evans", EMail:="PeterEvans@example.com"))
```

You'll see the following in the Immediate window:

```
Peter Evans
2020 Palm Ave., Santa Barbara, CA 90000
PeterEvans@example.com
OK
```

The function knows which value corresponds to which argument and can process them the same way that it processes positional arguments. Notice that the function's definition is the same, whether you call it with positional or named arguments. The difference is in how you call the function and not how you declare it.

Named arguments make code safer and easier to read, but because they require a lot of typing, most programmers don't use them. Besides, when IntelliSense is on, you can see the definition of the function as you enter the arguments, and this minimizes the chances of swapping two values by mistake.

More Types of Function Return Values

Functions are not limited to returning simple data types such as integers or strings. They might return custom data types and even arrays. The capability of functions to return all types of data makes them very flexible and can simplify coding, so we'll explore it in detail in the following sections. Using complex data types, such as structures and arrays, allows you to write functions that return multiple values.

Functions Returning Structures

Suppose you need a function that returns a customer's savings and checking account balances. So far, you've learned that you can return two or more values from a function by supplying arguments with the `ByRef` keyword. A more elegant method is to create a custom data type (a structure) and write a function that returns a variable of this type.

Here's a simple example of a function that returns a custom data type. This example outlines the steps you must repeat every time you want to create functions that return custom data types:

1. Create a new project and insert the declarations of a custom data type in the declarations section of the form:

    ```
    Structure CustBalance
       Dim SavingsBalance As Decimal
       Dim CheckingBalance As Decimal
    End Structure
    ```

2. Implement the function that returns a value of the custom type. In the function's body, you must declare a variable of the type returned by the function and assign the proper values to its fields. The following function assigns random values to the fields `CheckingBalance` and `SavingsBalance`. Then assign the variable to the function's name, as shown next:

```
Function GetCustBalance(ID As Long) As CustBalance
   Dim tBalance As CustBalance
   tBalance.CheckingBalance = CDec(1000 + 4000 * rnd())
   tBalance.SavingsBalance = CDec(1000 + 15000 * rnd())
   Return(tBalance)
End Function
```

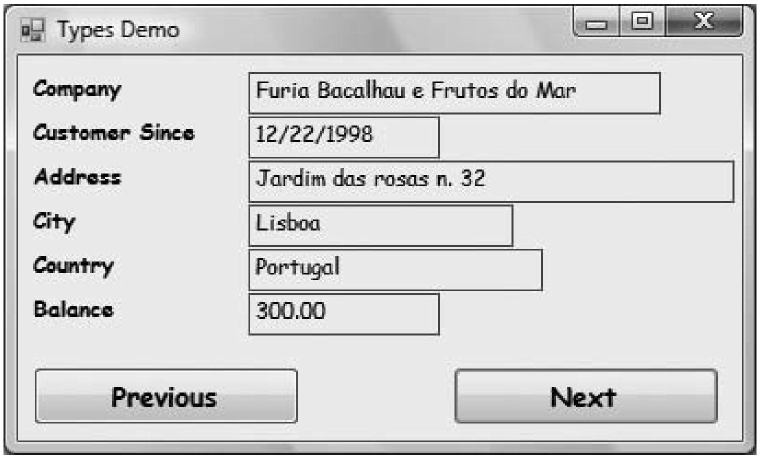
3. Place a button on the form from which you want to call the function. Declare a variable of the same type and assign to it the function's return value. The example that follows prints the savings and checking balances in the Output window:

```
Private Sub Button1_Click(...) Handles Button1.Click
    Dim balance As CustBalance
    balance = GetCustBalance(1)
    Debug.WriteLine(balance.CheckingBalance)
    Debug.WriteLine(balance.SavingsBalance)
End Sub
```

The code shown in this section belongs to the Structures sample project. Create this project from scratch, perhaps by using your own custom data type, to explore its structure and experiment with functions that return custom data types. In Chapter 10, ''Building Custom Classes,'' you'll learn how to build your own classes and you'll see how to write functions that return custom objects.

VB 2008 at Work: The Types Project

The Types project, which you'll find in this chapter's folder, demonstrates a function that returns a custom data type. The Types project consists of a form that displays record fields (see Figure 3.2).



Company	Furia Bacalhau e Frutos do Mar
Customer Since	12/22/1998
Address	Jardim das rosas n. 32
City	Lisboa
Country	Portugal
Balance	300.00

FIGURE 3.2 The Types project demonstrates functions that return custom data types.

Every time you click the Next button, the fields of the next record are displayed in the corresponding TextBox controls on the form. When all records are exhausted, the program wraps back to the first record.

The project consists of a single form and uses a custom data type, implemented with the following structure. The structure's declaration must appear in the form's code, outside any procedure, along with a couple of variable declarations:

```
Structure Customer
   Dim Company As String
   Dim Manager As String
   Dim Address As String
   Dim City As String
   Dim Country As String
   Dim CustomerSince As Date
   Dim Balance As Decimal
End Structure
Private Customers(9) As Customer
Private cust As Customer
Private currentIndex as Integer
```

The array *`Customers`* holds the data for 10 customers, and the `cust` variable is used as a temporary variable for storing the current customer's data. The *`currentIndex`* variable is the index of the current element of the array. The array is filled with `Customer` data, and the *`currentIndex`* variable is initialized to zero.

The `Click` event handler of the Next button calls the `GetCustomer()` function with an index value (which is the order of the current customer) to retrieve the data of the next customer, and displays the customer's fields on the Label controls on the form with the `ShowCustomer()` subroutine. Then it increases the value of the *`currentIndex`* variable to point to the current customer's index. You can open the Types project in Visual Studio and examine its code, which contains quite a few comments explaining its operation.

Functions Returning Arrays

In addition to returning custom data types, VB 2008 functions can also return arrays. This is an interesting possibility that allows you to write functions that return not only multiple values, but also an unknown number of values.

In this section, we'll write the `Statistics()` function, similar to the `CalculateStatistics()` function you saw a little earlier in this chapter. The `Statistics()` function returns the statistics in an array. Moreover, it returns not only the average and the standard deviation, but the minimum and maximum values in the data set as well. One way to declare a function that calculates all the statistics is as follows:

```
Function Statistics(ByRef DataArray() As Double) As Double()
```

This function accepts an array with the data values and returns an array of Doubles. To implement a function that returns an array, you must do the following:

1. Specify a type for the function's return value and add a pair of parentheses after the type's name. Don't specify the dimensions of the array to be returned here; the array will be declared formally in the function.

2. In the function's code, declare an array of the same type and specify its dimensions. If the function should return four values, use a declaration like this one:

```
Dim Results(3) As Double
```

The *Results* array, which will be used to store the results, must be of the same type as the function — its name can be anything.

3. To return the Results array, simply use it as an argument to the Return statement:

```
Return(Results)
```

4. In the calling procedure, you must declare an array of the same type without dimensions:

```
Dim Statistics() As Double
```

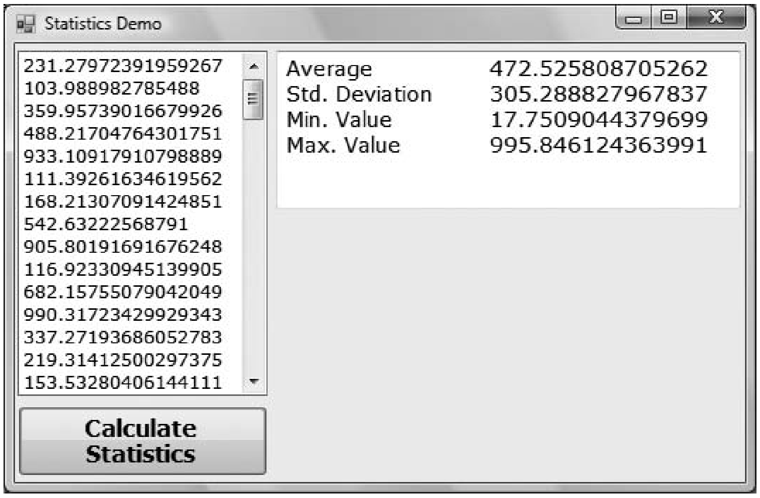
5. Finally, you must call the function and assign its return value to this array:

```
Stats() = Statistics(DataSet())
```

Here, *DataSet* is an array with the values whose basic statistics will be calculated by the Statistics() function. Your code can then retrieve each element of the array with an index value as usual.

VB 2008 AT WORK: THE STATISTICS PROJECT

The Statistics sample project demonstrates how to write a procedure that returns an array. When you run it, the Statistics application creates a data set of random values and then calls the Statistics() function to calculate the data set's basic statistics. The results are returned in an array, and the main program displays them in Label controls, as shown in Figure 3.3. Every time the Calculate Statistics button is clicked, a new data set is generated and its statistics are displayed.



```
231.27972391959267
103.988982785488
359.95739016679926
488.21704764301751
933.10917910798889
111.39261634619562
168.21307091424851
542.63222568791
905.80191691676248
116.92330945139905
682.15755079042049
990.31723429929343
337.27193686052783
219.31412500297375
153.53280406144111
```

```
Average         472.525808705262
Std. Deviation  305.288827967837
Min. Value      17.7509044379699
Max. Value      995.846124363991
```

FIGURE 3.3 The Statistics project calculates the basic statistics of a data set and returns them in an array.

The `Statistics()` function's code is based on the preceding discussion, and I will not show it here. You can open the Statistics project and examine the function's code, as well as how the main program uses the array returned by the `Statistics()` function.

Overloading Functions

There are situations in which the same function must operate on different data types or a different number of arguments. In the past, you had to write different functions, with different names and different arguments, to accommodate similar requirements. The Framework introduced the concept of *function overloading*, which means that you can have multiple implementations of the same function, each with a different set of arguments and possibly a different return value. Yet all overloaded functions share the same name. Let me introduce this concept by examining one of the many overloaded functions that come with the `.NET` Framework.

The `Next` method of the System.Random class returns an integer value from –2,147,483,648 to 2,147,483,647. (This is the range of values that can be represented by the Integer data type.) We should also be able to generate random numbers in a limited range of integer values. To emulate the throw of a die, we want a random value in the range from 1 to 6, whereas for a roulette game we want an integer random value in the range from 0 to 36. You can specify an upper limit for the random number with an optional integer argument. The following statement will return a random integer in the range from 0 to 99:

```
randomInt = rnd.Next(100)
```

You can also specify both the lower and upper limits of the random number's range. The following statement will return a random integer in the range from 1,000 to 1,999:

```
randomInt = rnd.Next(1000, 2000)
```

The same method behaves differently based on the arguments we supply. The behavior of the method depends either on the type of the arguments, the number of the arguments, or both. As you will see, there's no single function that alters its behavior based on its arguments. There are as many different implementations of the same function as there are argument combinations. All the functions share the same name, so they appear to the user as a single multifaceted function. These functions are overloaded, and you'll see how they're implemented in the following section.

If you haven't turned off the IntelliSense feature of the editor, as soon as you type the opening parenthesis after a function or method name, you see a yellow box with the syntax of the function or method. You'll know that a function, or a method, is overloaded when this box contains a number and two arrows. Each number corresponds to a different overloaded form, and you can move to the next or previous overloaded form by clicking the two little arrows or by pressing the arrow keys.

Let's return to the `Min()` function we implemented earlier in this chapter. The initial implementation of the `Min()` function is shown next:

```
Function Min(ByVal a As Double, ByVal b As Double) As Double
   Min = IIf(a < b, a, b)
End Function
```

By accepting Double values as arguments, this function can handle all numeric types. VB 2008 performs automatic widening conversions (it can convert Integers and Decimals to Doubles), so this trick makes the function work with all numeric data types. However, what about strings? If you attempt to call the `Min()` function with two strings as arguments, you'll get an exception. The `Min()` function just can't handle strings.

To write a `Min()` function that can handle both numeric and string values, you must, in essence, write two `Min()` functions. All `Min()` functions must be prefixed with the `Overloads` keyword. The following statements show two different implementations of the same function:

```
Overloads Function Min(ByVal a As Double, ByVal b As Double) As Double
   Min = Convert.ToDouble(IIf(a < b, a, b))
End Function

Overloads Function Min(ByVal a As String, ByVal b As String) As String
   Min = Convert.ToString(IIf(a < b, a, b))
End Function
```

We need a third overloaded form of the same function to compare dates. If you call the `Min()` function, passing as an argument two dates, as in the following statement, the `Min()` function will compare them as strings and return (incorrectly) the first date.

```
Debug.WriteLine(Min(#1/1/2009#, #3/4/2008#))
```

This statement is not even valid when the Strict option is on, so you clearly need another overloaded form of the function that accepts two dates as arguments, as shown here:

```
Overloads Function Min(ByVal a As Date, ByVal b As Date) As Date
   Min = IIf(a < b, a, b)
End Function
```

If you now call the `Min()` function with the dates `#1/1/2009#` and `#3/4/2008#`, the function will return the second date, which is chronologically smaller than the first.

Let's look into a more complicated overloaded function, which makes use of some topics discussed later in this book. The `CountFiles()` function counts the number of files in a folder that meet certain criteria. The criteria could be the size of the files, their type, or the date they were created. You can come up with any combination of these criteria, but the following are the most useful combinations. (These are the functions I would use, but you can create even more combinations or introduce new criteria of your own.) The names of the arguments are self-descriptive, so I need not explain what each form of the `CountFiles()` function does.

```
CountFiles(ByVal minSize As Integer, ByVal maxSize As Integer) As Integer
CountFiles(ByVal fromDate As Date, ByVal toDate As Date) As Integer
CountFiles(ByVal type As String) As Integer
CountFiles(ByVal minSize As Integer, ByVal maxSize As Integer, _
           ByVal type As String) As Integer
CountFiles(ByVal fromDate As Date, ByVal toDate As Date, _
           ByVal type As String) As Integer
```

Listing 3.10 shows the implementation of these overloaded forms of the `CountFiles()` function. (I'm not showing all overloaded forms of the function; you can open the OverloadedFunctions project in the IDE and examine the code.) Because we haven't discussed file operations yet, most of the code in the function's body will be new to you — but it's not hard to follow. For the benefit of readers who are totally unfamiliar with file operations, I included a statement that prints in the Immediate window the type of files counted by each function. The `Debug.WriteLine` statement prints the values of the arguments passed to the function, along with a description of the type of search it will perform. The overloaded form that accepts two integer values as arguments prints something like this:

```
You've requested the files between 1000 and 100000 bytes
```

whereas the overloaded form that accepts a string as an argument prints the following:

```
You've requested the .EXE files
```

LISTING 3.10: The Overloaded Implementations of the *CountFiles()* Function

```
Overloads Function CountFiles( _
                   ByVal minSize As Integer, _
                   ByVal maxSize As Integer) As Integer
   Debug.WriteLine("You've requested the files between " & _
           minSize &  " and " & maxSize & " bytes")
   Dim files() As String
   files = System.IO.Directory.GetFiles("c:\windows")
   Dim i, fileCount As Integer
   For i = 0 To files.GetUpperBound(0)
      Dim FI As New System.IO.FileInfo(files(i))
      If FI.Length >= minSize And FI.Length <= maxSize Then
         fileCount = fileCount + 1
      End If
   Next
   Return(fileCount)
End Function

Overloads Function CountFiles( _
                   ByVal fromDate As Date, _
                   ByVal toDate As Date) As Integer
   Debug.WriteLine("You've requested the count of files created from " & _
                     fromDate & " to " & toDate)
   Dim files() As String
   files = System.IO.Directory.GetFiles("c:\windows")
   Dim i, fileCount As Integer
   For i = 0 To files.GetUpperBound(0)
      Dim FI As New System.IO.FileInfo(files(i))
      If FI.CreationTime.Date >= fromDate And _
           FI.CreationTime.Date <= toDate Then
         fileCount = fileCount + 1
      End If
   Next
   Return(fileCount)
End Function
```

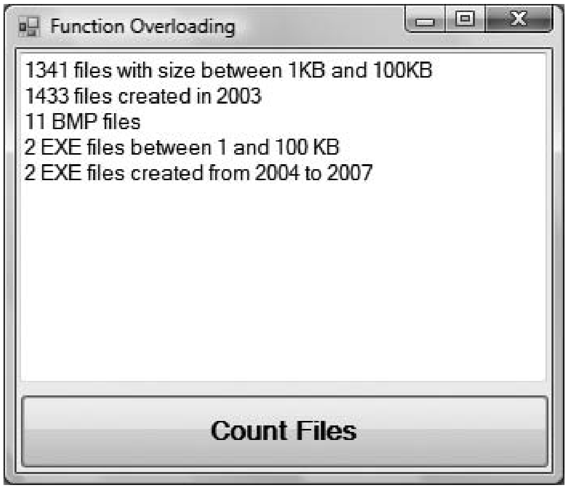
```
Overloads Function CountFiles(ByVal type As String) As Integer
   Debug.WriteLine("You've requested the " & type & " files")
   ' Function Implementation

End Function

Overloads Function CountFiles(_
                  ByVal minSize As Integer, _
                  ByVal maxSize As Integer, _
                  ByVal type As String) As Integer
   Debug.WriteLine("You've requested the " & type & _
                  " files between " & minSize & " and " & _
                  maxSize & " bytes")
   ' Function implementation
End Function

Overloads Function CountFiles(ByVal fromDate As Date, _
                  ByVal toDate As Date, ByVal type As String) As Integer
   Debug.WriteLine("You've requested the " & type & _
                  " files created from " & fromDate & " to " & toDate)
   ' Function implementation
End Function
```

If you're unfamiliar with the Directory and File objects, focus on the statement that prints to the Immediate window and ignore the statements that actually count the files that meet the specified criteria. After reading Chapter 15, ''Accessing Folders and Files,'' you can revisit this example and understand the statements that select the qualifying files and count them.

Start a new project and enter the definitions of the overloaded forms of the function on the form's level. Listing 3.10 is lengthy, but all the overloaded functions have the same structure and differ only in how they select the files to count. Then place a TextBox and a button on the form, as shown in Figure 3.4, and enter a few statements that exercise the various overloaded forms of the function (such as the ones shown in Listing 3.11) in the button's `Click` event handler.

Listing 3.11: Testing the Overloaded Forms of the *CountFiles()* Function

```
Private Sub Button1_Click(...) Handles Button1.Click
   TextBox1.AppendText(CountFiles(1000, 100000) & _
               " files with size between 1KB and 100KB" & vbCrLf)
   TextBox1.AppendText(CountFiles(#1/1/2006#, #12/31/2006#) & _
               " files created in 2006" & vbCrLf)
   TextBox1.AppendText(CountFiles(".BMP") & " BMP files" & vbCrLf)
   TextBox1.AppendText(CountFiles(1000, 100000, ".EXE") & _
               " EXE files between 1 and 100 KB" & vbCrLf)
   TextBox1.AppendText(CountFiles(#1/1/2006#, #12/31/2007#, ".EXE") & _
               " EXE files created in 2006 and 2007")
End Sub
```

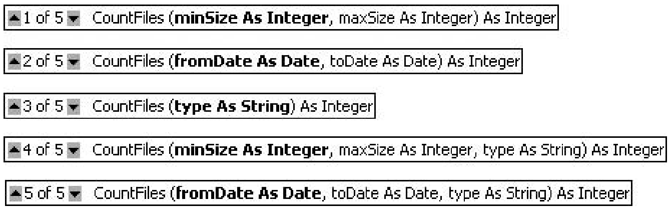

Function Overloading

```
1341 files with size between 1KB and 100KB
1433 files created in 2003
11 BMP files
2 EXE files between 1 and 100 KB
2 EXE files created from 2004 to 2007
```

FIGURE 3.4 The Overloaded Functions project

The button calls the various overloaded forms of the `CountFiles()` function one after the other and prints the results on the TextBox control. From now on, I'll be omitting the list of arguments in the most common event handlers, such as the `Click` event handler, because they're always the same and they don't add to the readability of the code. In place of the two arguments, I'll insert an ellipsis to indicate the lack of the arguments.

Function overloading is used heavily throughout the language. There are relatively few functions (or methods, for that matter) that aren't overloaded. Every time you enter the name of a function followed by an opening parenthesis, a list of its arguments appears in the drop-down list with the arguments of the function. If the function is overloaded, you'll see a number in front of the list of arguments, as shown in Figure 3.5. This number is the order of the overloaded form of the function, and it's followed by the arguments of the specific form of the function. The figure shows all the forms of the `CountFiles()` function.



```
1 of 5  CountFiles (minSize As Integer, maxSize As Integer) As Integer
2 of 5  CountFiles (fromDate As Date, toDate As Date) As Integer
3 of 5  CountFiles (type As String) As Integer
4 of 5  CountFiles (minSize As Integer, maxSize As Integer, type As String) As Integer
5 of 5  CountFiles (fromDate As Date, toDate As Date, type As String) As Integer
```

FIGURE 3.5 The overloaded forms of the `CountFiles()` function

The Bottom Line

Use Visual Basic's flow-control statements. Visual Basic provides several statements for controlling the sequence in which statements are executed: decision statements, which change the course of execution based on the outcome of a comparison, and loop statements, which repeat a number of statements while a condition is true or false.

Master It Explain briefly the decision statements of Visual Basic.

Write subroutines and functions. To manage large applications, we break our code into small, manageable units. These units of code are the subroutines and functions. Subroutines perform actions and don't return any values. Functions, on the other hand, perform calculations and return values. Most of the language's built-in functionality is in the form of functions.

Master It How will you create multiple overloaded forms of the same function?

Pass arguments to subroutines and functions. Procedures and functions communicate with one another via arguments, which are listed in a pair of parentheses following the procedure's name. Each argument has a name and a type. When you call the procedure, you must supply values for each argument and the types of the values should match the types listed in the procedure's definition.

Master It Explain the difference between passing arguments by value and passing arguments by reference.

Chapter 4

GUI Design and Event-Driven Programming

The first three chapters of this book introduced you to the basics of designing applications with Visual Studio 2008 and the components of the Visual Basic language. You know how to design graphical user interfaces (GUI) and how to use the statements of Visual Basic to program the events of the various controls. You also know how to write functions and subroutines and how to call the built-in functions and subroutines of Visual Basic.

In this chapter, you'll design a few more Windows applications — this time, a few practical applications with more functional interfaces and a bit of code that does something more practical. You'll put together the information presented so far in the book by building Windows applications with the visual tools of Visual Studio and you'll see how the application interacts with users by coding the events of interest. If you are new to Visual Studio, you should design the examples on your own using the instructions in the text, rather than open the same projects and look at the code.

In this chapter, you will learn how to do the following:

- Design graphical user interfaces
- Program events
- Write robust applications with error handling

On Designing Windows Applications

As you recall from Chapter 1, ''Getting Started with Visual Basic 2008,'' the design of a Windows application consists of two distinct phases: the design of the application's interface and the coding of the application. The design of the interface is performed with visual tools and consists of creating a form with the relevant elements. These elements are the building blocks of Windows applications and are called *controls*.

The available controls are shown in the Toolbox and are the same elements used by all Windows applications. In addition to being visually rich, the controls embed a lot of functionality. The TextBox control, for example, can handle text on its own, without any programming effort on your part. The ComboBox control expands the list with its items when users click the arrow button and displays the selected item in its edit box. In general, the basic functionality of the controls is built into the controls by design, so that all applications maintain a consistent look.

The interface dictates how users will interact with your application. To prompt users for text or numeric data, use TextBox controls. When it comes to specifying one or more of several options, you have many choices: You can use a ComboBox control from which users can select an option, or a few CheckBox controls on the form that users can select or clear. If you want to display a small number of mutually exclusive options, place a few RadioButton controls on the form. Every time the user selects an option, the previously selected one is cleared. To initiate actions, place one or more Button controls on the form.

Controls expose a large number of properties, which are displayed in the Properties window at design time. You use these properties to adjust not only the appearance of the controls on the form, but their functionality as well. The process of designing the interface consists mostly of setting the properties of the various controls.

An important aspect of the design of your application's user interface is the alignment of the controls on the form. Controls that are next to one another should be aligned horizontally. Controls that are stacked should have either their left or right edges aligned vertically. You should also make sure that the controls are spaced equally. The integrated development environment (IDE) provides all the tools for sizing, aligning, and spacing controls on the form, and you'll see these tools in action through examples in this chapter.

After you have designed the interface, you know how your application will interact with the user. The next step is to actually implement the interaction by writing some code. The programming model of Visual Basic is event-driven: As the user interacts with the controls on your form, some code is executed in response to user actions. The user's actions cause events, and each control recognizes its own set of events and handles them through subroutines, which are called *event handlers*. When users click a button, the control's `Click` event is fired, and you must insert the relevant code in the control's `Click` event handler. The event-driven programming model has proven very successful, because it allows developers to focus on handling specific actions. It allows you to break a large application into smaller, manageable units of code and implement each unit of code independently of any other.

Developing Windows applications is a conceptually simple process, but there's a methodology to it and it's not trivial. Fortunately, the IDE provides many tools to simplify the process; it will even catch most of the errors in your code as you type. You have seen how to use some of the tools of the IDE in the first three chapters. In this chapter, I'll present these tools through examples.

Building a Loan Calculator

One easy-to-implement, practical application is a program that calculates loan parameters. Visual Basic provides built-in functions for performing many types of financial calculations, and you need only a single line of code to calculate the monthly payment given the loan amount, its duration, and the interest rate. Designing the user interface, however, takes much more effort.

Regardless of the language you use, you must go through the following process to develop an application:

1. Decide what the application will do and how it will interact with the user.
2. Design the application's user interface according to the requirements of step 1.
3. Write the actual code behind the events you want to handle.

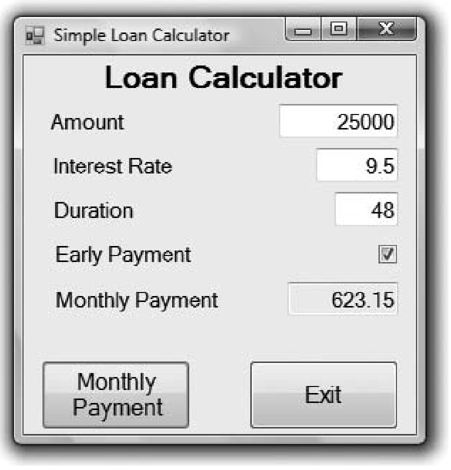
Understanding How the Loan Calculator Application Works

Following the first step of the process outlined previously, you decide that the user should be able to specify the amount of the loan, the interest rate, and the duration of the loan in months. You must, therefore, provide three text boxes in which the user can enter these values.

Another parameter affecting the monthly payment is whether payments are made at the beginning or at the end of each month, so you must also provide a way for the user to specify whether the payments will be early (first day of the month) or late (last day of the month). The most appropriate type of control for entering Yes/No or True/False type of information is the CheckBox control. This control is a toggle: If it's selected, you can clear it by clicking it; if it's cleared, you can select it by clicking again. The user doesn't enter any data in this control (which means you need not anticipate user errors with this control), and it's the simplest method for specifying values with two possible states.

Figure 4.1 shows a user interface that matches our design specifications. This is the main form of the LoanCalculator project, which you will find in this chapter's folder on the book's project download site.

FIGURE 4.1
LoanCalculator is a simple financial application.



Amount	25000
Interest Rate	9.5
Duration	48
Early Payment	✓
Monthly Payment	623.15

The user enters all the information on the form and then clicks the Monthly Payment button to calculate the monthly payment. The program will calculate the monthly payment and display it in the lower TextBox control. All the action takes place in the button's `Click` subroutine.

To calculate the monthly payments on a loan, we call the `Pmt ()` built-in function, whose syntax is the following:

```
MonthlyPayment = Pmt(InterestRate, Periods, Amount, FutureValue, Due)
```

The interest rate, argument *`InterestRate`*, is specified as a monthly rate. If the yearly interest rate is 16.5 percent, the value entered by the user in the Interest Rate box should be 14.5, and the monthly rate will be 0.145/12. The duration of the loan, the *`Periods`* argument, is specified in number of months, and the *`Amount`* argument is the loan's amount. The *`FutureValue`* argument is the value of the loan at the end of the period, which should be zero (it would be a positive value for an investment), and the last argument, *`Due`*, specifies when payments are due. The value of *`Due`* can be one of the constants DueDate.BegOfPeriod and DueDate.EndOfPeriod. These two constants are built into the language, and you can use them without knowing their exact value.

The present value of the loan is the amount of the loan with a negative sign. It's negative because you don't have the money now. You're borrowing it — it is money you owe to the bank. Future value represents the value of something at a stated time — in this case, what the loan will be worth when it's paid off. This is what one side owes the other at the end of the specified period. So the future value of a loan is zero.

You don't need to know how the `Pmt ()` function calculates the monthly payment, just how to call it and how to retrieve the results. To calculate the monthly payment on a loan of $25,000 with an interest rate of 14.5 percent, payable over 48 months, and payments due the last day of the payment period (which in our case is a month), you'd call the `Pmt()` function as follows:

```
Pmt(0.145 / 12, 48, -25000, 0, DueDate.EndOfPeriod)
```

The `Pmt()` function will return the value 689.448821287218. Because it's a dollar amount, we must round it to two decimal digits on our interface. Notice the negative sign in front of the *`Amount`* argument in the statement. If you specify a positive amount, the result will be a negative payment. The payment and the loan's amount have different signs because they represent different cash flows. The loan's amount is money you *owe* to the bank, whereas the payment is money you *pa*y to the bank.

The last two arguments of the `Pmt()` function are optional. If you omit them, Visual Basic uses their default values, which are 0 for the *`FutureValue`* argument and *`DueDate.BegOfPeriod`* for the *`Due`* argument. You can entirely omit these arguments and call the `Pmt()` function like this:

```
Pmt(0.145 / 12, 48, -25000)
```

Calculating the amount of the monthly payment given the loan parameters is quite simple. What you need to understand are the parameters of a loan and how to pass them to the `Pmt()` function. You must also know how the interest rate is specified to avoid invalid values. Although the calculation of the payment is trivial, designing the interface will take a bit of effort.

Designing the User Interface

Now that you know how to calculate the monthly payment, you can design the user interface. To do so, start a new project, name it *`LoanCalculator`*, and rename its form to *`frmLoan`*. Your first task is to decide the font and size of the text you'll use for the controls on the form. The form is the container of the controls, and they inherit some of the form's properties, such as the font. You can change the font later during the design, but it's a good idea to start with the right font. At any rate, don't try to align the controls if you're planning to change their fonts. The change will, most likely, throw off your alignment efforts.

The book's sample project uses the 10-point Verdana font. To change it, select the form with the mouse, double-click the name of the `Font` property in the Properties window to open the Font dialog box, and select the desired font and attributes. I'm using the Verdana and Seago fonts a lot because they're clean and they were designed for viewing on monitors. Of course, this is a personal choice. Avoid elaborate fonts and don't mix different fonts on the same form (or in different forms of the same application).

To design the form shown in Figure 4.1, follow these steps:

1. Place four labels on the form and assign the following captions (the `Text` property of each control) to them:

Name	Text
`Label1`	Amount
`Label2`	Duration
`Label3`	Interest Rate
`Label4`	Monthly Payment

   You don't need to change the default names of the four Label controls on the form because their captions are all we need. You aren't going to program them.

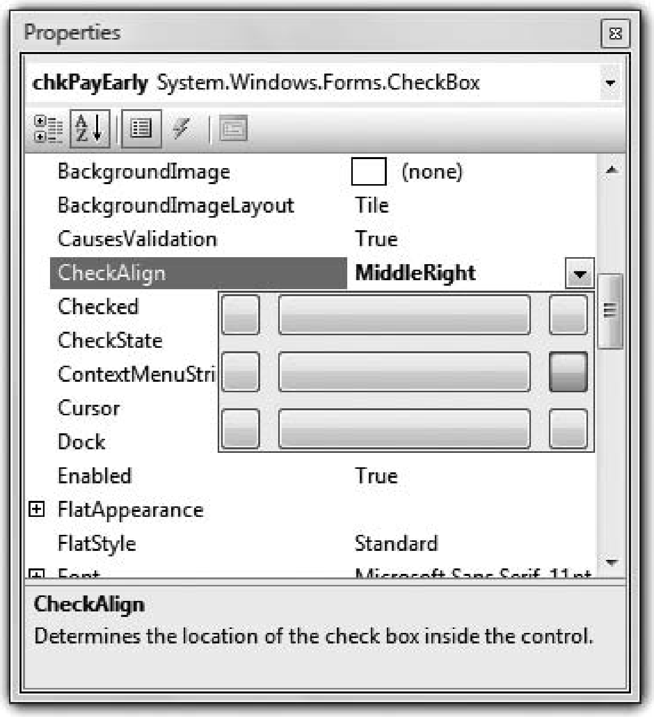
2. Place a TextBox control next to each label. Set their `Name` and `Text` properties to the following values. I used meaningful names for the TextBox controls because we'll use them in our code shortly to retrieve the values entered by the user on these controls. These initial values correspond to a loan of $25,000 with an interest rate of 14.5 percent and a payoff period of 48 months.

Name	Text
`txtAmount`	25000
`txtDuration`	48
`txtRate`	14.5
`txtPayment`	

3. The fourth TextBox control is where the monthly payment will appear. The user isn't supposed to enter any data in this box, so you must set its `ReadOnly` property to True to lock the control. You'll be able to change its value from within your code, but users won't be able to type anything in it. (We could have used a Label control instead, but the uniform look of TextBoxes on a form is usually preferred.) You will also notice that the TextBox controls have a 3D frame. Experiment with the control's `BorderStyle` property to discover the available styles for the control's frame (I've used the *`Fixed3D`* setting for the TextBox controls).

4. Next, place a CheckBox control on the form. By default, the control's caption is CheckBox1, and it appears to the right of the check box. Because we want the titles to be to the left of the corresponding controls, we'll change this default appearance.

5. Select the check box with the mouse, and in the Properties window locate the `CheckAlign` property. Its value is *`MiddleLeft`*. If you expand the drop-down list by clicking the arrow button, you'll see that this property has many different settings, and each setting is shown as a square. Select the button that will center the text vertically and right-align it horizontally. The string *MiddleRight* will appear in the Properties window when you click the appropriate button.


Properties

chkPayEarly System.Windows.Forms.CheckBox

BackgroundImage	(none)
BackgroundImageLayout	Tile
CausesValidation	True
CheckAlign	**MiddleRight**
Checked	
CheckState	
ContextMenuStri	
Cursor	
Dock	
Enabled	True
⊞ FlatAppearance	
FlatStyle	Standard

CheckAlign
Determines the location of the check box inside the control.

6. With the check box selected, locate the `Name` property in the Properties window, and set it to **chkPayEarly**.

7. Change the CheckBox's caption by entering the string **Early Payment** in its `Text` property field.

8. Place a Button control in the bottom-left corner of the form. Name it `bttnShowPayment`, and set its `Text` property to **`Monthly Payment`**.

9. Finally, place another Button control on the form, name it `bttnExit`, and set its `Text` property to **`Exit`**.

Aligning the Controls

Your next step is to align the controls on the form. The IDE provides commands to align the controls on the form, all of which can be accessed through the Format menu. To align the controls that are already on the form, follow these steps:

1. Select the four labels on the form. The handles of all selected controls will be black, except for one control whose handles will be white. To specify the control that will be used as a reference for aligning the other controls, click it after making the selection. (You can select multiple controls either by drawing a rectangle that encloses them with the mouse, or by clicking each control while holding down the Ctrl button.)
2. With the four text boxes selected, choose Format ➢ Align ➢ Left to left-align them. Don't include the check box in this selection.
3. Resize the CheckBox control. Its left edge should align with the left edges of the Label controls, and its right edge should align with the right edges of the Label controls.
4. Select all the Labels and the CheckBox controls and choose Format ➢ Vertical Spacing ➢ Make Equal. This action will space the controls vertically. Then align the baseline of each TextBox control with the baseline of the matching Label control. To do so, move each TextBox control with the mouse until you see a magenta line that connects the baseline of the TextBox control you're moving and that of the matching Label control.

Your form should now look like the one shown in Figure 4.1. Take a good look at it and check to see whether any of your controls are misaligned. In the interface design process, you tend to overlook small problems such as a slightly misaligned control. The user of the application, however, instantly spots such mistakes.

Programming the Loan Application

Now that you've created the interface, run the application and see how it behaves. Enter a few values in the text boxes, change the state of the check box, and test the functionality already built into the application. Clicking the Monthly Payment button won't have any effect because we have not yet added any code. If this were a prototype you were building for a customer, you would add a statement in the Monthly Payment button to display a random value in the Monthly Payment box. The purpose of the prototype is to get the customer's approval on the appearance and functionality of an application before you start coding it.

If you're happy with the user interface, stop the application, open the form, and double-click the Monthly Payment Button control. Visual Basic opens the code window and displays the definition of the `ShowPayment_ Click` event:

```
Private Sub bttnShowPayment_ Click(...) _
             Handles bttnShowPayment.Click

End Sub
```

Because all `Click` event handlers have the same signature (they provide the same two arguments), I'll be omitting the list of arguments from now on. Actually, all event handlers have two arguments, and the first of them is always the control that fired the event. The type of the second argument differs depending on the type of the event. Place the pointer between the lines `Private Sub` and `End Sub`, and enter the rest of the lines of Listing 4.1. (You don't have to reenter the first and last lines that declare the event handler.)

Listing 4.1: The Code behind the Monthly Payment Button

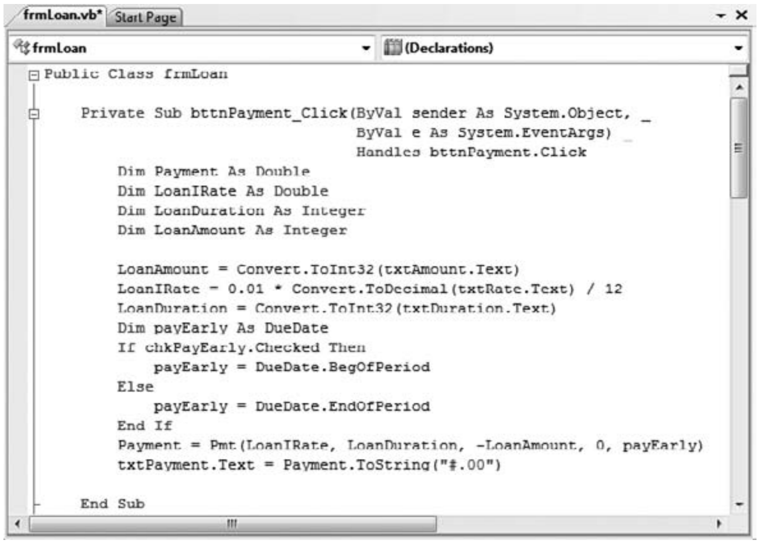
```
Private Sub bttnShowPayment_ Click(...) _
                Handles bttnShowPayment.Click
    Dim Payment As Double
    Dim LoanIRate As Double
    Dim LoanDuration As Integer
    Dim LoanAmount As Integer

    LoanAmount = Convert.ToInt32(txtAmount.Text)
    LoanIRate = 0.01 * Convert.ToDecimal(txtRate.Text) / 12
    LoanDuration = Convert.ToInt32(txtDuration.Text)
    Dim payEarly As DueDate
    If chkPayEarly.Checked Then
        payEarly = DueDate.BegOfPeriod
    Else
        payEarly = DueDate.EndOfPeriod
    End If
    Payment = Pmt(LoanIRate, LoanDuration, -LoanAmount, 0, payEarly)
    txtPayment.Text = Payment.ToString("#.00")
End Sub
```

The code window should now look like the one shown in Figure 4.2. Notice the underscore character at the end of the first part of the long line. The underscore lets you break long lines so that they will fit nicely in the code window. I'm using this convention in this book a lot to fit long lines on the printed page. The same statement you see as multiple lines in the book may appear in a single, long line in the project.

You don't have to break long lines manually as you enter code in the editor's window. Open the Edit menu and choose Advanced ➢ Word Wrap. The editor will wrap long lines automatically at a word boundary. While the word wrap feature is on, a check mark appears in front of the Edit ➢ Advanced ➢ Word Wrap command. To turn off word wrapping, select the same command again.

In Listing 4.1, the first line of code within the subroutine declares a variable. It lets the application know that *`Payment`* is a variable for storing a *floating-point number* (a number with a decimal part) — the Double data type. The line before the If statement declares a variable of the `DueDate` type. This is the type of the argument that determines whether the payment takes place at the beginning or the end of the month. The last argument of the `Pmt()` function must be a variable of this type, so we declare a variable of the `DueDate` type. As mentioned earlier in this chapter, `DueDate` is an enumeration with two members: *`BegOfPeriod`* and *`EndOfPeriod`*.

The first really interesting statement in the subroutine is the `If` statement that examines the value of the *`chkPayEarly`* CheckBox control. If the control is selected, the code sets the *`payEarly`* variable to *`DueDate.BegOfPeriod`*. If not, the code sets the same variable to *`DueDate.EndOfPeriod`*.



```
Public Class frmLoan

    Private Sub bttnPayment_Click(ByVal sender As System.Object, _
                                  ByVal e As System.EventArgs) _
                                  Handles bttnPayment.Click
        Dim Payment As Double
        Dim LoanIRate As Double
        Dim LoanDuration As Integer
        Dim LoanAmount As Integer

        LoanAmount = Convert.ToInt32(txtAmount.Text)
        LoanIRate = 0.01 * Convert.ToDecimal(txtRate.Text) / 12
        LoanDuration = Convert.ToInt32(txtDuration.Text)
        Dim payEarly As DueDate
        If chkPayEarly.Checked Then
            payEarly = DueDate.BegOfPeriod
        Else
            payEarly = DueDate.EndOfPeriod
        End If
        Payment = Pmt(LoanIRate, LoanDuration, -LoanAmount, 0, payEarly)
        txtPayment.Text = Payment.ToString("#.00")

    End Sub
```

FIGURE 4.2
The Show Payment button's `Click` event subroutine.

The ComboBox control's `Checked` property returns True if the control is selected at the time, and returns False otherwise. After setting the value of the *payEarly* variable, the code calls the `Pmt()` function, passing the values of the controls as arguments:

- The first argument is the interest rate. The value entered by the user in the *txtRate* TextBox is multiplied by 0.01 so that the value 14.5 (which corresponds to 14.5 percent) is passed to the `Pmt()` function as 0.145. Although we humans prefer to specify interest rates as integers (8 percent) or floating-point numbers larger than 1 (8.24 percent), the `Pmt()` function expects to read a number less than 1. The value 1 corresponds to 100 percent. Therefore, the value 0.1 corresponds to 10 percent. This value is also divided by 12 to yield the monthly interest rate.
- The second argument is the duration of the loan in months (the value entered in the *txtDuration* TextBox).
- The third argument is the loan's amount (the value entered in the *txtAmount* TextBox).
- The fourth argument (the loan's future value) is 0 by definition.
- The last argument is the *payEarly* variable, which is set according to the status of the *chkPayEarly* control.

The last statement in Listing 4.1 converts the numeric value returned by the `Pmt()` function to a string and displays this string in the fourth TextBox control. The result is formatted appropriately with the following expression:

```
Payment.ToString("#.00")
```

The *Payment* variable is numeric, and all numeric variables provide the method `ToString`, which formats the numeric value and converts it to a string. The character # stands for the integer part of the variable. The period separates the integer from the fractional part, which is rounded to two decimal digits. The `Pmt()` function returns a precise number, such as 372.2235687646345, and you must round it to two decimal digits and format it nicely before displaying it. For more information on formatting numeric (and other) values, see the section ''Formatting Numbers'' in Chapter 2, ''Variables and Data Types.'' Finally, the formatted string is assigned to the `Text` property of the TextBox control on the form.

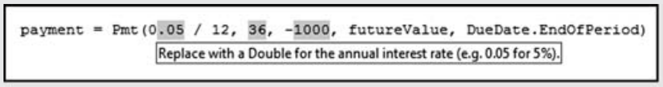
A Code Snippet for Calculating Monthly Loan Payments

If you didn't know about the `Pmt()` built-in function, how would you go about calculating loan payments? Code snippets to the rescue! Right-click somewhere in the code window and from the context menu choose the Insert Snippet command. Double-click the fundamentals to see another list of items. This time double-click the Math folder and then select the snippet *Calculate a Monthly Payment on a Loan*. The following code will be inserted at the location of the pointer (I've broken the last long statement into two lines to fit it on the printed page):

```
Dim futureValue As Double = 0
Dim payment As Double
payment1 = Pmt(0.05 / 12, 36, -1000, futureValue, DueDate.EndOfPeriod)
```

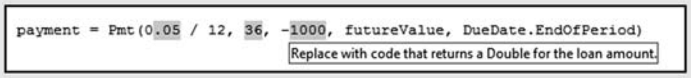
The snippet demonstrates the use of the `Pmt()` function. All you have to do is replace the values of the various parameters with the data from the appropriate controls on the form.

If you don't know how to use the arguments of the `Pmt()` function, rest the pointer over each argument and you will see a description for each argument, as shown here:



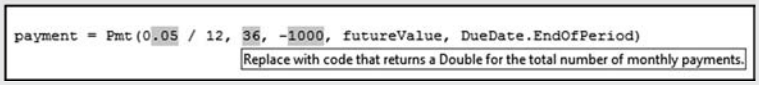
```
payment = Pmt(0.05 / 12, 36, -1000, futureValue, DueDate.EndOfPeriod)
```

Replace with a Double for the annual interest rate (e.g. 0.05 for 5%).



```
payment = Pmt(0.05 / 12, 36, -1000, futureValue, DueDate.EndOfPeriod)
```

Replace with code that returns a Double for the loan amount.



```
payment = Pmt(0.05 / 12, 36, -1000, futureValue, DueDate.EndOfPeriod)
```

Replace with code that returns a Double for the total number of monthly payments.

The code of the LoanCalculator sample project is a bit different and considerably longer than what I have presented here. The statements discussed in the preceding text are the bare minimum for calculating a loan payment. The user can enter all kinds of unreasonable values on the form and cause the program to crash. In the next section, you'll see how you can validate the data entered by the user, catch errors, and handle them gracefully (that is, give the user a chance to correct the data and proceed), as opposed to terminating the application with a runtime error.

Validating the Data

If you enter a non-numeric value in one of the fields, the program will crash and display an error message. For example, if you enter **twenty** in the Duration text box, the program will display the error message shown in Figure 4.3. A simple typing error can crash the program. This isn't the way Windows applications should work. Your applications must be able to handle all kinds of user errors, provide helpful messages, and in general, guide the user in running the application efficiently. If a user error goes unnoticed, your application will either end abruptly or will produce incorrect results without an indication.

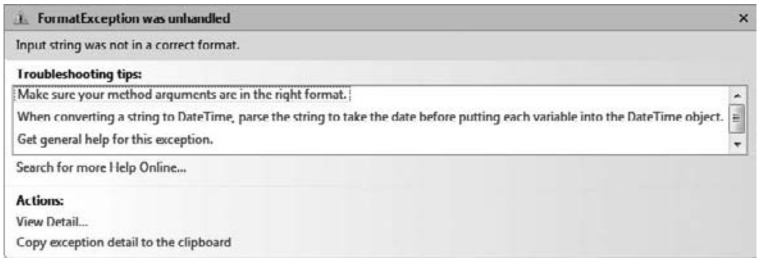

FormatException was unhandled
Input string was not in a correct format.
Troubleshooting tips:
Make sure your method arguments are in the right format.
When converting a string to DateTime, parse the string to take the date before putting each variable into the DateTime object.
Get general help for this exception.
Search for more Help Online...
Actions:
View Detail...
Copy exception detail to the clipboard

FIGURE 4.3
The FormatException error message means that you supplied a string where a numeric value was expected.

Visual Basic will take you back to the application's code window, in which the statements that caused the error will be highlighted in green. Obviously, we must do something about user errors. One way to take care of typing errors is to examine each control's contents; if the controls don't contain valid numeric values, display your own descriptive message and give the user another chance. Listing 4.2 is the revised Click event handler that examines the value of each text box before attempting to use it in any calculations.

LISTING 4.2: Revised Show Payment Button

```
Private Sub bttnShowPayment_ Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles bttnShowPayment.Click
    Dim Payment As Double
    Dim LoanIRate As Double
    Dim LoanDuration As Integer
    Dim LoanAmount As Integer

    ' Validate amount
    If IsNumeric(txtAmount.Text) Then
        LoanAmount = Convert.ToInt32(txtAmount.Text)
    Else
        MsgBox("Please enter a valid amount")
        Exit Sub
    End If
    ' Validate interest rate
    If IsNumeric(txtRate.Text) Then
        LoanIRate = 0.01 * Convert.ToDouble(txtRate.Text) / 12
    Else
```

```
            MsgBox("Invalid interest rate, please re-enter")
            Exit Sub
        End If
        ' Validate loan's duration
        If IsNumeric(txtDuration.Text) Then
            LoanDuration = Convert.ToInt32(txtDuration.Text)
        Else
            MsgBox("Please specify the loan's duration as a number of months")
            Exit Sub
        End If
        ' If all data were validated, proceed with calculations
        Dim payEarly As DueDate
        If chkPayEarly.Checked Then
            payEarly = DueDate.BegOfPeriod
        Else
            payEarly = DueDate.EndOfPeriod
        End If
        Payment = Pmt(LoanIRate, LoanDuration, -LoanAmount, 0, payEarly)
        txtPayment.Text = Payment.ToString("#.00")
    End Sub
```

First, we declare three variables in which the loan's parameters will be stored: *LoanAmount*, *LoanIRate*, and *LoanDuration*. These values will be passed to the `Pmt()` function as arguments. Each text box's value is examined with an `If` structure. If the corresponding text box holds a valid number, its value is assigned to the numeric variable. If not, the program displays a warning and exits the subroutine without attempting to calculate the monthly payment. Before exiting the subroutine, however, the code moves the focus to the text box with the invalid value because this is the control that the user will most likely edit. After fixing the incorrect value, the user can click the Show Payment button again. `IsNumeric()` is another built-in function that accepts a variable and returns True if the variable is a number, and returns False otherwise.

You can run the revised application and check it out by entering invalid values in the fields. Notice that you can't specify an invalid value for the last argument; the CheckBox control won't let you enter a value. You can only select or clear it, and both options are valid. The actual calculation of the monthly payment takes a single line of Visual Basic code. Displaying it requires another line of code. Adding the code to validate the data entered by the user, however, is an entire program. And that's the way things are.

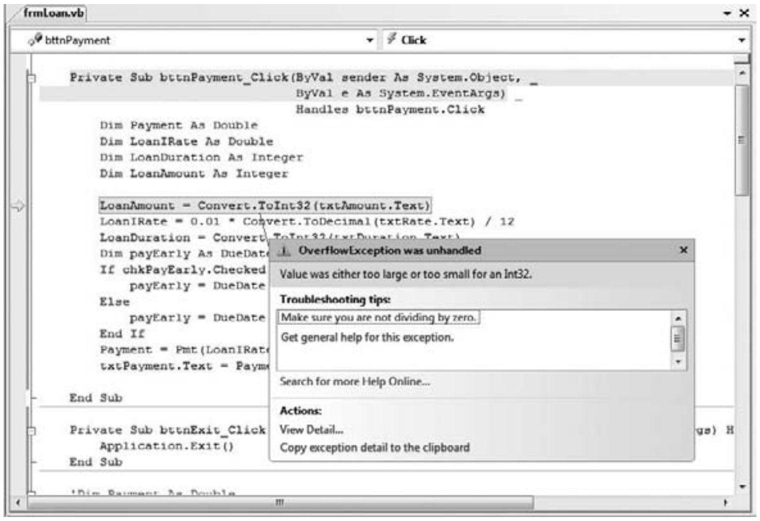
Writing Well-Behaved Applications

A well-behaved application must contain data-validation code. If an application such as LoanCalculator crashes because of a typing mistake, nothing really bad will happen. The user will try again or else give up on your application and look for a more professional one. However, if the user has been entering data for hours, the situation is far more serious. It's your responsibility as a programmer to make sure that only valid data are used by the application and that the application keeps working, no matter how the user misuses or abuses it.

The amount of code you write to validate user input is comparable to the amount of code that produces the results. Our sample application is not typical, because it calculates the result with a single function call, but in developing typical business applications, you must write a substantial amount of code to validate user input. The reason for validating user input is that you should provide specific error messages to help the user identify the error and correct it.

The applications in this book don't contain much data-validation code because it would obscure the ''useful'' code that applies to the topic at hand. Instead, they demonstrate specific techniques. You can use parts of the examples in your applications, but you should provide your own data-validation code (and error-handling code, as you'll see in a moment).

Now run the application one last time and enter an enormous loan amount. Try to find out what it would take to pay off the national debt with a reasonable interest rate in, say, 72 months. The program will crash again (as if you didn't know). This time the program will go down with a different error message, as shown in Figure 4.4. Visual Basic will complain about an overflow. The exact message is *Value was either too large or too small for an Int32*, and the program will stop at the line that assigns the contents of the *`txtAmount`* TextBox to the *`LoanAmount`* variable. Press the Break button, and the offending statement in the code will be highlighted.



```
    Private Sub bttnPayment_Click(ByVal sender As System.Object, _
                                  ByVal e As System.EventArgs) _
                                  Handles bttnPayment.Click
        Dim Payment As Double
        Dim LoanIRate As Double
        Dim LoanDuration As Integer
        Dim LoanAmount As Integer

        LoanAmount = Convert.ToInt32(txtAmount.Text)
        LoanIRate = 0.01 * Convert.ToDecimal(txtRate.Text) / 12
        LoanDuration = Convert
        Dim payEarly As DueDate
        If chkPayEarly.Checked
            payEarly = DueDate
        Else
            payEarly = DueDate
        End If
        Payment = Pmt(LoanIRat
        txtPayment.Text = Paym

    End Sub

    Private Sub bttnExit_Click
        Application.Exit()
    End Sub
```

OverflowException was unhandled

Value was either too large or too small for an Int32.

Troubleshooting tips:

Make sure you are not dividing by zero.

Get general help for this exception.

Search for more Help Online...

Actions:

View Detail...

Copy exception detail to the clipboard

FIGURE 4.4
Very large values can cause the application to crash and display this error message.

An *overflow* is a numeric value too large for the program to handle. This error is usually produced when you divide a number by a very small value. When you attempt to assign a very large value to an Integer variable, you'll also get an overflow exception.

Actually, in the LoanCalculator application, any amount greater than 2,147,483,647 will cause an overflow condition. This is the largest value you can assign to an Integer variable; it's plenty for our banking needs, but not nearly adequate for handling government deficits. As you'll see in the next chapter, Visual Basic provides other types of variables, which can store enormous values (making the national debt look really small). In the meantime, if you want to use the loan calculator, change the declaration of the *`LoanAmount`* variable to the following:

```
Dim LoanAmount As Double
```

The Double data type can hold much larger values. Besides, the Double data type can also hold noninteger values. Not that anyone will ever apply for a loan of $25,000 and some cents, but if you want to calculate the precise monthly payment for a debt you have accumulated, you should be able to specify a noninteger amount. In short, we should have declared the `LoanAmount` variable with the Double data type in the first place. By the way, there's another integer data type, the Long data type, which can hold much larger integer values.

An overflow error can't be caught with data-validation code. There's always a chance that your calculations will produce overflows or other types of math errors. Data validation won't help here; you just don't know the result before you carry out the calculations. We need something called *error handling*, or *exception handling*. This is additional code that can handle errors after they occur. In effect, you're telling VB that it shouldn't stop with an error message, which would be embarrassing for you and wouldn't help the user one bit. Instead, VB should detect the error and execute the proper statements that will handle the error. Obviously, you must supply these statements. (You'll see examples of handling errors at runtime shortly.)

The sample application works as advertised and it's fail-safe. Yet there's one last touch we can add to our application. The various values on the form are not always in synch. Let's say you've calculated the monthly payment for a specific loan and then you want to change the duration of the loan to see how it affects the monthly payment. As soon as you change the duration of the loan, and before you click the Monthly Payment button, the value in the Monthly Payment box doesn't correspond to the parameters of the loan. Ideally, the monthly payment should be cleared as soon as the user starts editing one of the loan's parameters. To do so, you must insert a statement that clears the *`txtPayment`* control. But what's the proper event handler for this statement? The TextBox control fires the `TextChanged` event every time its text is changed, and this is the proper place to execute the statement that clears the monthly payment on the form. Because there are three TextBox controls on the form, you must program the `TextChanged` event of all three controls, or write an event handler that handles all three events:

```
Private Sub txtAmount_ TextChanged(...) _
        Handles txtAmount.TextChanged, _
                txtDuration.TextChanged, txtRate.TextChanged
    txtPayment.Clear()
End Sub
```

Yes, you can write a common handler for multiple events, as long as the events are of the same type and they're all listed after the `Handles` keyword. You'll see another example of the same technique in the following sample project.

One of the sample projects for this chapter is a revised version of the LoanCalculator project, the LoanCalculator-Dates project, which uses a different interface. Instead of specifying the duration of the loan in months, this application provides two instances of the `DateTimePicker` control, which is used to specify dates. Delete the `TextBox` control and the corresponding Labels and insert two new Labels and two `DateTimePicker` controls on the form. Users can set the loan's starting and ending dates on these two controls and the program calculates the duration of the loan in moths with the following statement:

```
LoanDuration = DateDiff(DateInterval.Month, _
                        dtFrom.Value, dtTo.Value) + 1
```

`dtFrom` and *`dtTo`* are the names of the two DateTimePicker controls. The `DateDiff()` function returns the difference between two dates in the interval supplier as the first argument to the

function. The rest of the code doesn't change; as long as the `LoanDuration` variable has the correct value, the same statements will produce the correct result. If you open the project you'll find a few more interesting statements that set the *dtFrom* control to the first date of the selected month and the *dtTo* control to the last date of the selected month.

Building a Calculator

Our next application is more advanced, but not as advanced as it looks. It's a calculator with a typical visual interface that demonstrates how Visual Basic can simplify the programming of fairly advanced operations. If you haven't tried it, you may think that writing an application such as this one is way too complicated for a beginner, but it isn't. The MathCalculator application is shown in Figure 4.5.

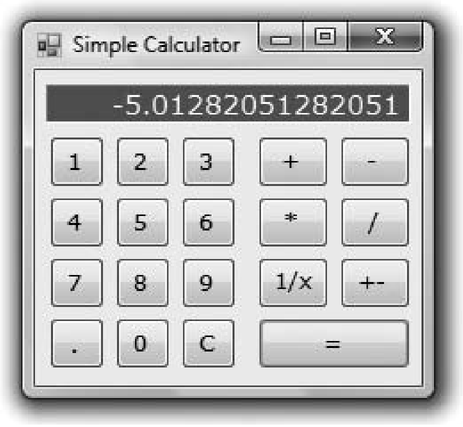


FIGURE 4.5
Calculator application window

The application emulates the operation of a hand-held calculator and implements the basic arithmetic operations. It has the look of a math calculator, and you can easily expand it by adding more features. In fact, adding features such as cosines and logarithms is actually simpler than performing the basic arithmetic operations. This interface will also give us a chance to exercise most of the tools of the IDE for aligning and spacing the controls on a form.

Designing the User Interface

The application's interface is straightforward, but it takes a bit of effort. You must align the buttons on the form and make the calculator look as much like a hand-held calculator as possible. Start a new project, the MathCalculator project, and rename its main form from `Form1.vb` to **`frmCalculator.vb`**.

Designing the interface of the application isn't trivial because it's made up of many buttons, all perfectly aligned on the form. To simplify the design, follow these steps:

1. Select a font that you like for the form. All the command buttons you'll place on the form will inherit this font. The MathCalculator sample application uses 10-point Verdana font. I've used a size of 12 points for the Period button, because the 10-point period was too small and very near the bottom of the control.
2. Add the Label control, which will become the calculator's display. Set its `BorderStyle` property to Fixed3D so that it will have a 3D look, as shown in Figure 4.5. Change its

`ForeColor` and `BackColor` properties too, if you want it to look different from the rest of the form. The sample project uses colors that emulate the — now extinct — green CRT monitors.

3. Draw a Button control on the form, change its `Text` property to **1**, and name it **`bttn1`**. Size the button carefully so that its caption is centered on the control. The other buttons on the form will be copies of this one, so make sure you've designed the first button as best as you can before you start making copies of it. You can also change the button's style with the `FlatStyle` property. (You can experiment with the Popup, Standard, and System settings of this property.)
4. Place the button in its final position on the form. At this point, you're ready to create the other buttons for the calculator's digits. Right-click the button and choose Copy from the context menu. The Button control is copied to the Clipboard, and now you can paste it on the form (which is much faster than designing an identical button).
5. Right-click somewhere on the form, choose Paste, and the button copied to the Clipboard will be pasted on the form. The copy will have the same caption as the button it was copied from, and its name will be `Button1`.
6. Now set the button's `Name` to **`bttn2`** and its `Text` property to **2**. This button is the digit **2**. Place the new button to the right of the previous button. You don't have to align the two buttons perfectly now; later we'll use the Format menu to align the buttons on the form. As you move the control around on the form, one or more lines may appear at times. These lines are called *snap lines*, and they appear as soon as a control is aligned (vertically or horizontally) with one or more of the existing controls on the form. The snap lines allow you to align controls with the mouse. Blue snap lines appear when the control's edge is aligned with the edge of another control. Red snap lines appear when the control's baseline is aligned with the baseline of another control. The baseline is the invisible line on which the characters of the control's caption are based.
7. Repeat steps 5 and 6 eight more times, once for each numeric digit. Each time a new Button control is pasted on the form, Visual Basic names it `Button1` and sets its caption to 1; you must change the `Name` and `Text` properties. You can name the buttons anything you like, but a name that indicates their role in the application is preferred.
8. When the buttons of the numeric digits are all on the form, place two more buttons, one for the C (Clear) operation and one for the Period button. Name them **`bttnClear`** and **`bttnPeriod`**, and set their captions accordingly. Use a larger font size for the Period button to make its caption easier to read.
9. When all the digit buttons of the first group are on the form and in their approximate positions, align them by using the commands of the Format menu. You can use the snap lines to align horizontally and vertically the various buttons on the form, but you must still space the controls manually, which isn't a trivial task. Here's how you can align the buttons perfectly via the Format menu:
   a. First, align the buttons of the top row. Start by aligning the 1 button with the left side of the `lblDisplay` Label. Then select all the buttons of the top row and make their horizontal spacing equal (choose Format ➢ Horizontal Spacing ➢ Make Equal). Then do the same with the buttons in the first column; this time, make sure that their vertical distances are equal (Format ➢ Vertical Spacing ➢ Make Equal).

b. Now you can align the buttons in each row and each column separately. Use one of the buttons you aligned in the last step as the guide for the rest of them. The buttons can be aligned in many ways, so don't worry if somewhere in the process you ruin the alignment. You can always use the Undo command in the Edit menu. Select the three buttons on the second row and align their Tops by using the first button as a reference. To set the anchor control for the alignment, click it with the mouse while holding down the Ctrl key. Do the same for the third and fourth rows of buttons. Then do the same for the four columns of buttons, using the top button as a reference.

10. Now, place the buttons for the arithmetic operations on the form — addition (+), subtraction (−), multiplication (*), and division (/).

11. Finally, place the Equals button on the form and make it wide enough to span the space of two operation buttons. Use the commands on the Format menu to align these buttons, as shown in Figure 4.5. The form shown in Figure 4.5 has a few more buttons, which you can align by using the same techniques you used to align the numeric buttons.

If you don't feel quite comfortable with the alignment tools of the IDE, you can still position the controls on the form through the x and y components of each control's `Location` property. (They're the x- and y-coordinates of the control's upper-left corner on the form.) The various alignment tools are among the first tools of the IDE you'll master, and you'll be creating forms with perfectly aligned controls in no time at all.

Programming the MathCalculator

Now you're ready to add some code to the application. Double-click one of the digit buttons on the form, and you'll see the following in the code window:

```
Private Sub bttn1_ Click(ByVal sender As System.Object, _
              ByVal e As System.EventArgs) Handles bttn1.Click

End Sub
```

This is the `Click` event's handler for a single digit button. Your first attempt is to program the `Click` event handler of each digit button, but repeating the same code 10 times isn't very productive. (Not to mention that if we decide to edit the code later, the process must be repeated 10 times.) We're going to use the same event handler for all buttons that represent digits. All you have to do is append the names of the events to be handled by the same subroutine after the `Handles` keyword. You should also change the name of the event handler to something that indicates its role. Because this subroutine handles the `Click` event for all the digit buttons, let's call it `DigitClick()`. Here's the revised declaration of a subroutine that can handle all the digit buttons:

```
Private Sub DigitClick(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) _
               Handles bttn0.Click, bttn1.Click, bttn2.Click, _
               bttn3.Click, bttn4.Click, bttn5.Click, bttn6.Click, _
               bttn7.Click, bttn8.Click, bttn9.Click
End Sub
```

You don't have to type all the event names; as soon as you insert the first comma after `bttn0.Click`, a drop-down list with the names of the controls will open, and you can select the name of the next button with the down arrow. Press the spacebar to select the desired control (`bttn1`, `bttn2`, and so on), and then type the period. This time, you'll see another list with the names of the events for the selected control. Locate the `Click` event and select it by pressing the spacebar. Type the next comma and repeat the process for all the buttons. This extremely convenient feature of the language is IntelliSense: The IDE presents the available and valid keywords as you type.

When you press a digit button on a hand-held calculator, the corresponding digit is appended to the display. To emulate this behavior, insert the following line in the `Click` event handler:

```
lblDisplay.Text = lblDisplay.Text + sender.Text
```

This line appends the digit clicked to the calculator's display. The *`sender`* argument of the `Click` event represents the control that was clicked (the control that fired the event). The `Text` property of this control is the caption of the button that was clicked. For example, if you have already entered the value 345, clicking the digit 0 displays the value 3450 on the Label control that acts as the calculator's display.

The expression `sender.Text` is not the best method of accessing the `Text` property of the button that was clicked, but it will work as long as the Strict option is off. As discussed in Chapter 2, we must cast the *`sender`* object to a specific type (the Button type) and then call its `Text` method:

```
CType(sender, Button).Text
```

The code behind the digit buttons needs a few more lines. After certain actions, the display should be cleared. After pressing one of the buttons that correspond to math operations, the display should be cleared in anticipation of the second operand. Actually, the display must be cleared as soon as the first digit of the second operand is pressed, and not as soon as the math operator button is pressed. Likewise, the display should also be cleared after the user clicks the Equals button. Revise the `DigitClick` event handler, as shown in Listing 4.3.

LISTING 4.3: The *DigitClick* Event

```
Private Sub DigitClick(ByVal sender As System.Object, _
                        ByVal e As System.EventArgs) _
                        Handles bttn1.Click, bttn2.Click, bttn3.Click, _
                        bttn4.Click, bttn5.Click, bttn6.Click, _
                        bttn7.Click, bttn8.Click, bttn9.Click
   If clearDisplay Then
      lblDisplay.Text = ""
      clearDisplay = False
   End If
   lblDisplay.Text = lblDisplay.Text + sender.text
End Sub
```

The *clearDisplay* variable is declared as Boolean, which means it can take a True or False value. Suppose that the user has performed an operation and the result is on the calculator's display. The user now starts typing another number. Without the `If` clause, the program would continue to append digits to the number already on the display. This is not how calculators work. When the user starts entering a new number, the display must be cleared. And our program uses the *clearDisplay* variable to know when to clear the display.

The Equals button sets the *clearDisplay* variable to True to indicate that the display contains the result of an operation. The `DigitClick()` subroutine examines the value of this variable each time a new digit button is pressed. If the value is True, `DigitClick()` clears the display and then prints the new digit on it. The subroutine also sets *clearDisplay* to False so that when the next digit is pressed, the program won't clear the display again.

What if the user makes a mistake and wants to undo an entry? The typical hand-held calculator has no Backspace key. The Clear key erases the current number on the display. Let's implement this feature. Double-click the C button and enter the code of Listing 4.4 in its `Click` event.

Listing 4.4: Programming the Clear Button

```
Private Sub bttnClear_ Click(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles bttnClear.Click
   lblDisplay.Text = ""
End Sub
```

Now we can look at the Period button. A calculator, no matter how simple, should be able to handle fractional numbers. The Period button works just like the digit buttons, with one exception. A digit can appear any number of times in a numeric value, but the period can appear only once. A number such as 99.991 is valid, but you must make sure that the user can't enter numbers such as 23.456.55. After a period is entered, this button must not insert another one. The code in Listing 4.5 accounts for this.

Listing 4.5: Programming the Period Button

```
Private Sub bttnPeriodClick(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles bttnPeriod.Click
   If lblDisplay.Text.IndexOf(".") >= 0 Then
      Exit Sub
   Else
      lblDisplay.Text = lblDisplay.Text & "."
   End If
End Sub
```

`IndexOf` is a method that can be applied to any string. The expression `lblDisplay.Text` is a string (the text on the Label control), so we can call its `IndexOf` method. The expression `lblDisplay.Text.IndexOf(''.'')` returns the location of the first instance of the period in the caption of the Label control. If this number is zero or positive, the number entered contains a period already, and another can't be entered. In this case, the program exits the subroutine. If the method returns −1, the period is appended to the number entered so far, just like a regular digit.

Check out the operation of the application. We have already created a functional user interface that emulates a hand-held calculator with data-entry capabilities. It doesn't perform any operations yet, but we have already created a functional user interface with only a small number of statements.

Coding the Math Operations

Now we can move to the interesting part of the application: the coding of the math operations. Let's start by defining three variables:

`Operand1`	The first number in the operation
`Operator`	The desired operation
`Operand2`	The second number in the operation

When the user clicks one of the math symbols, the value on the display is stored in the variable *`Operand1`*. If the user then clicks the Plus button, the program must make a note to itself that the current operation is an addition and set the *`clearDisplay`* variable to True so that the user can enter another value (the second value to be added). The symbol of the operation is stored in the *`Operator`* variable. The user enters another value and then clicks the Equals button to see the result. At this point, our program must do the following:

1. Read the value on the display into the *`Operand2`* variable.
2. Perform the operation indicated by the *`Operator`* variable with the two operands.
3. Display the result and set the *`clearDisplay`* variable to True.

The Equals button must perform the following operation:

```
Operand1 Operator Operand2
```

Suppose that the number on the display when the user clicks the Plus button is 3342. The user then enters the value 23 and clicks the Equals button. The program must carry out the addition:

```
3342 + 23
```

If the user clicked the Division button, the operation is as follows:

```
3342 / 23
```

Variables are local in the subroutines in which they are declared. Other subroutines have no access to them and can't read or set their values. Sometimes, however, variables must be accessed from many places in a program. The variables *`Operand1`*, *`Operand2`*, and *`Operator`*, as well as the *`clearDisplay`* variable, must be accessed from within more than one subroutine, so they must be declared outside any subroutine; their declarations usually appear at the beginning of the code with the following statements:

```
Dim clearDisplay As Boolean
Dim Operand1 As Double
Dim Operand2 As Double
Dim Operator As String
```

These variables are called *form-wide variables*, or simply *form variables*, because they are visible from within any subroutine on the form. Let's see how the program uses the `Operator` variable. When the user clicks the Plus button, the program must store the value "+" in the `Operator` variable. This takes place from within the Plus button's `Click` event.

All variables that store numeric values are declared as variables of the Double type, which can store values with the greatest possible precision. The Boolean type takes two values: True and False. You have already seen how the `clearDisplay` variable is used.

With the variable declarations out of the way, we can now implement the operator buttons. Double-click the Plus button and, in the `Click` event's handler, enter the lines shown in Listing 4.6.

Listing 4.6: The Plus Button

```
Private Sub bttnPlus_ Click(...) Handles bttnPlus.Click
   Operand1 = Convert.ToDouble(lblDisplay.Text)
   Operator = "+"
   clearDisplay = True
End Sub
```

The variable `Operand1` is assigned the value currently on the display. The `Convert.ToDouble()` method converts its argument to a double value. The `Text` property of the Label control is a string. The actual value stored in the `Text` property is not a number. It's a string such as `428`, which is different from the numeric value 428. That's why we use the `Convert.ToDouble` method to convert the value of the Label's caption to a numeric value. The remaining buttons do the same, and I won't show their listings here.

After the second operand is entered, the user can click the Equals button to calculate the result. When this happens, the code of Listing 4.7 is executed.

Listing 4.7: The Equals Button

```
Private Sub bttnEquals_ Click(...) Handles bttnEquals.Click
   Dim result As Double
   Operand2 = Convert.ToDouble(lblDisplay.Text)
   Select Case Operator
      Case "+"
         result = Operand1 + Operand2
      Case "-"
```

```
            result = Operand1 - Operand2
        Case "*"
            result = Operand1 * Operand2
        Case "/"
            If Operand2 <> "0" Then
                result = Operand1 / Operand2
    End Select
    lblDisplay.Text = result.ToString
    clearDisplay = True
End Sub
```

The *result* variable is declared as Double so that the result of the operation will be stored with maximum precision. The code extracts the value displayed in the Label control and stores it in the variable *Operand2*. It then performs the operation with a `Select Case` statement. This statement compares the value of the *Operator* variable to the values listed after each `Case` statement. If the value of the *Operator* variable matches one of the `Case` values, the following statement is executed.

Division takes into consideration the value of the second operand, because if it's zero, the division can't be carried out. The last statement carries out the division only if the divisor is not zero. If *Operand2* happens to be zero, nothing happens.

Now run the application and check it out. It works just like a hand-held calculator, and you can't crash it by specifying invalid data. We didn't have to use any data-validation code in this example because the user doesn't get a chance to type invalid data. The data-entry mechanism is foolproof. The user can enter only numeric values because there are only numeric digits on the calculator. The only possible error is to divide by zero, and that's handled in the Equals button.

Of course, users should be able to just type the numeric values; you shouldn't force them to click their digits. To intercept keystrokes from within your code, you must first set the form's `KeyPreview` property to True. Each keystroke is reported to the control that has the focus at the time and fires the keystroke-related events: the `KeyDown`, `KeyPress`, and `KeyUp` events. Sometimes we need to handle certain keystrokes from a central place, and we set the form's `KeyPreview` property to True, so that keystrokes are reported first to the form and then to the control that has the focus. We can intercept the keystrokes in the form's `KeyPress` event and handle them in this event handler. Insert the statements shown in Listing 4.8 in the form's `KeyPress` event handler.

LISTING 4.8: Handling Keystrokes at the Form's Level

```
Private Sub CalculatorForm_ KeyPress(...) Handles Me.KeyPress
    Select Case e.KeyChar
        Case "1" : bttn1.PerformClick()
        Case "2" : bttn2.PerformClick()
        Case "3" : bttn3.PerformClick()
        Case "4" : bttn4.PerformClick()
        Case "5" : bttn5.PerformClick()
        Case "6" : bttn6.PerformClick()
        Case "7" : bttn7.PerformClick()
```

```
            Case "8" : bttn8.PerformClick()
            Case "9" : bttn9.PerformClick()
            Case "0" : bttn0.PerformClick()
            Case "." : bttnPeriod.PerformClick()
            Case "C", "c" : bttnClear.PerformClick()
            Case "+" : bttnPlus.PerformClick()
            Case "-" : bttnMinus.PerformClick()
            Case "*" : bttnMultiply.PerformClick()
            Case "/" : bttnDivide.PerformClick()
            Case "=" : bttnEquals.PerformClick()
        End Select
    End Sub
```

This event handler examines the key pressed by the user and invokes the `Click` event handler of the appropriate button by calling its `PerformClick` method. This method allows you to ''click'' a button from within your code. When the user presses the digit **3**, the form's `KeyPress` event handler intercepts the keystrokes and emulates the click of the `bttn3` button.

Using Simple Debugging Tools

Our sample applications work nicely and are quite easy to test and fix if you discover something wrong with them (but only because they're very simple applications). As you write code, you'll soon discover that something doesn't work as expected, and you should be able to find out why and then fix it. The process of eliminating errors is called *debugging*, and Visual Studio provides the tools to simplify the process of debugging. (These tools are discussed in detail in Appendix B.) There are a few simple debugging techniques you should know, even as you work with simple projects.

Open the MathCalculator project in the code editor and place the pointer in the line that calculates the difference between the two operands. Let's pretend there's a problem with this line, and we want to follow the execution of the program closely to find out what's going wrong with the application. Press F9, and the line will be highlighted in brown. This line has become a *breakpoint:* As soon as it is reached, the program will stop.

Press F5 to run the application and perform a subtraction. Enter a number; then click the minus button, and then another number, and finally the Equals button. The application will stop, and the code editor will open. The breakpoint will be highlighted in yellow. You're still in runtime mode, but the execution of the application is suspended. You can even edit the code in break mode and then press F5 to continue the execution of the application. Hover the pointer over the *`Operand1`* and *`Operand2`* variables in the code editor's window. The value of the corresponding variable will appear in a small ToolTip box. Move the pointer over any variable in the current event handler to see its value. These are the values of the variables just prior to the execution of the highlighted statement.

The *`result`* variable is zero because the statement hasn't been executed yet. If the variables involved in this statement have their proper values (if they don't, you know that the problem is prior to this statement and perhaps in another event handler), you can execute this statement by pressing F10, which executes only the highlighted statement. The program will stop at the next line. The next statement to be executed is the `End Select` statement.

Find an instance of the *`result`* variable in the current event handler, rest the pointer over it, and you will see the value of the variable after it has been assigned a value. Now you can press F10 to execute another statement or press F5 to return to normal execution mode.

You can also evaluate expressions involving any of the variables in the current event handler by entering the appropriate statement in the Immediate window. The Immediate window appears at the bottom of the IDE. If it's not visible, open the Debug menu and choose Windows ➤ Immediate. The current line in the command window is prefixed with the greater-than symbol (reminiscent of the DOS days). Place the cursor next to it and enter the following statement:

```
? Operand1 / Operand2
```

The quotient of the two values will appear in the following line. The question mark is just a shorthand notation for the `Print` command. If you want to know the current value on the calculator's display, enter the following statement:

```
? lblDisplay.Text
```

This statement requests the value of a control's property on the form. The current value of the Label control's `Text` property will appear in the following line. You can also evaluate math expressions with statements such as the following:

```
? Math.Log(3/4)
```

`Log()` is the logarithm function and a method of the Math class. With time, you'll discover that the Immediate window is a handy tool for debugging applications. If you have a statement with a complicated expression, you can request the values of the expression's individual components and make sure they can be evaluated.

Now move the pointer over the breakpoint and press F9 again. This will toggle the breakpoint status, and the execution of the program won't halt the next time this statement is executed.

If the execution of the program doesn't stop at a breakpoint, it means that the statement is never reached. In this case, you must search for the bug in statements that are executed before the breakpoint is reached. If you didn't assign the proper value to the *`Operator`* variable, the `Case` clause for the subtraction operation will never be reached. You should place the breakpoint at the first executable statement of the Equal button's `Click` event handler to examine the values of all variables the moment this subroutine starts its execution. If all variables have the expected values, you will continue testing the code forward. If not, you'd have to test the statements that lead to this statement — the statements in the event handlers of the various buttons.

Another simple technique for debugging applications is to print the values of certain variables in the Immediate window. Although this isn't a debugging tool, it's common among VB programmers (and very practical, I might add). Many programmers print the values of selected variables before and after the execution of some complicated statements. To do so, use the statement `Debug.WriteLine` followed by the name of the variable you want to print, or an expression:

```
Debug.WriteLine(Operand1)
```

This statement sends its output to the Immediate window. This is a simple technique, but it works. You can also use it to test a function or method call. If you're not sure about the syntax of a function, pass an expression that contains the specific function to the `Debug.WriteLine` statement as an argument. If the expected value appears in the Immediate window, you can go ahead and use it in your code.

In the project's folder, you will find the `MoreFeatures.txt` document, which describes how to add more features to the math calculator. Such features include the inversion of a number (the 1/x button), the negation of a number (the +/− button), and the usual math functions (logarithms, square roots, trigonometric functions, and so on).

Exception Handling

Crashing this application won't be as easy as crashing the LoanCalculator application. If you start multiplying very large numbers, you won't get an overflow exception. Enter a very large number by repeatedly typing the digit **9**; then multiply this value with another equally large value. When the result appears, click the multiplication symbol and enter another very large value. Keep multiplying the result with very large numbers until you exhaust the value range of the Double data type (that is, until the result is so large that it can't be stored to a variable of the Double type). When this happens, the string *infinity* will appear in the display. This is Visual Basic's way of telling you that it can't handle very large numbers. This isn't a limitation of VB; it's the way computers store numeric values: They provide a limited number of bytes for each variable. (We discussed oddities such as infinity in Chapter 2.)

You can't create an overflow exception by dividing a number by zero, either, because the code will not even attempt to carry out this calculation. In short, the MathCalculator application is pretty robust. However, we can't be sure that users won't cause the application to generate an exception, so we must provide some code to handle all types of errors.

Exceptions versus *Errors*

Errors are now called *exceptions*. You can think of them as exceptions to the normal (or intended) flow of execution. If an exception occurs, the program must execute special statements to handle the exception — statements that wouldn't be executed normally. I think they're called exceptions because *error* is a word nobody likes, and most people can't admit they wrote code that contains errors. The term *exception* can be vague. What would you rather tell your customers: that the application you wrote has errors or that your code has raised an exception? You may not have noticed it, but the term *bug* is not used as frequently anymore; bugs are now called *known issues*. The term *debugging*, however, hasn't changed yet.

How do you prevent an exception raised by a calculation? Data validation won't help. You just can't predict the result of an operation without actually performing the operation. And if the operation causes an overflow, you can't prevent it. The answer is to add a *structured exception handler*. Most of the application's code is straightforward, and you can't easily generate an exception for demonstration purposes. The only place where an exception may occur is the handler of the Equals button, where the calculations take place. This is where we must add an exception handler. The outline of the structured exception handler is the following:

```
Try
   { statements block}
Catch Exception
```

```
   { handler block}
Finally
   { clean-up statements block}
End Try
```

The program will attempt to perform the calculations, which are coded in the *statements block*. If the program succeeds, it continues with the *cleanup statements*. These statements are mostly cleanup code, and the `Finally` section of the statement is optional. If missing, the program execution continues with the statement following the `End Try` statement. If an error occurs in the first block of statements, the `Catch Exception` section is activated, and the statements in the *handler block* are executed.

The `Catch` block is where you handle the error. There's not much you can do about errors that result from calculations. All you can do is display a warning and give the user a chance to change the values. There are other types of errors, however, that can be handled much more gracefully. If your program can't read a file from a CD drive, you can give the user a chance to insert the CD and retry. In other situations, you can prompt the user for a missing value and continue. If the application attempts to write to a read-only file, for example, chances are that the user specified a file on a CD drive, or a file with its read-only attribute set. You can display a warning, exit the subroutine that saves the data, and give the user a chance to either select another filename or change the read-only attribute of the selected file.

In general, there's no unique method to handle all exceptions. You must consider all types of exceptions that your application may cause and handle them on an individual basis. What's important about error handlers is that your application doesn't crash; it simply doesn't perform the operation that caused the exception (this is also known as the *offending operation*, or *offending statement*) and continues.

The error handler for the MathCalculator application must inform the user that an error occurred and abort the calculations — not even attempt to display a result. If you open the Equals button's `Click` event handler, you will find the statements detailed in Listing 4.9.

LISTING 4.9: Revised Equals Button

```
Private Sub bttnEquals_ Click(ByVal sender As System.Object, _
             ByVal e As System.EventArgs) Handles bttnEquals.Click
   Dim result As Double
   Operand2 = Convert.ToDouble(lblDisplay.Text)
   Try
      Select Case Operator
         Case "+"
            result = Operand1 + Operand2
         Case "-"
            result = Operand1 - Operand2
         Case "*"
            result = Operand1 * Operand2
         Case "/"
            If Operand2 <> "0" Then result = Operand1 / Operand2
        End Select
        lblDisplay.Text = result
   Catch exc As Exception
      MsgBox(exc.Message)
      result = "ERROR"
   Finally
      clearDisplay = True
   End Try
End Sub
```

Most of the time, the error handler remains inactive and doesn't interfere with the operation of the program. If an error occurs, which most likely will be an overflow error, the error-handling section of the `Try...Catch...End Try` statement will be executed. This code displays a message box with the description of the error, and it also displays the string `ERROR` on the calculator's display. The `Finally` section is executed regardless of whether an exception occurred. In this example, the `Finally` section sets the `clearDisplay` variable to True so that when another digit button is clicked, a new number will appear on the display.

The Bottom Line

Design graphical user interfaces. A Windows application consists of a graphical user interface and code. The interface of the application is designed with visual tools and consists of controls that are common to all Windows applications. You drop controls from the Toolbox window onto the form, size and align the controls on the form, and finally set their properties through the Properties window. The controls include quite a bit of functionality right out of the box, and this functionality is readily available to your application without a single line of code.

Master It Describe the process of aligning controls on a form.

Program events. Windows applications follow an event-driven model: We code the events to which we want our application to respond. The `Click` events of the various buttons are typical events to which an application reacts. You select the actions to which you want your application to react and program these events accordingly.

When an event is fired, the appropriate event handler is automatically invoked. Event handlers are subroutines that pass two arguments to the application: the `sender` object (which is an object that represents the control that fired the event) and the `e` argument (which carries additional information about the event).

Master It How will you handle certain keystrokes regardless of the control that receives them?

Write robust applications with error handling. Numerous conditions can cause an application to crash, but a professional application should be able to detect abnormal conditions and handle them gracefully. To begin with, *you should always validate your data* before you attempt to use them in your code. A well-known computer term is ''garbage in, garbage out'', which means you shouldn't perform any calculations on invalid data.

Master It How will you execute one or more statements in the context of a structured exception handler?

Chapter 5

The Vista Interface

With the introduction of Windows Vista and the .NET Framework 3.5, Microsoft has overhauled its entire graphical presentation technology. The new application programming interface (API) is known as the *Windows Presentation Foundation (WPF)*. WPF is one of the core technologies in the .NET Framework 3.5 and is integrated into Windows Vista. WPF is also supported on Windows XP.

WPF offers a whole new approach to developing graphical user interfaces (UIs) for the Windows platform. WPF is designed to take advantage of the graphics engines and display capabilities of the modern computer, and is vector based and resolution independent. This means that your UIs automatically scale depending on the size and resolution of the user's screen. This is enhanced by the ability to place objects using relative positioning (so that objects sit relative to other objects) rather than the more traditional absolute positioning used in WinForms.

You can create UIs for traditional desktop applications as well as for web-based applications with WPF. Although initially limited to Internet Explorer for web-based applications, the introduction of Microsoft's Silverlight enables WPF to go cross-browser. You can find more information on Silverlight at `www.microsoft.com/silverlight/`.

WPF offers the ability to truly separate the UI from the business logic of your application. Although we will be working with WPF in Visual Studio 2008, there are additional Microsoft and other third-party tools for creating UIs in WPF. In particular, Microsoft's Expression Blend (as part of Expression Studio) offers a powerful graphical tool for creating user interface designs. More information on Microsoft's Expression products can be obtained from `www.microsoft.com/expression`.

In this chapter, you will learn how to do the following:

- Create a simple WPF application
- Data-bind controls in WPF
- Use a data template to control data presentation

Introducing XAML

Extensible Application Markup Language (XAML) is the XML language used to describe the user interface definition. When you create a WPF-based UI, it is written into an XAML file. You can create and edit XAML with a simple text editor such as Notepad if you wish, or use a more sophisticated tool such as XML Notepad 2007, available from `www.microsoft.com/downloads/details.aspx?familyid = 72 d6aa49-787 d-4118-ba5f-4f30fe913628`. When working with Visual Studio 2008, you have the option to write the code behind your UIs, creating functionality along with your user interface; you also have the ability to create and attach any of the full range of resources available through Visual Studio.

Documents created with XAML have the file extension `.xaml`. In Visual Studio 2008, the design surface for XAML documents is a tabbed panel where you can easily swap between the graphical user interface (GUI) view and source code view of your work.

When projects containing XAML files are compiled, the XAML is converted into Binary Application Markup Language (BAML) before being included in the assembly. The main purpose of this is to improve application performance because compiled BAML will process much faster than raw XAML. From a development point of view, it is not necessary to have an intimate understanding of BAML because you will be working mainly in XAML. It is, however, possible to manually compile your XAML into BAML by using the Windows Application Compiler that ships with the .NET Framework, if you are so inclined.

XAML is very flexible, and can be used to represent everything from layout panels, controls, and graphics to 3D representations and animations.

The following code snippet gives the basic format of an XAML page. This is the typical code skeleton that is generated when you create a new WPF window in Visual Studio 2008. Most of the formatting and layout code will go between the `<Grid>` tags.

```
<Window x:Class="Window3" _
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window3" Height="300" Width="300">
    <Grid>

    </Grid>
```

The following XAML snippet demonstrates the representation for a TextBox control. Normally, if we were to drop a TextBox control onto the XAML page shown in the preceding code, the following code would appear between the `<Grid>` tags:

```
<TextBox Height="21" Margin="56,106,102,0" Name="TextBox1" _
 VerticalAlignment="Top" BorderThickness="2">
   <TextBox.BitmapEffect>
      <EmbossBitmapEffect />
   </TextBox.BitmapEffect>
</TextBox>
```

This example of the TextBox control not only includes some basic properties for the control such as `Name` and `Height`, but also includes a more sophisticated visual effect property: `EmbossBitmapEffect`.

You can also use XAML to generate simple shapes (primitives) such as circles and rectangles. The following code snippet demonstrates how you can use XAML to generate a simple rectangle:

```
<Rectangle Height="74" Margin="20,0,64,27" Name="Rectangle1" _
 Stroke="Black" VerticalAlignment="Bottom" Fill="Red">
   <Rectangle.BitmapEffect>
      <DropShadowBitmapEffect />
   </Rectangle.BitmapEffect>
</Rectangle>
```

In this example, the rectangle has a red fill color and black border. We have also included a drop shadow effect. These effects were all achieved by simply modifying the Properties window for the TextBox and Rectangle controls. However, many of WPF's properties are available to be used across virtually all the controls, even if they do not appear in the Properties window. For example, you can rewrite the code snippet for the rectangle to modify the `Fill` property to create a linear gradient from red to blue, as follows:

```
<Rectangle Height="74" Margin="20,0,64,27" Name="Rectangle1" _
 Stroke="Black" VerticalAlignment="Bottom" >
    <Rectangle.BitmapEffect>
       <DropShadowBitmapEffect />
    </Rectangle.BitmapEffect>
    <Rectangle.Fill>
       <LinearGradientBrush>
          <GradientStop Color="Red" Offset="0"/>
          <GradientStop Color="Blue" Offset="1"/>
       </LinearGradientBrush>
    </Rectangle.Fill>
 </Rectangle>
```

You cannot achieve this effect through the available settings in the Properties window for the Rectangle — it needs to be typed directly into the XAML.

A number of these additional properties can be picked up from using Microsoft's Expression Blend tool to create XAML pages and examining the code generated. Refer to the ''Expression Blend Overview'' section later in this chapter.

Introducing the WPF Controls

When you first open a WPF project in Visual Studio 2008, you will notice that the Toolbox contains a very similar set of controls to those found in the other development environments. However, although most of the controls appear similar, they can behave very differently.

This section introduces some of the core controls and lists some of the fundamental differences between them and their traditional Windows counterparts.

One of the first things you will notice when opening up Visual Studio 2008 to a WPF page is that the page has a magnification control in the top-left corner, and resizing the page automatically resizes any controls on the page as well. Controls stay placed on the page relative to each other, rather than fixed into some position relative to the page itself. Controls also behave differently from the traditional Windows Forms model as you move them around the page, and the underlying form (or rather *window*) itself looks a little different.

To create a new WPF project, choose File ➤ New Project. In the New Project dialog box, choose WPF Application. Keep the default name and click OK. In keeping with the idea that you are designing an interface separated from the business logic of your application, the default designer opens to `Window1.xaml`.

If you examine the Toolbox on the left side of Visual Studio, you will see many familiar items from the Standard Toolbox there. However, as you will see, some of them exhibit different behavior. Note that all the WPF controls differ from their WinForms cousins in the wide range of style and formatting properties that are available to them. Table 5.1 gives a brief overview of the roles of some of the new controls and some changes in the existing controls. This is not an exhaustive list of the controls, but it does look at some key ones.

TABLE 5.1: The WPF Standard Controls

WPF CONTROL	FEATURES
Border	Creates a border around an object. A broad range of properties exists to enable you to change the appearance of the border.
Button	Apart from the obvious and extensive range of style properties common to virtually all the WPF controls, the Button control looks and behaves as you would expect. Double-clicking the control opens to a code skeleton for the `Button_Click` event handler in code-behind (typically `Window1.xaml.vb`).
Grid	Defines a tabular area of columns and rows. Useful for displaying data. One of the main layout controls.
Label	Displays data via the `Content` property.
StackPanel	Arranges child components (other controls) either horizontally or vertically. One of the main layout controls.
TextBox	Unlike the Label control, TextBox continues to use the `Text` property to display data.
Canvas	Provides an area where you can explicitly position controls relative to the position of the Canvas control.
DockPanel	Provides a space where you arrange controls vertically or horizontally relative to each other. One of the main layout controls.
InkCanvas	Sets up a drawing surface within your application.
MediaElement	Contains audio or video content.
TextBlock	Displays small chunks of text.
UniformGrid	Creates a grid with fixed cell size.
WrapPanel	Displays elements sequentially (depending on the orientation property) and wraps to the next line where necessary.

Next we will create a couple of simple applications to demonstrate the functionality of the WPF controls.

Simple ''Hello World'' WPF Application

In this section, you will come to grips with setting up a basic WPF application and using the controls by creating a simple ''Hello World'' application.

Open Visual Studio 2008 and complete the following steps:

1. Choose File ➤ New Project. From the New Project dialog box, choose WPF Application and rename the project **MyWpfApplication1**. Click OK.
2. The Designer should open to `Window1.xaml` in Split mode (Design And Source Code). If you need more space, you can collapse the bottom pane (usually the XAML pane, although this can be swapped around) and tab between the two views.
3. Drag a Button control and a Label control onto your form. Move the controls around on the form to see how the positioning and resize properties work. The <`Grid`> tags in XAML view should now contain the following (without the line breaks and with variations in the `Height` and `Margin` properties):

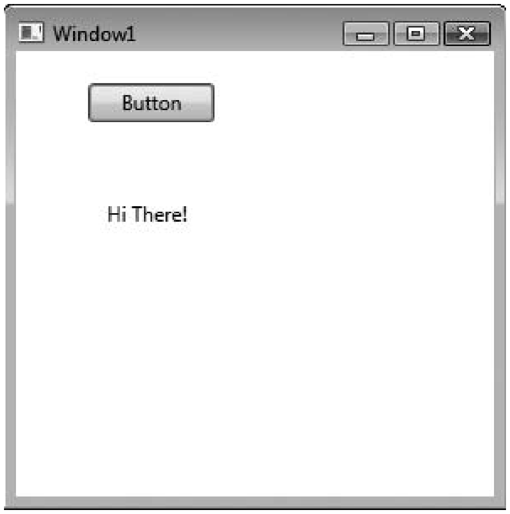
```
<Grid>
   <Button Height="23" HorizontalAlignment="Left" Margin="43,19,0,0" _
 Name="Button1" VerticalAlignment="Top" Width="75">Button</Button>
   <Label Height="23" Margin="49,83,109,0" Name="Label1" _
 VerticalAlignment="Top">Label</Label>
</Grid>
```

4. In Design mode, double-click the Button control to enter code-behind. This should open up to `Window1.xaml.vb` with a `Button1_Click` event code skeleton. Compete the `Button1_Click` event with the following code:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As _
 System.Windows.RoutedEventArgs) Handles Button1.Click
   Label1.Content = "Hi There!"
End Sub
```

Press F5 to run the application. Figure 5.1 illustrates the running application in the Designer window after `Button1` has been clicked.

FIGURE 5.1
The running MyWpfApplication



Simple Drawing Program

This next example illustrates how you can easily create a simple drawing program in WPF with a few controls and very little code. In this example, we will use the InkCanvas control to create the drawing surface. The control will be added at runtime, enabling us to dynamically alter properties such as pen color. We will use a ComboBox control to set the pen color and a Button control to clear the screen.

Begin by opening Visual Studio 2008 and choosing File ➤ New Project. Then complete the following steps:

1. From the New Project dialog box, choose WPF Application and rename the project **WpfDraw**. Click OK.

2. This should open `Window.xaml` in Split mode. From the Standard Toolbox, drop a StackPanel control onto the `Window1` form on the Design surface and set the `Margin` property of the StackPanel (in the Properties box) to ''0,0,0,25''. This `Margin` property will extend the StackPanel to the top, left, and right borders of the form. The margin will leave a 25-pixel (px) gutter at the bottom of the form, where we can place a Button control. Keep the default name for the control of `StackPanel1`.

3. Set the `VerticalAlignment` property of `StackPanel1` to Top.

4. From the Standard Toolbox, drop a ComboBox control into the `StackPanel1` control. In the Properties window for the ComboBox control, set the `HorizontalAlignment` property to Left. Keep the default name of `ComboBox1`. The ComboBox should be located in the top-left corner of `Window1`.

5. In the XAML window, adjust the entry for `ComboBox1` to read as shown here. Keep any of the illustrated line breaks on a single line. This will create the pen color items in the ComboBox.

```
<ComboBox Height="25" Name="ComboBox1" Width="120" _
 HorizontalAlignment="Left">
   <ComboBoxItem IsSelected="True">Red Pen</ComboBoxItem>
   <ComboBoxItem>Green Pen</ComboBoxItem>
   <ComboBoxItem>Blue Pen</ComboBoxItem>
</ComboBox>
```

6. From the Standard Toolbox, drop a Button control onto `Window1` in the space under the StackPanel. Keep the default name of `Button1`. In the Properties window for `Button1`, set the `Height` property to 25, the `VerticalAlignment` property to Bottom, the `HorizontalAlignment` to Left, and the `Content` property to Clear Screen.

7. The final XAML markup for `Button1` should be similar to the following snippet (ignore the line break):

```
<Button Height="25" Name="Button1" Width="75" HorizontalAlignment="Left" _
 VerticalAlignment="Bottom">Clear Screen</Button>
```

Listing 5.1 gives the full XAML markup for this stage of the development.

LISTING 5.1: Full XAML Markup for WpfDraw

```
<Window x:Class="Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="300" Width="300">
   <Grid>
      <StackPanel Name="StackPanel1" Margin="0,0,0,25" _
 VerticalAlignment="Top">
         <ComboBox Height="25" Name="ComboBox1" Width="120" _
 HorizontalAlignment="Left">
            <ComboBoxItem IsSelected="True">Red Pen</ComboBoxItem>
            <ComboBoxItem>Green Pen</ComboBoxItem>
            <ComboBoxItem>Blue Pen</ComboBoxItem>
         </ComboBox>
      </StackPanel>
      <Button Height="25" Name="Button1" Width="75" _
HorizontalAlignment="Left" VerticalAlignment="Bottom"> _
Clear Screen</Button>
    </Grid>
</Window>
```

The next step is to add the code-behind for the application. In the Design window, double-click the `Button1` control to enter code-behind (`Window1.xaml.vb`). Continue with the following steps:

1. Directly under the Class Window declaration, add the following line of code to declare an instance of the InkCanvas control:

   ```
   Dim myink As New InkCanvas
   ```

2. Select the `Window1_Loaded` code skeleton (choose Class Name ➢ Window Events, and Method Name ➢ Loaded).
3. Add the following line of code to the `Window_Loaded` skeleton. This will add the InkCanvas control to the StackPanel control in `Window1` when the window loads:

   ```
   StackPanel1.Children.Add(myink)
   ```

4. In the `Button1_Click` event handler, add the following line of code. This will enable the user to erase any drawings he has created:

   ```
   myink.Strokes.Clear()
   ```

5. Select the code skeleton for `ComboBox1_SelectionChanged` and add the following snippet. This code will set the pen color of the InkCanvas control depending on the selected color in the ComboBox:

```
If ComboBox1.SelectedIndex = 0 Then
      myink.DefaultDrawingAttributes.Color = Colors.Red
   ElseIf ComboBox1.SelectedIndex = 1 Then
      myink.DefaultDrawingAttributes.Color = Colors.Green
   ElseIf ComboBox1.SelectedIndex = 2 Then
      myink.DefaultDrawingAttributes.Color = Colors.Blue
End If
```

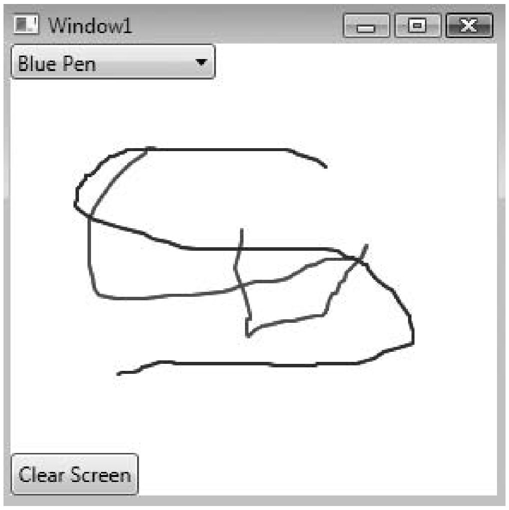
Listing 5.2 gives the full code-behind listing for WpfDraw.

Listing 5.2: Full Code-Behind Listing for WpfDraw

```
Class Window1
  Dim myink As New InkCanvas

  Private Sub Window1_Loaded(ByVal sender As Object, ByVal e As_
 System.Windows.RoutedEventArgs) Handles Me.Loaded
     StackPanel1.Children.Add(myink)
  End Sub

  Private Sub Button1_Click(ByVal sender As System.Object, _
 ByVal e As System.Windows.RoutedEventArgs) Handles Button1.Click
    myink.Strokes.Clear()
  End Sub

  Private Sub ComboBox1_SelectionChanged(ByVal sender As _
System.Object, ByVal e As _
System.Windows.Controls.SelectionChangedEventArgs) Handles _
ComboBox1.SelectionChanged
  If ComboBox1.SelectedIndex = 0 Then
     myink.DefaultDrawingAttributes.Color = Colors.Red
  ElseIf ComboBox1.SelectedIndex = 1 Then
   myink.DefaultDrawingAttributes.Color = Colors.Green
  ElseIf ComboBox1.SelectedIndex = 2 Then
   myink.DefaultDrawingAttributes.Color = Colors.Blue
  End If
 End Sub
End Class
```

Press F5 to test the application. Figure 5.2 illustrates the running application.

FIGURE 5.2
The running WpfDraw



Data-Binding WPF Controls

The ability to bind WPF controls to external (and internal) data sources is an important aspect of being able to separate UI design from business logic and functionality.

WPF controls can be bound to data sources as well as style sources. This way, you can set up a set of styles for your application and then bind each control to those styles, thus enabling centralized management of the look and feel of your user interfaces.

There are many options available to the developer when connecting to data with WPF. WPF offers a flexible and powerful framework for data connectivity and management. A full discussion is beyond the scope of this chapter; refer to the Microsoft documentation for a more detailed view. The article titled ''Data Binding Overview'' (search the Help documentation) is a good start.

In this section, you will see how to carry out three basic data-binding tasks with WPF. You will bind controls to an array, to a data template, and to a database.

Data-Binding Example 1: Binding to an Array and a Data Template

In this example, we will set up a simple class called Contacts to hold `Surname` and `FirstName` elements. We will then use an array of contacts to hold some data that can be accessed from our XAML page. The presentation of the data in the XAML page will be defined by a data template. To keep things simple, we will do everything within the one project.

Begin by opening Visual Studio 2008 and choosing File ➢ New Project. Then complete the following steps:

1. From the New Project dialog box, choose WPF Application and rename the project **WpfBinding1**. Click OK.
2. We will begin by creating the Contacts class. You should be open to `Window1.xaml` in Split view. From Solution Explorer, click the View Code icon to switch to code-behind

(Window1.xaml.vb). Alternatively, you can double-click on the Window1 heading in the Window1 form in the Designer.

3. In code-behind, add the following code for the Contacts class directly under the Class Window1 declaration. The Contacts class has two properties: FirstName and Surname. You will find that as you type the following code, much of the code skeleton is automatically generated by Visual Studio:

```
Private Class contacts
   Dim _name As String
   Dim _surname As String
   Public Sub New(ByVal FirstName As String, ByVal Surname As String)
      _name = FirstName
      _surname = Surname
   End Sub
   Public ReadOnly Property FirstName() As String
      Get
         Return _name
      End Get
   End Property
   Public ReadOnly Property Surname() As String
      Get
         Return _surname
      End Get
   End Property
 End Class
```

4. Select the Window1_Loaded code skeleton (choose Class Name ➤ Window Events, and Method Name ➤ Loaded).

5. Add the following code to the Window1_Loaded skeleton. This code declares an array of contacts named person and adds some names to it. The final line uses the DataContext property of Window1 to bind the person array to Window1.

```
Dim person As New ArrayList
   person.Add(New contacts("Fred", "Bloggs"))
   person.Add(New contacts("Betty", "Smith"))
   person.Add(New contacts("Jane", "Doe"))
   person.Add(New contacts("Bill", "Jones"))
   person.Add(New contacts("Jenny", "Day"))

Me.DataContext = person
```

This completes the code-behind for this project.

The next set of steps involves setting up the XAML for the project. For this example, we will write directly to the XAML code. You will find that as you continue to work with WPF, it is often easier to work directly with the XAML and use the Designer only to provide a visual check that everything is going together as it should.

1. Switch back to XAML view for `Window1.xaml`. The default XAML should look similar to the following snippet:

```
<Window x:Class="Window4"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window4" Height="300" Width="300" Name="Window1">
    <Grid>

    </Grid>
</Window>
```

2. Our key areas of interest here are the `<Grid>` tags. Add a name attribute to the `<Grid>` tag to read `<Grid Name = ''MyGrid''>`.

3. The next step is to add the data template. This is contained within `<GridResources>` tags, which sit within the `<Grid Name = ''MyGrid''>` tags. Create the following snippet:

```
<Grid.Resources>
   <DataTemplate x:Key="NameStyle">
      <Grid>
         <Grid.ColumnDefinitions>
            <ColumnDefinition Width="60" />
            <ColumnDefinition Width="*" />
         </Grid.ColumnDefinitions>
         <TextBlock Grid.Column="0" Text="{Binding Path=FirstName}" />
         <TextBlock Grid.Column="1" Text="{Binding Path=Surname}" />
      </Grid>
   </DataTemplate>
</Grid.Resources>
```

The purpose of this code is to define a data template that can be referenced by using its key, `NameStyle`. The data template defines a grid with two columns — the first column is 60px wide, and the second column occupies the remainder of the available space. Attached to the first column is a TextBlock control that has its `Text` property bound to the `FirstName` element of the `person` array that we declared in the code-behind. Remember that we used the `DataContext` property to attach `person` to `Window1`. Similarly, the second TextBlock has its `Text` property bound to `Surname`, and the control is attached to the second column of the grid.

4. Add the following snippet directly below the `<Grid.Resources >` . . . `< /Grid.Resources>` section. Do not include the line breaks:

```
<TextBlock Text="Current Selection = "  />
<TextBlock Text="{Binding Path=FirstName}" Margin="110,0,0,0"  />
<ListBox Margin="0,40,0,0" _
  ItemTemplate="{StaticResource NameStyle}" ItemsSource="{Binding} " _
  IsSynchronizedWithCurrentItem="true" Name="ListBox1"  />
```

The purpose of this code is to provide the content controls (two TextBlocks and a ListBox) to present the data on the window. We have used the `Margin` property to control the layout of the controls. In this instance, the use of `Margin` is suitable because we wish to simplify the code, but laying out in this manner is limiting if you wish to alter text content or styles at a later date. There are a number of alternative methods for achieving the same layout using combinations of the layout controls, which provide a little more flexibility. For example, the following snippet achieves the same result by using Grid rows and a nested StackPanel control to manage the layout of the content controls:

```
<Grid.RowDefinitions>
   <RowDefinition Height="40" />
   <RowDefinition Height="*" />
</Grid.RowDefinitions>
<StackPanel Orientation="Horizontal"  Grid.Row="0">
   <TextBlock Text="Current Selection = "  />
   <TextBlock Text="{Binding Path=FirstName}"  />
</StackPanel>

<ListBox  Grid.Row="1" _
  ItemTemplate="{StaticResource NameStyle}" ItemsSource="{Binding} " _
  IsSynchronizedWithCurrentItem="true" Name="ListBox1"  />
```

The second TextBlock uses `Binding` to data-bind the control to the `FirstName` field of the `person` array. The ListBox uses the `StaticResource` statement to bind to the `NameStyle` data layout. By simply writing *Binding*, we indicate that the control is bound to the data source (in this case, the `person` array) attached to the parent container (`Window1`). In this way, the ListBox will display the full set of records in the array. The `NameStyle` data layout ensures that each record is displayed as a `FirstName`, `Surname` combination by using two TextBlock controls set up in adjacent Grid columns.

Listing 5.3 gives the full XAML source for this project. Delete the line breaks.

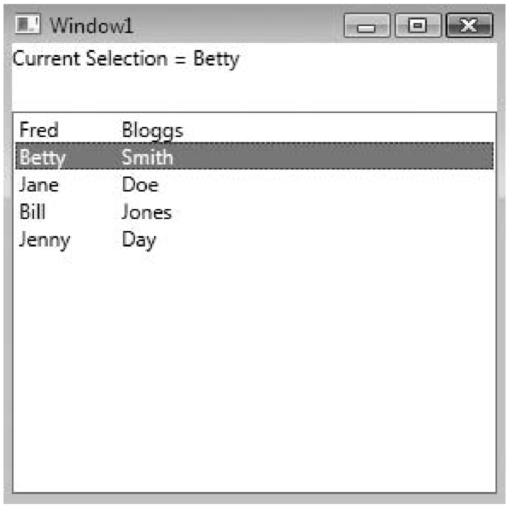
Listing 5.3: Full XAML Source Code for the WpfBinding1 Project

```
<Window x:Class="Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300">

    <Grid Name="myGrid">
        <Grid.Resources>
            <DataTemplate x:Key="NameStyle">
                <Grid>
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="60" />
                        <ColumnDefinition Width="*" />
                    </Grid.ColumnDefinitions>
```

```
                        <TextBlock Grid.Column="0" Text="{Binding _
                          Path=FirstName}" />
                        <TextBlock Grid.Column="1" _
                          Text="{Binding Path=Surname}" />
                    </Grid>
                </DataTemplate>
            </Grid.Resources>

            <Grid.RowDefinitions>
                <RowDefinition Height="40" />
                <RowDefinition Height="*" />
            </Grid.RowDefinitions>
            <StackPanel Orientation="Horizontal"  Grid.Row="0">
                <TextBlock Text="Current Selection = "  />
                <TextBlock Text="{Binding Path=FirstName}"  />
            </StackPanel>

            <ListBox  Grid.Row="1"
              ItemTemplate="{StaticResource NameStyle}" _
    ItemsSource="{Binding} " IsSynchronizedWithCurrentItem="true" _
    Name="ListBox1"  />

        </Grid>
    </Window>
```

Finish up by testing the application. Making a selection in the list should be reflected in the Current Selection TextBlock. The running application is shown in Figure 5.3.

FIGURE 5.3
The running WpfBinding1 project



Current Selection = Betty

Fred	Bloggs
Betty	Smith
Jane	Doe
Bill	Jones
Jenny	Day

Data-Binding Example 2: Binding to a Database

In this next example, we will see how to connect the WPF page created in the previous example to a database. This project not only demonstrates the technique of connecting to a database, but also emphasizes the way that WPF enables the developer to separate the user interface aspects of a project from the data and business logic.

Begin by opening the previous project, WpfBinding1. Continue with the following steps:

1. First, we will use SQL Server Express to create a new database. From the Project menu, choose Add New Item.
2. From the Add New Item dialog box, select Service-Based Database. Name the database **`Contacts.mdf`** and click the Add button.
3. You will be presented with the Data Source Configuration Wizard with a message saying that the database does not contain any objects. Keep the default name: ContactsDataSet. Click the Finish button to create an empty dataset and close the wizard. We will return to the dataset later to complete it.
4. Over in the Toolbox area, click the Server Explorer tab. Under the Data Connections tree, expand the entry for `Contacts.mdf`. Right-click the Tables entry and choose Add New Table from the context menu.
5. The new database table should now be open in the Designer window. Set up the database fields as shown in Table 5.2.

TABLE 5.2: Database Fields for *Contacts.mdf*

Column Name	Data Type
ID	nchar(10)
FirstName	nchar(10)
Surname	nchar(10)

6. Right-click the ID entry and choose Set Primary Key from the context menu.
7. Click the Save button on the Standard Toolbar. A Choose Name dialog box should open for the table. Enter **Customers** and click OK. The Customers table should now be visible in the Tables entry of `Contacts.mdf` in the Data Connections tree in Server Explorer.
8. Over in Solution Explorer, double-click the entry for `ContactsDataSet.xsd` to open the dataset in the Designer window.
9. Return to the Server Explorer. Drag the Customers table from the Data Connections tree in Server Explorer onto the Design surface for the dataset. This should set up the Customers DataTable in the dataset and also establish a CustomersTableAdapter.

10. Save your work.

11. In Server Explorer, right-click the Customers table in the Data Connections tree and choose Show Table Data.

12. Enter the data from Table 5.3 into the Customers table.

TABLE 5.3: Data Entries for Customers Table

ID	FirstName	Surname
1	Fred	Bloggs
2	Wilma	Smith
3	Bill	Green

13. You are now ready to attach the database to `Window1.xaml`. We will do this in code-behind for `Window1.xaml`. Double-click the entry for `Window1.xaml.vb` in Solution Explorer.

14. Add the following snippet to the code screen just under the entry for Class Window1. This code declares instances of the ContactsDataSet and the CustomersTableAdapter.

```
Dim myDataset As New ContactsDataSet
Dim myCustomersAdapter As New _
    ContactsDataSetTableAdapters.CustomersTableAdapter
```

15. In the `sub` for `Window1_Loaded`, add the following code snippet. The purpose of the first of these lines is to use the myCustomersAdapter to load the data from the Contacts database into myDataset. The second line attaches the dataset as the data source for `Window1`:

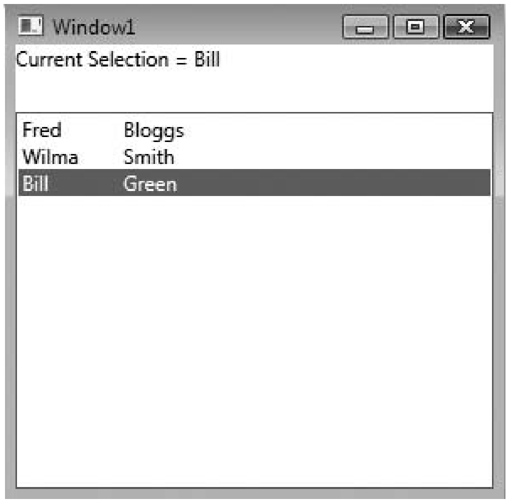
```
myCustomersAdapter.Fill(myDataset.Tables("Customers"))
Me.DataContext = myDataset.Tables("Customers")
```

16. Finally, comment out (or delete) the line of code attaching the original `person` array to `Window1`: `'Me.DataContext = person`.

The field names in the dataset are the same as the ones used in the original array, so we can run the application without needing to change `Window1.xaml`. Note that you can now delete or comment out any of the code that we originally used to create the `person` array or contacts class.

This project is now completed. Test the application by pressing F5 or clicking the green arrow. The running version should appear as shown in Figure 5.4.

FIGURE 5.4
The updated WpfBinding1 application



Current Selection = Bill

Fred	Bloggs
Wilma	Smith
Bill	Green

Creating a WPF Browser Application

WPF applications can also be displayed as web applications. The XAML for such an application is virtually identical to its desktop equivalent. You may have to make only some minor layout and presentation changes to optimize the appearance of your application in a web browser window as opposed to on the desktop.

The main limitation to this technology is that such applications are restricted to Internet Explorer 6 and above unless you create a Silverlight implementation. Microsoft is currently releasing Silverlight as a cross-platform, cross-browser plug-in. Refer to `www.microsoft.com/silverlight` for more information.

In this example, we will create a WPF browser version of the simple drawing program, WpfDraw, that we created earlier in this chapter.

1. Start Visual Studio 2008 and begin by choosing File ➤ New Project.
2. In the New Project dialog box, choose WPF Browser Application. Keep the default name of WpfBrowserApplication1 and click OK.
3. This will open up `Page1.xaml` in Split mode. Switch to XAML view and add the code from Listing 5.4 (without the line breaks). This is essentially the same markup that we used for WpfDraw earlier in the chapter, with some layout modifications to control the distribution of the controls and make the application a little more presentable in a web browser window.

LISTING 5.4: XAML for WpfBrowserApplication1

```
<Page x:Class="Page1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Page1">
   <Grid>
      <StackPanel Margin="25">
         <StackPanel Name="StackPanel1" Margin="0,0,0,0" _
```

```
 VerticalAlignment="Top">
               <ComboBox Height="25" Name="ComboBox1" Width="120" _
 HorizontalAlignment="Left">
                   <ComboBoxItem IsSelected="True">Red Pen _
</ComboBoxItem>
                   <ComboBoxItem>Green Pen</ComboBoxItem>
                   <ComboBoxItem>Blue Pen</ComboBoxItem>
               </ComboBox>
           </StackPanel>
           <Button  Height="25" Name="Button1" Width="75" _
HorizontalAlignment="Left" VerticalAlignment="Bottom">Clear Screen _
</Button>
       </StackPanel>
    </Grid>
</Page>
```

4. The code-behind for WpfBrowserApplication1 remains virtually unchanged from WpfDraw as well. Use the Solution Explorer to switch to code-behind (`Page1.xaml.vb`) and add the code from Listing 5.5. The only difference with this code from the original WpfDraw is that we add some additional layout and design properties to `myink` in the `Page1_Loaded` sub to again help with presentation in the web browser window.

LISTING 5.5: Code-Behind for WpfBrowserApplication1

```
Class Page1
   Dim myink As New InkCanvas

   Private Sub Page1_Loaded(ByVal sender As Object, _
ByVal e As System.Windows.RoutedEventArgs) Handles Me.Loaded
      StackPanel1.Children.Add(myink)
      myink.Background = Brushes.Cornsilk
      myink.HorizontalAlignment = Windows.HorizontalAlignment.Left
      myink.Height = 400
      myink.Width = 400
   End Sub

   Private Sub Button1_Click(ByVal sender As System.Object, _
 ByVal e As System.Windows.RoutedEventArgs) Handles Button1.Click
      myink.Strokes.Clear()
   End Sub
   Private Sub ComboBox1_SelectionChanged(ByVal sender As  _
System.Object, ByVal e As  _
System.Windows.Controls.SelectionChangedEventArgs) Handles _
ComboBox1.SelectionChanged
      If ComboBox1.SelectedIndex = 0 Then
         myink.DefaultDrawingAttributes.Color = Colors.Red
      ElseIf ComboBox1.SelectedIndex = 1 Then
         myink.DefaultDrawingAttributes.Color = Colors.Green
```

```
            ElseIf ComboBox1.SelectedIndex = 2 Then
                myink.DefaultDrawingAttributes.Color = Colors.Blue
            End If
        End Sub

    End Class
```

Press F5 or click the green arrow to run the application. It will open in a web browser window and should appear and function as shown in Figure 5.5.

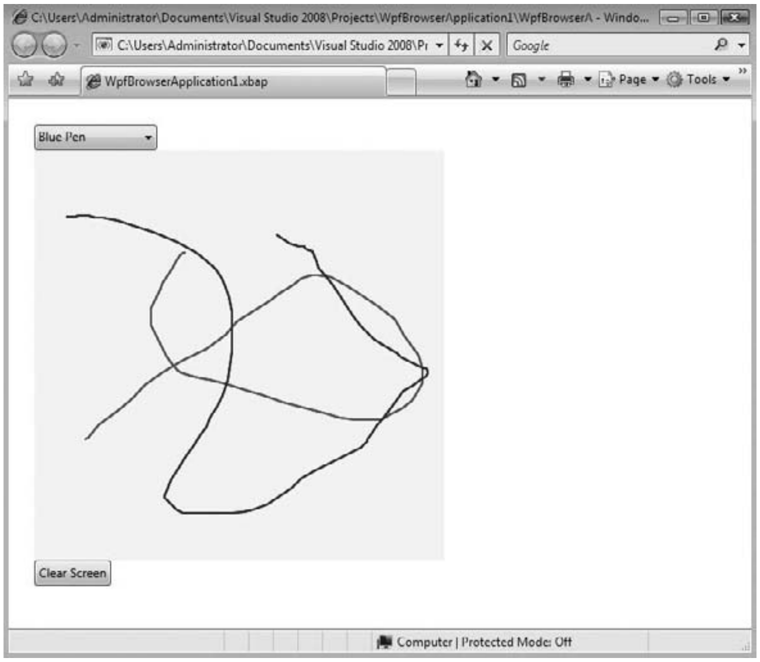

C:\Users\Administrator\Documents\Visual Studio 2008\Projects\WpfBrowserApplication1\WpfBrowserA - Windo...
C:\Users\Administrator\Documents\Visual Studio 2008\Pr
Google
WpfBrowserApplication1.xbap
Page
Tools
Blue Pen
Clear Screen
Computer | Protected Mode: Off

FIGURE 5.5
The running WpfBrowserApplication1

Expression Blend Overview

Expression Blend is part of Microsoft's new Expression Studio package. Expression Studio contains four major components:

Expression Web	Website designer
Expression Blend	User interface designer
Expression Design	Vector and bitmap graphics editor
Expression Media	Media manager

Each component can be purchased independently or as part of the Expression Studio package. A fifth component, Expression Encoder, is currently available as an independent purchase. Expression Encoder encodes media content into a format suitable for Silverlight.

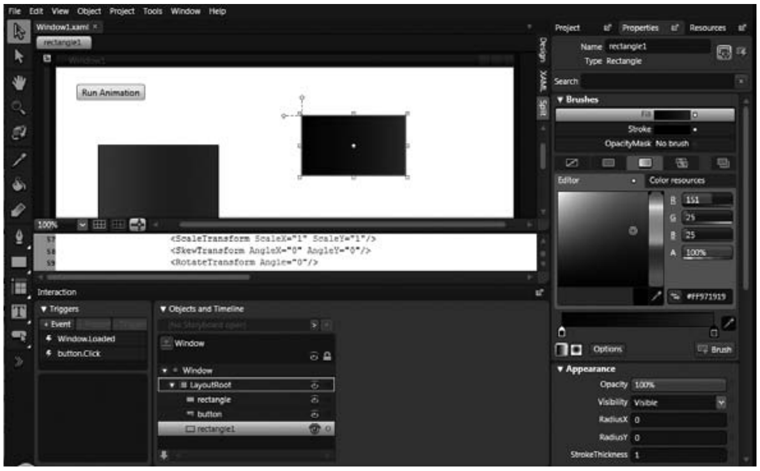
More information and trial downloads can be obtained from the Microsoft website at `www.microsoft.com/expression/expression-studio/overview.aspx`.

Expression Blend is specifically designed as a tool for creating XAML-based (WPF) interfaces. It has a much stronger graphical focus on design than the WPF tools in Visual Studio 2008 and would suit developers who are more focused on visual design than working with code. Expression Blend also exposes a greater range of tools and properties at the GUI level than Visual Studio 2008, and as such is a great way to explore the possibilities of UI design with WPF. For example, if you are interested in animation with WPF, and want to work with Visual Studio 2008 but are struggling with the documentation, you can create a simple animation in Expression Blend using the graphical tools, examine the XAML markup that is generated, and transfer it to Visual Studio.

A demonstration version of Expression Blend can be downloaded from the Microsoft website. A number of samples ship with the package that you can open and work with.

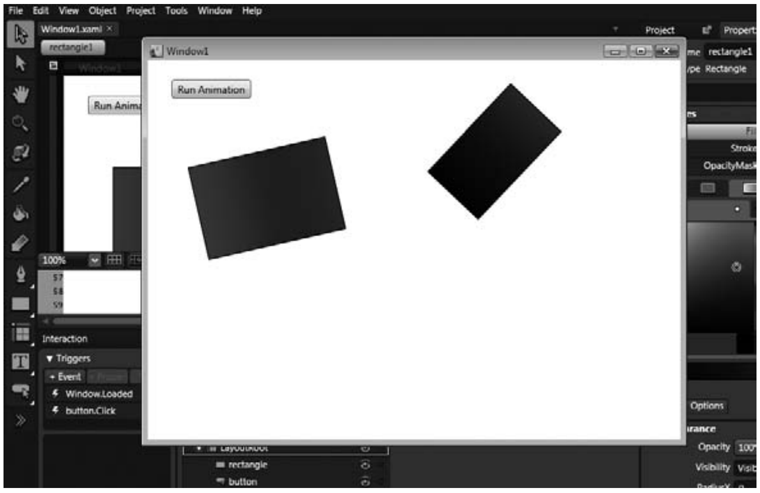
Figure 5.6 illustrates the interface for Expression Blend with a simple animation project. Figure 5.7 illustrates the simple animation project shown in Figure 5.6, running in a test window.

FIGURE 5.6
Interface for Expression Blend



```
<ScaleTransform ScaleX="1" ScaleY="1"/>
<SkewTransform AngleX="0" AngleY="0"/>
<RotateTransform Angle="0"/>
```

FIGURE 5.7
Expression Blend with running project



We can extract the markup for the animated rectangle from the XAML generated in Expression Blend and use it in a Visual Studio 2008 WPF project. Two sections of code are used to generate the animation. The first section is the code defining the rectangle and its behavior, as illustrated in the following code snippet. This code includes the red-to-blue fill gradient that was illustrated earlier in the chapter (remove the line breaks):

```
<Rectangle HorizontalAlignment="Left" Margin="57,106,0,0" _
 VerticalAlignment="Top" Width="165" Height="111" _
 Stroke="#FF000000" RenderTransformOrigin="0.5,0.5" _
 x:Name="rectangle">
   <Rectangle.Fill>
      <LinearGradientBrush EndPoint="1,0.5" StartPoint="0,0.5">
         <GradientStop Color="Red" Offset="0"/>
         <GradientStop Color="Blue" Offset="1"/>
      </LinearGradientBrush>
   </Rectangle.Fill>
   <Rectangle.RenderTransform>
      <TransformGroup>
         <ScaleTransform ScaleX="1" ScaleY="1"/>
         <SkewTransform AngleX="0" AngleY="0"/>
         <RotateTransform Angle="0"/>
         <TranslateTransform X="0" Y="0"/>
      </TransformGroup>
   </Rectangle.RenderTransform>
</Rectangle>
```

The second code chunk used with the animation is the one generated to handle the actual time line of the animation and the trigger used to start it. In this example, the code has been doctored slightly to animate only a single rectangle and to be used in a `Page.xaml` file in a WPF browser application. The trigger to fire the animation in this case is the loading of the page. The time line information is managed in the `<Page.Resources>` tags while the triggers are handled by the `<Page.Triggers>` tags. The code is illustrated in the following snippet (remove the line breaks):

```
<Page.Resources>
   <Storyboard x:Key="Timeline1">
      <DoubleAnimationUsingKeyFrames BeginTime="00:00:00" _
Storyboard.TargetName="rectangle" _
Storyboard.TargetProperty="(UIElement.RenderTransform). _
(TransformGroup.Children)[2].(RotateTransform.Angle)">
         <SplineDoubleKeyFrame KeyTime="00:00:05" Value="500"/>
      </DoubleAnimationUsingKeyFrames>
   </Storyboard>
</Page.Resources>
<Page.Triggers>
   <EventTrigger RoutedEvent="FrameworkElement.Loaded">
      <BeginStoryboard Storyboard="{StaticResource Timeline1}"/>
   </EventTrigger>
</Page.Triggers>
```

To test this code in a WPF browser application, create a new WPF Browser Application project. Copy the contents of the <`Page.Resources`> and <`Page.Triggers` > code from the preceding snippet into the XAML for `Page1.xaml`, immediately following the class declaration and before the <`Grid`> tags. Copy the first code snippet governing the rectangle and its behavior between the <`Grid`>...</`Grid`> tags. Run the project and you should have a rectangle performing a rotation in a web page. If you play with the attribute values in the <`SplineDoubleKeyFrame`> tag, you can alter the length and extent of the rotation.

A comprehensive user guide is available under the Help menu of the Expression Blend package.

The Bottom Line

Create a simple WPF application. WPF is a new and powerful technology for creating user interfaces. WPF is one of the core technologies in the .NET Framework 3.5 and is integrated into Windows Vista. WPF is also supported on Windows XP. WPF takes advantage of the graphics engines and display capabilities of the modern computer and is vector based and resolution independent.

Master It Develop a simple ''Hello World'' type of WPF application that displays a Button control and Label control. Clicking the button should set the content property of a Label control to `Hi There!`

Data-bind controls in WPF. The ability to bind controls to a data source is an essential aspect of separating the UI from the business logic in an application.

Master It Data-bind a Label control to one field in a record returned from a database on your computer.

Use a data template to control data presentation. WPF enables a very flexible approach to presenting data by using data templates. The developer can create and fully customize data templates for data formatting.

Master It Create a data template to display a Name, Surname, Gender combination in a horizontal row in a ComboBox control. Create a simple array and class of data to feed the application.

Chapter 6

Basic Windows Controls

In previous chapters, we explored the environment of Visual Basic and the principles of event-driven programming, which is the core of VB's programming model. In the process, we briefly explored a few basic controls through the examples. The .NET Framework provides many more controls, and all of them have a multitude of trivial properties (such as `Font`, `BackgroundColor`, and so on), which you can set either in the Properties window or from within your code.

This chapter explores in depth the basic Windows controls: the controls you'll use most often in your applications because they are the basic building blocks of typical rich client-user interfaces. Rather than look at controls' background and foreground color, font, and other trivial properties, we'll look at the properties unique to each control and see how these properties are used in building functional, rich user interfaces.

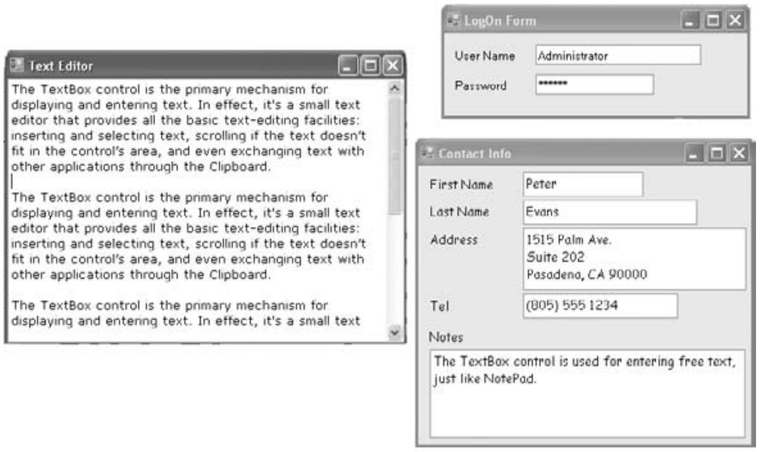
In this chapter, you'll learn how to do the following:

- ◆ Use the TextBox control as a data-entry and text-editing tool
- ◆ Use the ListBox, CheckedListBox, and ComboBox controls to present lists of items
- ◆ Use the ScrollBar and TrackBar controls to enable users to specify sizes and positions with the mouse

The TextBox Control

The TextBox control is the primary mechanism for displaying and entering text. It is a small text editor that provides all the basic text-editing facilities: inserting and selecting text, scrolling if the text doesn't fit in the control's area, and even exchanging text with other applications through the Clipboard.

The TextBox control is an extremely versatile data-entry tool that can be used for entering and editing single lines of text, such as a number or a password, or an entire text file. Figure 6.1 shows a few typical examples. All the boxes in Figure 6.1 contain text — some a single line, some several lines. The scroll bars you see in some text boxes are part of the control. You can specify which scroll bars (vertical and/or horizontal) will be attached to the control, and they will appear automatically whenever the control's contents exceed the visible area of the control.



The TextBox control is the primary mechanism for displaying and entering text. In effect, it's a small text editor that provides all the basic text-editing facilities: inserting and selecting text, scrolling if the text doesn't fit in the control's area, and even exchanging text with other applications through the Clipboard.

The TextBox control is the primary mechanism for displaying and entering text. In effect, it's a small text editor that provides all the basic text-editing facilities: inserting and selecting text, scrolling if the text doesn't fit in the control's area, and even exchanging text with other applications through the Clipboard.

The TextBox control is the primary mechanism for displaying and entering text. In effect, it's a small text

User Name: Administrator
Password: ******

First Name: Peter
Last Name: Evans
Address: 1515 Palm Ave.
Suite 202
Pasadena, CA 90000
Tel: (805) 555 1234
Notes: The TextBox control is used for entering free text, just like NotePad.

FIGURE 6.1
Typical uses of the TextBox control

Basic Properties

Let's start with the properties that specify the appearance and, to some degree, the functionality of the TextBox control; these properties are usually set at design time through the Properties window. Then, we'll look at the properties that allow you to manipulate the control's contents and interact with users from within your code.

TextAlign

This property sets (or returns) the alignment of the text on the control, and its value is a member of the `HorizontalAlignment` enumeration: *`Left`*, *`Right`*, or *`Center`*. The TextBox control doesn't allow you to format text (mix different fonts, attributes, or colors), but you can set the font in which the text will be displayed with the `Font` property, as well as the control's background color with the `BackColor` property.

MultiLine

This property determines whether the TextBox control will hold a single line or multiple lines of text. Every time you place a TextBox control on your form, it's sized for a single line of text and you can change its width only. To change this behavior, set the `MultiLine` property to True. When creating multiline TextBoxes, you will most likely have to set one or more of the `MaxLength`, `ScrollBars`, and `WordWrap` properties in the Properties window.

MaxLength

This property determines the number of characters that the TextBox control will accept. Its default value is 32,767, which was the maximum number of characters the VB 6 version of the control could hold. Set this property to zero, so that the text can have any length, up to the control's capacity limit — 2,147,483,647 characters, to be exact. To restrict the number of characters that the user can type, set the value of this property accordingly.

The `MaxLength` property of the TextBox control is often set to a specific value in data-entry applications, which prevents users from entering more characters than can be stored in a database field. A TextBox control for entering international standard book numbers (ISBNs), for instance, shouldn't accept more than 13 characters.

ScrollBars

This property lets you specify the scroll bars you want to attach to the TextBox if the text exceeds the control's dimensions. Single-line text boxes can't have a scroll bar attached, even if the text exceeds the width of the control. Multiline text boxes can have a horizontal or a vertical scroll bar, or both.

If you attach a horizontal scroll bar to the TextBox control, the text won't wrap automatically as the user types. To start a new line, the user must press Enter. This arrangement is useful for implementing code editors in which lines must break explicitly. If the horizontal scroll bar is missing, the control inserts soft line breaks when the text reaches the end of a line, and the text is wrapped automatically. You can change the default behavior by setting the `WordWrap` property.

WordWrap

This property determines whether the text is wrapped automatically when it reaches the right edge of the control. The default value of this property is True. If the control has a horizontal scroll bar, however, you can enter very long lines of text. The contents of the control will scroll to the left, so the insertion point is always visible as you type. You can turn off the horizontal scroll bar and still enter long lines of text; just use the left/right arrows to bring any part of the text into view. You can experiment with the `WordWrap` and `ScrollBars` properties in the TextPad sample application, which is described later in this chapter.

Notice that the `WordWrap` property has no effect on the actual line breaks. The lines are wrapped automatically, and there are no hard breaks (returns) at the end of each line. Open the TextPad project, enter a long paragraph, and resize the window — the text is automatically adjusted to the new width of the control.

A Functional Text Editor by Design

A TextBox control with its `MaxLength` property set to `0`, its `MultiLine` and `WordWrap` properties set to True, and its `ScrollBars` property set to `Vertical` is, on its own, a functional text editor. Place a TextBox control with these settings on a form, run the application, and check out the following:

- Enter text and manipulate it with the usual editing keys: Delete, Insert, Home, and End.
- Select multiple characters with the mouse or the arrow keys while holding down the Shift key.
- Move segments of text around with Copy (Ctrl+C), Cut (Ctrl+X), and Paste (Ctrl+V, or Shift+Insert) operations.
- Right-click the control to see its context menu; it contains all the usual text-editing commands.
- Exchange data with other applications through the Clipboard.

You can do all this without a single line of code! If you use the `My` object, you can save and load files by using two lines of code. Shortly, you'll see what you can do with the TextBox control if you add some code to your application, but first let's continue our exploration of the properties that allow us to manipulate the control's functionality.

AcceptsReturn, AcceptsTab

These two properties specify how the TextBox control reacts to the Return (Enter) and Tab keys. The Enter key activates the default button on the form, if there is one. The default button is usually an OK button that can be activated with the Enter key, even if it doesn't have the focus. In a multiline TextBox control, however, we want to be able to use the Enter key to change lines. The default value of the `AcceptsReturn` property is True, so pressing Enter creates a new line on the control. If you set it to False, users can still create new lines in the TextBox control, but they'll have to press Ctrl+Enter. If the form contains no default button, the Enter key creates a new line regardless of the `AcceptsReturn` setting.

Likewise, the `AcceptsTab` property determines how the control reacts to the Tab key. Normally, the Tab key takes you to the next control in the Tab order, and we generally avoid changing the default setting of the `AcceptsTab` property. In a multiline TextBox control, however, you may want the Tab key to insert a Tab character in the text of the control instead; to do this, set the control's `AcceptsTab` property to True (the default value is False). If you change the default value, users can still move to the next control in the Tab order by pressing Ctrl+Tab. Notice that the `AcceptsTab` property has no effect on other controls. Users may have to press Ctrl+Tab to move to the next control while a TextBox control has the focus, but they can use the Tab key to move from any other control to the next one.

CharacterCasing

This property tells the control to change the casing of the characters as they're entered by the user. Its default value is *`Normal`*, and characters are displayed as typed. You can set it to *`Upper`* or *`Lower`* to convert the characters to upper- or lowercase automatically.

PasswordChar

This property turns the characters typed into any character you specify. If you don't want to display the actual characters typed by the user (when entering a password, for instance), use this property to define the character to appear in place of each character the user types.

The default value of this property is an empty string, which tells the control to display the characters as entered. If you set this value to an asterisk (*), for example, the user sees an asterisk in the place of every character typed. This property doesn't affect the control's `Text` property, which contains the actual characters. If the `PasswordChar` property is set to any character, the user can't copy or cut the text on the control.

ReadOnly, Locked

If you want to display text on a TextBox control but prevent users from editing it (such as for an agreement or a contract they must read, software installation instructions, and so on), you can set the `ReadOnly` property to True. When `ReadOnly` is set to True, you can put text on the control from within your code, and users can view it, yet they can't edit it.

To prevent editing of the TextBox control with VB 6, you had to set the `Locked` property to True. Oddly, the `Locked` property is also supported, but now it has a very different function. The `Locked` property of VB 2008 locks the control at design time (so that you won't move it or change its properties by mistake as you design the form).

Text-Manipulation Properties

Most of the properties for manipulating text in a TextBox control are available at runtime only. This section presents a breakdown of each property.

Text

The most important property of the TextBox control is the `Text` property, which holds the control's text. You can set this property at design time to display some text on the control initially.

Notice that there are two methods of setting the `Text` property at design time. For single-line TextBox controls, set the `Text` property to a short string, as usual. For multiline TextBox controls, open the `Lines` property and enter the text in the String Collection Editor window, which will appear. In this window, each paragraph is entered as a single line of text. When you're finished, click OK to close the window; the text you entered in the String Collection Editor window will be placed on the control. Depending on the width of the control and the setting of the `WordWrap` property, paragraphs may be broken into multiple lines.

At runtime, use the `Text` property to extract the text entered by the user or to replace the existing text. The `Text` property is a string and can be used as an argument with the usual string-manipulation functions of Visual Basic. You can also manipulate it with the members of the String class. The following expression returns the number of characters in the `TextBox1` control:

```
Dim strLen As Integer = TextBox1.Text.Length
```

The `IndexOf` method of the String class will locate a specific string in the control's text. The following statement returns the location of the first occurrence of the string ***`Visual`*** in the text:

```
Dim location As Integer
location = TextBox1.Text.IndexOf("Visual")
```

For more information on locating strings in a TextBox control, see the section ''VB 2008 at Work: The TextPad Project'' later in this chapter, where we'll build a text editor with search-and-replace capabilities. For a detailed discussion of the String class, see Chapter 13, ''Handling Strings, Characters, and Dates.''

To store the control's contents in a file, use a statement such as the following:

```
StrWriter.Write(TextBox1.Text)
```

Similarly, you can read the contents of a text file into a TextBox control by using a statement such as the following:

```
TextBox1.Text = StrReader.ReadToEnd
```

where `StrReader` and `StrWriter` are two properly declared `StreamReader` and `StreamWriter` variables. File operations are discussed in detail in Chapter 15, ''Accessing Folders and Files.'' You will also find out how to print text files in Chapter 20, ''Printing with Visual Basic 2008.''

To locate all instances of a string in the text, use a loop like the one in Listing 6.1. This loop locates successive instances of the string **`Basic`** and then continues searching from the character following the previous instance of the word in the text. To locate the last instance of a string in the text, use the `LastIndexOf` method. You can write a loop similar to the one in Listing 6.1 that scans the text backward.

Listing 6.1: Locating All Instances of a String in a TextBox

```
Dim startIndex = -1
startIndex = TextBox1.Text.IndexOf("Basic", startIndex + 1)
While startIndex > 0
   Console.WriteLine "String found at " & startIndex
   startIndex = TextBox1.Text.IndexOf("Basic", startIndex + 1)
End While
```

To test this code segment, place a multiline TextBox and a Button control on a form; then enter the statements of the listing in the button's `Click` event handler. Run the application and enter some text on the TextBox control. Make sure that the text contains the word *Basic* or change the code to locate another word, and click the button. Notice that the `IndexOf` method performs a case-sensitive search.

Use the `Replace` method to replace a string with another within the line, the `Split` method to split the line into smaller components (such as words), and any other method exposed by the String class to manipulate the control's text. The following statement appends a string to the existing text on the control:

```
TextBox1.Text = TextBox1.Text & newString
```

This statement has appeared in just about any VB 6 application that manipulated text with the TextBox control. It is an inefficient method to append text to the control, especially if the control contains a lot of text already.

Now, you can use the `AppendText` method to append strings to the control, which is far more efficient than manipulating the `Text` property directly. To append a string to a TextBox control, use the following statement:

```
TextBox1.AppendText(newString)
```

The `AppendText` method appends the specified text to the control as is, without any line breaks between successive calls. If you want to append individual paragraphs to the control's text, you must insert the line breaks explicitly, with a statement such as the following (`vbCrLf` is a constant for the carriage return/new line characters):

```
TextBox1.AppendText(newString & vbCrLf)
```

Lines

In addition to the `Text` property, you can access the text on the control by using the `Lines` property. The `Lines` property is a string array, and each element holds a paragraph of text. The first paragraph is stored in the element `Lines(0)`, the second paragraph in the element `Lines(1)`, and so on. You can iterate through the text lines with a loop such as the following:

```
Dim iLine As Integer
For iLine = 0 To TextBox1.Lines.GetUpperBound(0) - 1
   { process string TextBox1.Lines(iLine) }
Next
```

You must replace the line in brackets with the appropriate code, of course. Because the Lines property is an array, it supports the GetUpperBound method, which returns the index of the last element in the array. Each element of the Lines array is a string, and you can call any of the String class's methods to manipulate it. Just keep in mind that you can't alter the text on the control by editing the Lines array. However, you can set the control's text by assigning an array of strings to the Lines property.

Text-Selection Properties

The TextBox control provides three properties for manipulating the text selected by the user: SelectedText, SelectionStart, and SelectionLength. Users can select a range of text with a click-and-drag operation, and the selected text will appear in reverse color. You can access the selected text from within your code through the SelectedText property, and its location in the control's text through the SelectionStart and SelectionLength properties.

SelectedText

This property returns the selected text, enabling you to manipulate the current selection from within your code. For example, you can replace the selection by assigning a new value to the SelectedText property. To convert the selected text to uppercase, use the ToUpper method of the String class:

```
TextBox1.SelectedText = TextBox1.SelectedText.ToUpper
```

SelectionStart, SelectionLength

Use these two properties to read the text selected by the user on the control, or to select text from within your code. The SelectionStart property returns or sets the position of the first character of the selected text, somewhat like placing the cursor at a specific location in the text and selecting text by dragging the mouse. The SelectionLength property returns or sets the length of the selected text.

Suppose that the user is seeking the word **Visual** in the control's text. The IndexOf method locates the string but doesn't select it. The following statements select the word in the text, highlight it, and bring it into view, so that users can spot it instantly:

```
Dim seekString As String = "Visual"
Dim strLocation As Long
strLocation = TextBox1.Text.IndexOf(seekString)
If strLocation > 0 Then
   TextBox1.SelectionStart = strLocation
   TextBox1.SelectionLength = seekString.Length
End If
TextBox1.ScrollToCaret()
```

These lines locate the string **Visual** (or any user-supplied string stored in the seekString variable) in the text and select it by setting the SelectionStart and SelectionLength properties of the TextBox control. If the located string lies outside the visible area of the control, the user must scroll the text to bring the selection into view. The TextBox control provides the ScrollToCaret method, which brings the section of the text with the cursor (the *caret position*) into view.

The few lines of code shown previously form the core of a text editor's Find command. Replacing the current selection with another string is as simple as assigning a new value to the `SelectedText` property, and this technique provides you with an easy implementation of a Find and Replace operation.

Locating the Cursor Position in the Control

The `SelectionStart` and `SelectionLength` properties always have a value even if no text is selected on the control. In this case, `SelectionLength` is `0`, and `SelectionStart` is the current position of the pointer in the text. If you want to insert some text at the pointer's location, simply assign it to the `SelectedText` property, even if no text is selected on the control.

HideSelection

The selected text in the TextBox does not remain highlighted when the user moves to another control or form; to change this default behavior, set the `HideSelection` property to False. Use this property to keep the selected text highlighted, even if another form or a dialog box, such as a Find & Replace dialog box, has the focus. Its default value is True, which means that the text doesn't remain highlighted when the TextBox loses the focus.

Text-Selection Methods

In addition to properties, the TextBox control exposes two methods for selecting text. You can select some text by using the `Select` method, whose syntax is shown next:

```
TextBox1.Select(start, length)
```

The `Select` method is equivalent to setting the `SelectionStart` and `SelectionLength` properties. To select the characters 100 through 105 on the control, call the `Select` method, passing the values 99 and 6 as arguments:

```
TextBox1.Select(99, 6)
```

As a reminder, the order of the characters starts at 0 (the first character's index is 0, the second character's index is 1, and the last character's index is the length of the string minus 1).

If the range of characters you select contains hard line breaks, you must take them into consideration as well. Each hard line break counts for two characters (carriage return and line feed). If the TextBox control contains the string **ABCDEFGHI**, the following statement will select the range **DEFG**:

```
TextBox1.Select(3, 4)
```

If you insert a line break every third character and the text becomes the following, the same statement will select the characters **DE** only:

```
ABC
DEF
GHI
```

In reality, it has also selected the two characters that separate the first two lines, but special characters aren't displayed and can't be highlighted. The length of the selection, however, is 4.

A variation of the `Select` method is the `SelectAll` method, which selects all the text on the control.

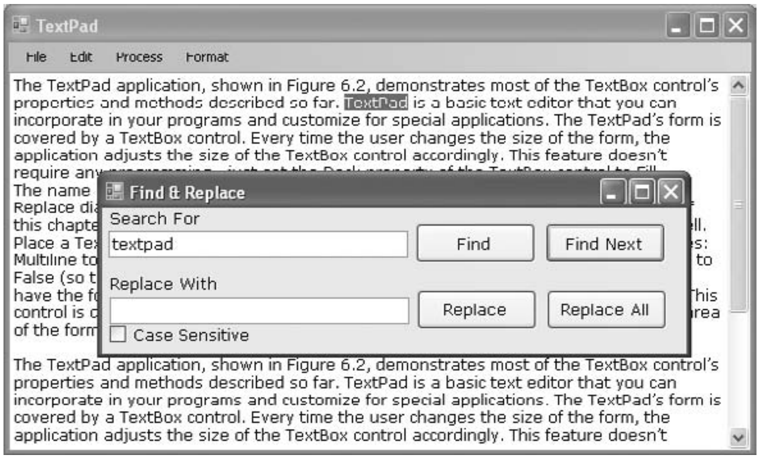
Undoing Edits

An interesting feature of the TextBox control is that it can automatically undo the most recent edit operation. To undo an operation from within your code, you must first examine the value of the `CanUndo` property. If it's True, the control can undo the operation; then you can call the `Undo` method to undo the most recent edit.

An edit operation is the insertion or deletion of characters. Entering text without deleting any is considered a single operation and will be undone in a single step. Even if the user has spent an hour entering text (without making any corrections), you can make all the text disappear with a single call to the `Undo` method. Fortunately, the deletion of the text becomes the most recent operation, which can be undone with another call to the `Undo` method. In effect, the `Undo` method is a toggle. When you call it for the first time, it undoes the last edit operation. If you call it again, it redoes the operation it previously undid. The deletion of text can be undone only if no other editing operation has taken place in the meantime. You can disable the redo operation by calling the `ClearUndo` method, which clears the undo buffer of the control. You should call it from within an `Undo` command's event handler to prevent an operation from being redone. In most cases, you should give users the option to redo an operation, especially because the `Undo` method can delete an enormous amount of text from the control.

VB 2008 at Work: The TextPad Project

The TextPad application, shown in Figure 6.2, demonstrates most of the TextBox control's properties and methods described so far. TextPad is a basic text editor that you can incorporate into your programs and customize for special applications. The TextPad project's main form is covered by a TextBox control, whose size is adjusted every time the user resizes the form. This feature doesn't require any programming — just set the `Dock` property of the TextBox control to Fill.



```
The TextPad application, shown in Figure 6.2, demonstrates most of the TextBox control's
properties and methods described so far. TextPad is a basic text editor that you can
incorporate in your programs and customize for special applications. The TextPad's form is
covered by a TextBox control. Every time the user changes the size of the form, the
application adjusts the size of the TextBox control accordingly. This feature doesn't
require any
The name
Replace di
this chapte                                                                          ll.
Place a Te                                                                           s:
Multiline to                                                                         to
False (so t
have the fo                                                                          This
control is c                                                                         rea
of the form

The TextPad application, shown in Figure 6.2, demonstrates most of the TextBox control's
properties and methods described so far. TextPad is a basic text editor that you can
incorporate in your programs and customize for special applications. The TextPad's form is
covered by a TextBox control. Every time the user changes the size of the form, the
application adjusts the size of the TextBox control accordingly. This feature doesn't
```

FIGURE 6.2 TextPad demonstrates the most useful properties and methods of the TextBox control.

The name of the application's main form is `frmTextPad`, and the name of the Find & Replace dialog box is `frmFind`. You can design the two forms as shown in the figures of this chapter, or open the TextPad project. To design the application's interface from scratch, place a MenuStrip control on the form and dock it to the top of the form. Then place a TextBox control on the main form, name it **`txtEditor`**, and set the following properties: `Multiline` to True, `MaxLength` to 0 (to edit text documents of any length), `HideSelection` to False (so that the selected text remains highlighted even when the main form doesn't have the focus), and `Dock` to Fill, so that it will fill the form.

The menu bar of the form contains all the commands you'd expect to find in text-editing applications; they're listed in Table 6.1.

Table 6.1: The TextPad Form's Menu

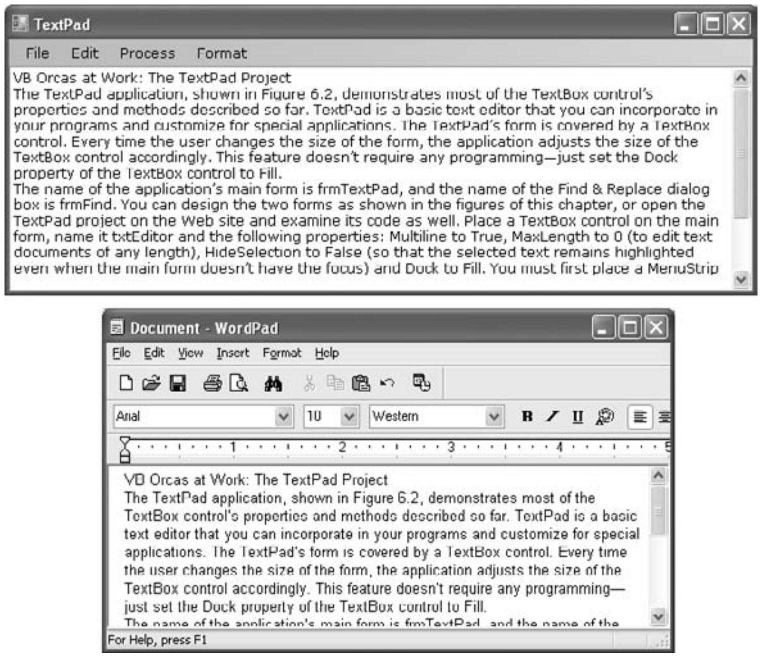
Menu	Command	Description
File	New	Clears the text
	Open	Loads a new text file from disk
	Save	Saves the text to its file on disk
	Save As	Saves the text with a new filename on disk
	Print	Prints the text
	Exit	Terminates the application
Edit	Undo/Redo	Undoes/redoes the last edit operation
	Copy	Copies selected text to the Clipboard
	Cut	Cuts the selected text
	Paste	Pastes the Clipboard's contents to the editor
	Select All	Selects all text in the control
	Find & Replace	Displays a dialog box with Find and Replace options
Process	Convert To Upper	Converts selected text to uppercase
	Convert To Lower	Converts selected text to lowercase
	Number Lines	Numbers the text lines
Format	Font	Sets the text's font, size, and attributes
	Page Color	Sets the control's background color
	Text Color	Sets the color of the text
	WordWrap	Toggle menu item that turns text wrapping on and off

The File menu commands are implemented with the Open and Save As dialog boxes, the Font command with the Font dialog box, and the Color command with the Color dialog box. These dialog boxes are discussed in the following chapters, and as you'll see, you don't have to design them yourself. All you have to do is place a control on the form and set a few properties; the Framework takes it from there. The application will display the standard Open File/Save File/Font/Color dialog boxes, in which the user can select or specify a filename or select a font or color. Of course, we'll provide a few lines of code to actually move the text into a file (or read it from a file and display it on the control), change the control's background color, and so on. I'll discuss the commands of the File menu in Chapter 8, ''More Windows Controls.''

The Editing Commands

The options on the Edit menu move the selected text to and from the Clipboard. For the TextPad application, all you need to know about the Clipboard are the `SetText` method, which places the currently selected text on the Clipboard, and the `GetText` method, which retrieves information from the Clipboard (see Figure 6.3).

Figure 6.3
The Copy, Cut, and Paste operations can be used to exchange text with any other application.



VB Orcas at Work: The TextPad Project
The TextPad application, shown in Figure 6.2, demonstrates most of the TextBox control's properties and methods described so far. TextPad is a basic text editor that you can incorporate in your programs and customize for special applications. The TextPad's form is covered by a TextBox control. Every time the user changes the size of the form, the application adjusts the size of the TextBox control accordingly. This feature doesn't require any programming—just set the Dock property of the TextBox control to Fill.
The name of the application's main form is frmTextPad, and the name of the Find & Replace dialog box is frmFind. You can design the two forms as shown in the figures of this chapter, or open the TextPad project on the Web site and examine its code as well. Place a TextBox control on the main form, name it txtEditor and the following properties: Multiline to True, MaxLength to 0 (to edit text documents of any length), HideSelection to False (so that the selected text remains highlighted even when the main form doesn't have the focus) and Dock to Fill. You must first place a MenuStrip

VB Orcas at Work: The TextPad Project
The TextPad application, shown in Figure 6.2, demonstrates most of the TextBox control's properties and methods described so far. TextPad is a basic text editor that you can incorporate in your programs and customize for special applications. The TextPad's form is covered by a TextBox control. Every time the user changes the size of the form, the application adjusts the size of the TextBox control accordingly. This feature doesn't require any programming—just set the Dock property of the TextBox control to Fill.
The name of the application's main form is frmTextPad, and the name of the

The Copy command, for example, is implemented with a single line of code (`txtEditor` is the name of the TextBox control). The Cut command does the same, and it also clears the selected text. The code for these and for the Paste command, which assigns the contents of the Clipboard to the current selection, is presented in Listing 6.2.

Listing 6.2: The Cut, Copy, and Paste Commands

```
Private Sub EditCopyItem_Click(...) _
                Handles EditCopyItem.Click
    If txtEditor.SelectionLength > 0 Then
        Clipboard.SetText(txtEditor.SelectedText)
    End If
End Sub
```

```
Private Sub EditCutItem_Click(...) _
                Handles EditCutItem.Click
    Clipboard.SetText(txtEditor.SelectedText)
    txtEditor.SelectedText = ""
End Sub

Private Sub EditPasteItem_Click(...) _
                 Handles EditPasteItem.Click
    If Clipboard.ContainsText Then
        txtEditor.SelectedText = Clipboard.GetText
    End If
End Sub
```

If no text is currently selected, the Clipboard's text is pasted at the pointer's current location. If the Clipboard contains a bitmap (placed there by another application) or any other type of data that the TextBox control can't handle, the paste operation will fail; that's why we handle the Paste operation with an `If` statement. You could provide some hint to the user by including an `Else` clause that informs them that the data on the Clipboard can't be used with a text-editing application.

The Process and Format Menus

The commands of the Process and Format menus are straightforward. The Format menu commands open the Font or Color dialog box and change the control's `Font`, `ForeColor`, and `BackColor` properties. You will learn how to use these controls in the following chapter. The Upper Case and Lower Case commands of the Process menu are also trivial: they select all the text, convert it to uppercase or lowercase, respectively, and assign the converted text to the control's `SelectedText` property with the following statements:

```
txtEditor.SelectedText = txtEditor.SelectedText.ToLower
txtEditor.SelectedText = txtEditor.SelectedText.ToUpper
```

Notice that the code uses the `SelectedText` property to convert only the selected text, not the entire document. The Number Lines command inserts a number in front of each text line and demonstrates how to process the individual lines of text on the control. However, it doesn't remove the line numbers, and there's no mechanism to prevent the user from editing the line numbers or inserting/deleting lines after they have been numbered. Use this feature to create a numbered listing or to number the lines of a file just before saving it or sharing it with another user. Listing 6.3 shows the Number Lines command's code and demonstrates how to iterate through the TextBox control's `Lines` array.

Listing 6.3: The Number Lines Command

```
Private Sub ProcessNumberLinesItem_Click(...) _
                Handles ProcessNumberLines.Click
    Dim iLine As Integer
    Dim newText As New System.Text.StringBuilder()
    For iLine = 0 To txtEditor.Lines.Length - 1
```

```
            newText.Append((iLine + 1).ToString & vbTab & _
                           txtEditor.Lines(iLine) & vbCrLf)
        Next
        txtEditor.SelectAll()
        Clipboard.SetText(newText.ToString)
        txtEditor.Paste()
    End Sub
```

This event handler uses a `StringBuilder` variable. The StringBuilder class, which is discussed in detail in Chapter 12, ''Designing Custom Windows Controls,'' is equivalent to the String class; it exposes similar methods and properties, but it's much faster at manipulating dynamic strings than the String class.

Search and Replace Operations

The last option in the Edit menu — and the most interesting — displays a Find & Replace dialog box (shown in Figure 6.2). This dialog box works like the similarly named dialog box of Microsoft Word and many other Windows applications. The buttons in the Find & Replace dialog box are relatively self-explanatory:

Find The Find command locates the first instance of the specified string in the text after the cursor location. If a match is found, the Find Next, Replace, and Replace All buttons are enabled.

Find Next This command locates the next instance of the string in the text. Initially, this button is disabled; it's enabled only after a successful Find operation.

Replace This replaces the current selection with the replacement string and then locates the next instance of the same string in the text. Like the Find Next button, it's disabled until a successful Find operation occurs.

Replace All This replaces all instances of the string specified in the Search For box with the string in the Replace With box.

Design a form like the one shown in Figure 6.2 and set its `TopMost` property to True. We want this form to remain on top of the main form, even when it doesn't have the focus.

Whether the search is case-sensitive or not depends on the status of the Case Sensitive CheckBox control. If the string is found in the control's text, the program highlights it by selecting it. In addition, the program calls the TextBox control's `ScrollToCaret` method to bring the selection into view. The Find Next button takes into consideration the location of the pointer and searches for a match *after* the current location. If the user moves the pointer somewhere else and then clicks the Find Next button, the program will locate the first instance of the string after the current location of the pointer — and not after the last match. Of course, you can always keep track of the location of each match and continue the search from this location. The Find button executes the code shown in Listing 6.4.

Listing 6.4: The Find Button

```
    Private Sub bttnFind_Click(...) Handles bttnFind.Click
        Dim selStart As Integer
        If chkCase.Checked = True Then
            selStart = _
```

```
                  frmTextPad.txtEditor.Text.IndexOf( _
                  searchWord.Text, StringComparison.Ordinal)
        Else
            selStart = _
                frmTextPad.txtEditor.Text.IndexOf( _
                searchWord.Text, _
                StringComparison.OrdinalIgnoreCase)
        End If
        If selStart = -1 Then
            MsgBox("Can't find word")
            Exit Sub
        End If
        frmTextPad.txtEditor.Select( _
                      selStart, searchWord.Text.Length)
        bttnFindNext.Enabled = True
        bttnReplace.Enabled = True
        bttnReplaceAll.Enabled = True
        frmTextPad.txtEditor.ScrollToCaret()
    End Sub
```

The Find button examines the value of the `chkCase` CheckBox control, which specifies whether the search will be case-sensitive and calls the appropriate form of the `IndexOf` method. The first argument of this method is the string we're searching for; the second argument is the search mode, and its value is a member of the `StringComparison` enumeration: `Ordinal` for case-sensitive searches and `OrdinalIgnoreCase` for case-insensitive searches. If the `IndexOf` method locates the string, the program selects it by calling the control's `Select` method with the appropriate arguments. If not, it displays a message. Notice that after a successful Find operation, the Find Next, Replace, and Replace All buttons on the form are enabled.

The code of the Find Next button is the same, but it starts searching at the character following the current selection. This way, the `IndexOf` method locates the next instance of the same string. Here's the statement that locates the next instance of the search argument:

```
selStart = frmTextPad.txtEditor.Text.IndexOf( _
            searchWord.Text,  _
            frmTextPad.txtEditor.SelectionStart + 1, _
            StringComparison.Ordinal)
```

The Replace button replaces the current selection with the replacement string and then locates the next instance of the find string. The Replace All button replaces all instances of the search word in the document. Listing 6.5 presents the code behind the Replace and Replace All buttons.

Listing 6.5: The Replace and Replace All Operations

```
    Private Sub bttnReplace_Click(...) _
                   Handles bttnReplace.Click
        If frmTextPad.txtEditor.SelectedText <> "" Then
            frmTextPad.txtEditor.SelectedText = replaceWord.Text
        End If
         bttnFindNext_Click(sender, e)
    End Sub
```

```
Private Sub bttnReplaceAll_Click(...) _
              Handles bttnReplaceAll.Click
    Dim curPos, curSel As Integer
    curPos = frmTextPad.txtEditor.SelectionStart
    curSel = frmTextPad.txtEditor.SelectionLength
    frmTextPad.txtEditor.Text = _
         frmTextPad.txtEditor.Text.Replace( _
         searchWord.Text.Trim, replaceWord.Text.Trim)
    frmTextPad.txtEditor.SelectionStart = curPos
    frmTextPad.txtEditor.SelectionLength = curSel
End Sub
```

The `Replace` method is case-sensitive, which means that it replaces instances of the search argument in the text that have the exact same spelling as its first argument. For a case-insensitive replace operation, you must write the code to perform consecutive case-insensitive search-and-replace operations. Alternatively, you can use the `Replace` built-in function to perform case-insensitive searches. Here's how you'd call the `Replace` function to perform a case-insensitive replace operation:

```
Replace(frmTextPad.txtEditor.Text,  searchWord.Text.Trim, _
      replaceWord.Text.Trim, , , CompareMethod.Text)
```

The last, optional, argument determines whether the search will be case-sensitive (`CompareMethod.Binary`) or case-insensitive (`CompareMethod.Text`).

The Undo/Redo Commands

The Undo command (shown in Listing 6.6) is implemented with a call to the `Undo` method. However, because the `Undo` method works like a toggle, we must also toggle its caption from Undo to Redo (and vice versa) each time the command is activated.

Listing 6.6: The Undo/Redo Command of the Edit Menu

```
Private Sub EditUndoItem_Click(...) _
            Handles EditUndoItem.Click
    If EditUndoItem.Text = "Undo" Then
        If txtEditor.CanUndo Then
            txtEditor.Undo()
            EditUndoItem.Text = "Redo"
        End If
    Else
        If txtEditor.CanUndo Then
            txtEditor.Undo()
            EditUndoItem.Text = "Undo"
        End If
    End If
End Sub
```

If you edit the text after an undo operation, you can no longer redo the last undo operation. This means that as soon as the contents of the TextBox control change, the caption of the first command in the Edit menu must become Undo, even if it's Redo at the time. The Redo command is available only after undoing an operation and before editing the text. So, how do we know that the text has been edited? The TextBox control fires the `TextChanged` event every time its contents change. We'll use this event to restore the caption of the Undo/Redo command to Undo. Insert the following statement in the `TextChanged` event of the TextBox control:

```
EditUndoItem.Text = "Undo"
```

The TextBox control can't provide more-granular undo operations — unlike Word, which keeps track of user actions (insertions, deletions, replacements, and so on) and then undoes them in steps. If you need a more-granular undo feature, you should use the RichTextBox control, which is discussed in detail in Chapter 8. The RichTextBox control can display formatted text, but it can also be used as an enhanced TextBox control. By the way, setting the menu item's caption from within the TextChanged event handler is an overkill, because this event takes place every time the user presses a key. However, the operation takes no time at all and doesn't make the application less responsive. A better choice would be the `DropDownOpening` event of the `editFormat` item, which is fired every time the user opens the Edit menu.

Capturing Keystrokes

The TextBox control has a single unique event, the `TextChanged` event, which is fired every time the text on the control is changed, either because the user has typed a character or because of a paste operation. Another event that is quite common in programming the TextBox control is the `KeyPress` event, which occurs every time a key is pressed and reports the character that was pressed. You can use this event to capture certain keys and modify the program's behavior depending on the character typed.

Suppose that you want to use the TextPad application to prepare messages for transmission over a telex line. As you may know, a telex can't transmit lowercase characters or special symbols. The editor must convert the text to uppercase and replace the special symbols with their equivalent strings: DLR for $, AT for @, O/O for %, BPT for #, and AND for &. You can modify the default behavior of the TextBox control from within the `KeyPress` event so that it converts these characters as the user types.

By capturing keystrokes, you can process the data as they are entered, in real time. For example, you can make sure that a TextBox accepts only numeric or hexadecimal characters and rejects all others. To implement an editor for preparing text for telex transmission, use the `KeyPress` event handler shown in Listing 6.7.

Listing 6.7: Handling Keystrokes for a TELEX message

```
Private Sub txtEditor_KeyPress( _
             ByVal sender As Object, _
             ByVal e As System.Windows. _
                   Forms.KeyPressEventArgs) _
             Handles txtEditor.KeyPress
    If System.Char.IsControl(e.KeyChar) Then Exit Sub
    Dim ch As Char = Char.ToUpper(e.KeyChar)
```

```
        Select Case ch.ToString
            Case "@"
                txtEditor.SelectedText = "AT"
            Case "#"
                txtEditor.SelectedText = "BPT"
            Case "$"
                txtEditor.SelectedText = "DLR"
            Case "%"
                txtEditor.SelectedText = "O/O"
            Case "&"
                txtEditor.SelectedText = "AND"
            Case Else
                txtEditor.SelectedText = ch
        End Select
        e.Handled = True
    End Sub
```

The very first executable statement in the event handler examines the key that was pressed and exits if it is a special editing key (Delete, Backspace, Ctrl+V, and so on). If so, the handler exits without taking any action. The `KeyChar` property of the `e` argument of the `KeyPress` event reports the key that was pressed. The code converts it to a string and then uses a `Case` statement to handle individual keystrokes. If the user pressed the $ key, for example, the code displays the characters `DLR`. If no special character was pressed, the code displays the character pressed as is from within the `Case Else` clause of the `Select` statement.

Cancelling Keystrokes

Before you exit the event handler, you must ''kill'' the original key pressed, so that it won't appear on the control. You do this by setting the `Handled` property to True, which tells VB that it shouldn't process the keystroke any further. If you omit this statement, the special characters will be printed twice: once in their transformed format (DLR$, AT@, and so on) and once as regular characters. You can also set the `SuppressKeyPress` property to True to cancel a keystroke; the Common Language Runtime (CLR) will not pass the keystroke to the appropriate control.

Capturing Function Keys

Another common feature in text-editing applications is the assignment of special operations to the function keys. The Notepad application, for example, uses the F5 function key to insert the current date at the cursor's location. You can do the same with the TextPad application, but you can't use the `KeyPress` event — the `KeyChar` argument doesn't report function keys. The events that can capture the function keys are the `KeyDown` and `KeyUp` events. Also, unlike the `KeyPress` event, these two events don't report the character pressed, but instead report the key's code (a special number that distinguishes each key on the keyboard, also known as the *scancode*), through the `e.KeyCode` property.

The keycode is unique for each key, not each character. Lower- and uppercase characters have different ASCII values but the same keycode because they are on the same key. For example, the number 4 and the $ symbol have the same keycode because the same key on the keyboard generates both characters. When the key's code is reported, the KeyDown and KeyUp events also report the state of the Shift, Ctrl, and Alt keys through the e.Shift, e.Alt, and e.Control properties.

The KeyUp event handler shown in Listing 6.8 uses the F5 and F6 function keys to insert the current date and time in the document. It also uses the F7 and F8 keys to insert two predefined strings in the document.

LISTING 6.8: *KeyUp* Event Examples

```
Private Sub txtEditor_KeyUp(ByVal sender As Object, _
      ByVal e As System.Windows.Forms.KeyEventArgs) _
      Handles txtEditor.KeyUp
    Select Case e.KeyCode
        Case Keys.F5 :
                 txtEditor.SelectedText = _
                 Now().ToLongDateString
        Case Keys.F6 :
                 txtEditor.SelectedText = _
                 Now().ToLongTimeString
        Case Keys.F7 :
                 txtEditor.SelectedText = _
                 "MicroWeb Designs, Inc."
        Case Keys.F8 :
                 txtEditor.SelectedText = _
                 "Another user-supplied string"
    End Select
End Sub
```

Windows already uses many of the function keys (for example, the F1 key for help), and you shouldn't modify their original functions. With a little additional effort, you can provide users with a dialog box that lets them assign their own strings to function keys. You'll probably have to take into consideration the status of the Shift, Control, and Alt properties of the event's e argument, which report the status of the Shift, Ctrl, and Alt keys, respectively. To find out whether *two* of the modifier keys are pressed along with a key, use the AND operator with the appropriate properties of the e argument. The following If clause detects the Ctrl and Alt keys:

```
If e.Control AND e.Alt Then
   { Both Alt and Control keys were down}
End If
```

Auto-complete Properties

One set of interesting properties of the TextBox control are the autocomplete properties. Have you noticed how Internet Explorer prompts you with possible matches as soon as you start typing an address or your username in a text box (or in the address bar of the browser)? You can easily implement such boxes with a single-line TextBox control and the autocomplete properties. Basically, you have to tell the TextBox control how it should prompt the user with strings that match the characters already entered on the control and where the matches will come from. Then, the control can display in a drop-down list the strings that begin with the characters already typed by the user. The user can either continue typing (in which case the list of options becomes shorter) or select an item from the list. The autocomplete properties apply to single-line TextBox controls only; they do not take effect on multiline TextBox controls.

In many cases, an autocomplete TextBox control is more functional than a ComboBox control, and you should prefer it. You will see that the ComboBox also supports the autocomplete properties, because they make the control so much easier to use only with the keyboard.

The `AutoCompleteMode` property determines whether, and how, the TextBox control will prompt users, and its setting is a member of the `AutoCompleteMode` enumeration (`AutoSuggest`, `AutoAppend`, `AutoSuggestAppend`, and `None`). In `AutoAppend` mode, the TextBox control selects the first matching item in the list of suggestions and completes the text. In `AutoSuggestAppend` mode, the control suggests the first matching item in the list, as before, but it also expands the list. In `AutoSuggest` mode, the control simply opens a list with the matching items but doesn't select any of them. Regular TextBox controls have their `AutoCompleteMode` property set to False.

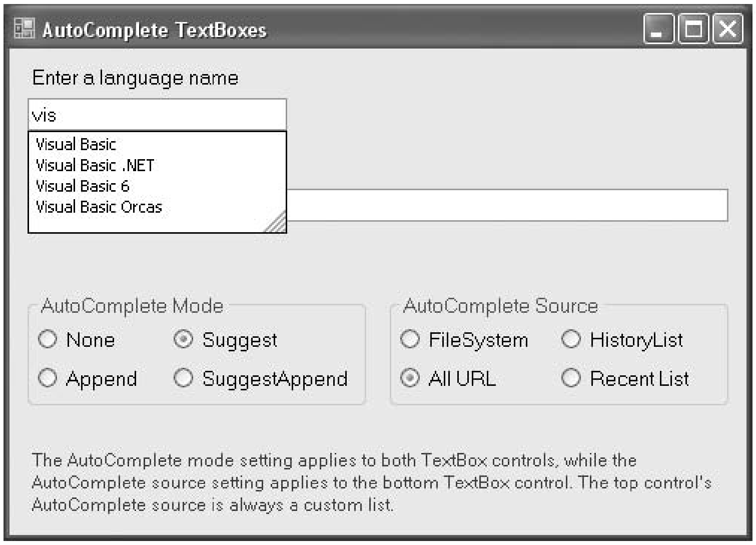
The `AutoCompleteSource` property determines where the list of suggestions comes from; its value is a member of the `AutoCompleteSource` enumeration, which is shown in Table 6.2.

TABLE 6.2: The Members of the *AutoCompleteSource*

MEMBER	DESCRIPTION
`AllSystemSources`	The suggested items are the names of system resources.
`AllUrl`	The suggested items are the URLs visited by the target computer. Does not work if you're deleting the recently viewed pages.
`CustomSource`	The suggested items come from a custom collection.
`FileSystem`	The suggested items are filenames.
`HistoryList`	The suggested items come from the computer's history list.
`RecentlyUsedList`	The suggested items come from the Recently Used folder.
`None`	The control doesn't suggest any items.

To demonstrate the basics of the autocomplete properties, I've included the AutoCompleteTextBoxes project, whose main form is shown in Figure 6.4. This project allows you to set the autocomplete mode and source for a single-line TextBox control. The top TextBox control uses a custom list of words, while the lower one uses one of the built-in autocomplete sources (file system, URLs, and so on).

FIGURE 6.4
Suggesting words with the `AutoComplete-Source` property


AutoComplete TextBoxes

Enter a language name

vis

Visual Basic
Visual Basic .NET
Visual Basic 6
Visual Basic Orcas

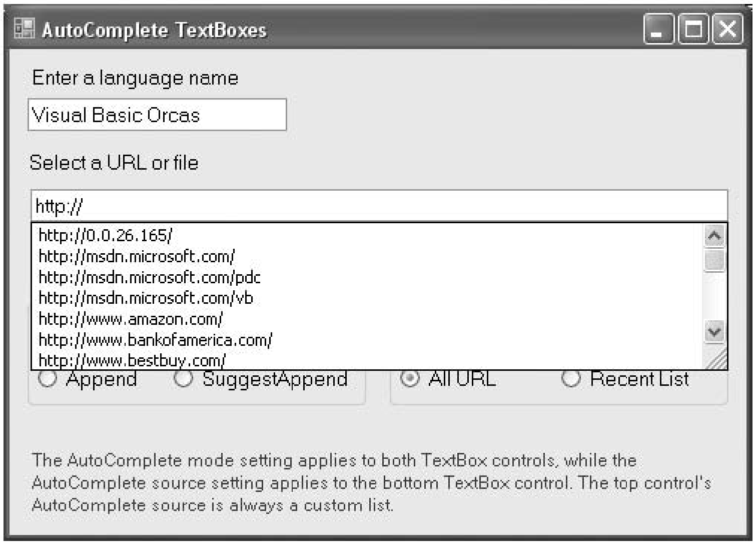
AutoComplete Mode

None [✓ Suggest]
Append SuggestAppend

AutoComplete Source

FileSystem HistoryList
[✓ All URL] Recent List

The AutoComplete mode setting applies to both TextBox controls, while the AutoComplete source setting applies to the bottom TextBox control. The top control's AutoComplete source is always a custom list.


AutoComplete TextBoxes

Enter a language name

Visual Basic Orcas

Select a URL or file

http://

http://0.0.26.165/
http://msdn.microsoft.com/
http://msdn.microsoft.com/pdc
http://msdn.microsoft.com/vb
http://www.amazon.com/
http://www.bankofamerica.com/
http://www.bestbuy.com/

Append SuggestAppend [✓ All URL] Recent List

The AutoComplete mode setting applies to both TextBox controls, while the AutoComplete source setting applies to the bottom TextBox control. The top control's AutoComplete source is always a custom list.

If you set the `AutoCompleteSource` to CustomSource, you must also populate an `AutoCompleteStringCollection` object with the desired suggestions and assign it to the `AutoCompleteCustomSource` property. The `AutoCompleteStringCollection` is just a collection of strings. Listing 6.9 shows statements in a form's `Load` event that prepare such a list and use it with the *`TextBox1`* control.

Listing 6.9: Populating a Custom *AutoCompleteSource* Property

```
Private Sub Form1_Load(...) _
            Handles MyBase.Load
    Dim knownWords As New AutoCompleteStringCollection
    knownWords.Add("Visual Basic 2008")
    knownWords.Add("Visual Basic .NET")
    knownWords.Add("Visual Basic 6")
    knownWords.Add("Visual Basic")
    knownWords.Add("Framework")
    TextBox1.AutoCompleteCustomSource = knownWords
    TextBox1.AutoCompleteSource = AutoCompleteSource.CustomSource
    TextBox1.AutoCompleteMode =
AutoCompleteMode.Suggest
    TextBox2.AutoCompleteSource =
AutoCompleteSource.RecentlyUsedList
    TextBox2.AutoCompleteMode =
AutoCompleteMode.Suggest
End Sub
```

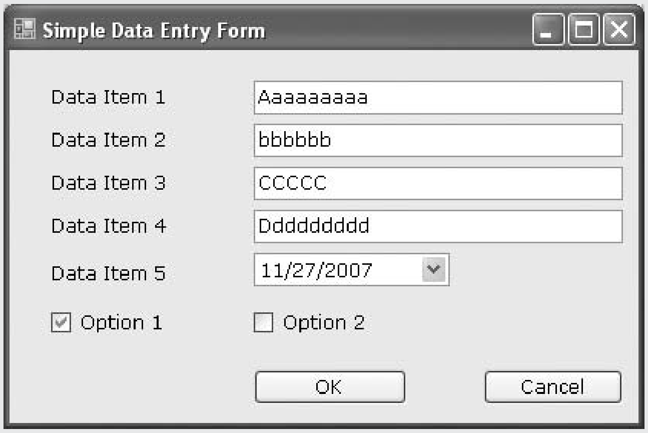
The *TextBox1* control on the form will open a drop-down list with all possible matches in the `knownWords` collection as soon as the user starts typing in the control, as shown in the top part of Figure 6.4. To see the autocomplete properties in action, open the AutoCompleteTextBoxes project and examine its code. The main form of the application, shown in Figure 6.4, allows you to change the `AutoCompleteMode` property of both TextBox controls on the form, and the `AutoComplete-Source` property of the bottom TextBox control. The first TextBox uses a list of custom words, which is set up when the form is loaded, with the statements in Listing 6.9.

Real World Scenario

Real-World Data-Entry Applications

Typical business applications contain numerous forms for data entry, and the most common element on data-entry forms is the TextBox control. Data-entry operators are very efficient with the keyboard and they should be able to use your application without reaching for the mouse.

Seasoned data-entry operators can’t live without the Enter key; they reach for this key at the end of each operation. In my experience, a functional interface should add intelligence to this keystroke: the Enter key should perform the ‘‘obvious’’ or ‘‘most likely’’ operation at any time. When entering data, for example, it should take the user to the next control in the Tab order. Consider a data-entry screen like the one shown in the following image, which contains several TextBox controls, a DataTimePicker control for entering dates, and two CheckBox controls. This is the main form of the Simple Data Entry Form sample project, which you will find along with the other chapter’s projects.



The application uses the Enter key intelligently: every time the Enter key is pressed, the focus is moved to the next control in the Tab order. Even if the current control is a CheckBox, this keystroke doesn't change the status of the CheckBox controls; it simply moves the focus forward.

You could program the KeyUp event of each control to react to the Enter key, but this approach can lead to maintenance problems if you're going to add new controls to an existing form. The best approach is to intercept the Enter keystroke at the form's level, before it reaches a control. To do so, you must set the KeyPreview property of the form to True. This setting causes the key events to be fired at the form's level first and then to the control that has the focus. In essence, it allows you to handle certain keystrokes for multiple controls at once. The KeyUp event handler of the sample project's main form intercepts the Enter keystroke and reacts to it by moving the focus to the next control in the Tab order via the ProcessTabKey method. This method simulates the pressing of the Tab key, and it's called with a single argument, which is a Boolean value: True moves the focus forward, and False moves it backward. Here's the code in the KeyUp event handler of the application's form that makes the interface much more functional and intuitive:

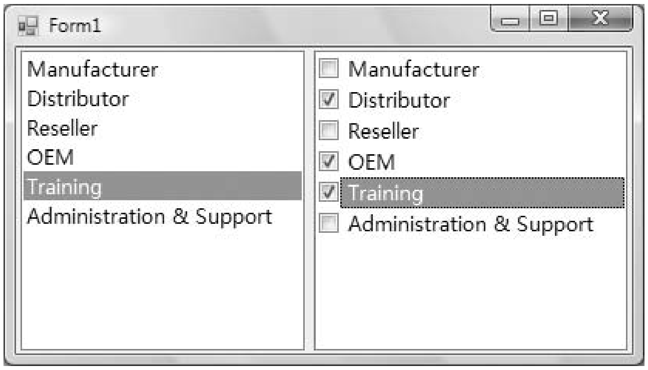
```
Private Sub frmDataEntry_KeyUp( _
               ByVal sender As Object, _
               ByVal e As System.Windows.Forms.KeyEventArgs) _
               Handles Me.KeyUp
    If e.KeyCode = Keys.Enter And Not (e.Alt Or e.Control) Then
        If Me.ActiveControl.GetType Is GetType(TextBox) Or _
              Me.ActiveControl.GetType Is GetType(CheckBox) Or _
              Me.ActiveControl.GetType Is _
              GetType(DateTimePicker) Then
            If e.Shift Then
                Me.ProcessTabKey(False)
            Else
                Me.ProcessTabKey(True)
            End If
        End If
    End If
End Sub
```

There are a couple of things you should notice about this handler. First, it doesn't react to the Enter key if it was pressed along with the Alt or Ctrl keys. The Shift key, on the other hand, is used to control the direction in the Tab order. The focus moves forward with the Enter keystroke and moves backward with the Shift + Enter keystroke. Also, the focus is handled automatically only for the TextBox, CheckBox, and DataTimePicker controls. When the user presses the Enter key when a button has the focus, the program reacts as expected by invoking the button's `Click` event handler.

The ListBox, CheckedListBox, and ComboBox Controls

The ListBox, CheckedListBox, and ComboBox controls present lists of choices, from which the user can select one or more. The first two are illustrated in Figure 6.5.

Figure 6.5
The ListBox and CheckedListBox controls



The ListBox control occupies a user-specified amount of space on the form and is populated with a list of items. If the list of items is longer than can fit on the control, a vertical scroll bar appears automatically.

The CheckedListBox control is a variation of the ListBox control. It's identical to the ListBox control, but a check box appears in front of each item. The user can select any number of items by selecting the check boxes in front of them. As you know, you can also select multiple items from a ListBox control by pressing the Shift and Ctrl keys.

The ComboBox control also contains multiple items but typically occupies less space on the screen. The ComboBox control is an expandable ListBox control: The user can expand it to make a selection, and collapse it after the selection is made. The real advantage of the ComboBox control, however, is that the user can enter new information in the ComboBox, rather than being forced to select from the items listed.

To add items at design time, locate the `Items` property in the control's Properties window and click the ellipsis button. A new window will pop up — the String Collection Editor window — in which you can add the items you want to display in the list. Each item must appear on a separate text line, and blank text lines will result in blank lines in the list. These items will appear in the list when the form is loaded, but you can add more items (or remove existing ones) from within your code at any time. They appear in the same order as entered on the String Collection Editor window unless the control has its `Sorted` property set to True, in which case the items are automatically sorted, regardless of the order in which you've specified them.

This section first examines the ListBox control's properties and methods. Later, you'll see how the same properties and methods can be used with the ComboBox control.

Basic Properties

In this section, you'll find the properties that determine the functionality of the three controls. These properties are usually set at design time, but you can change their setting from within your application's code.

IntegralHeight

This property is a Boolean value (True/False) that indicates whether the control's height will be adjusted to avoid the partial display of the last item. When set to True, the control's actual height changes in multiples of the height of a single line, so only an integer number of rows are displayed at all times.

Items

The `Items` property is a collection that holds the control's items. At design time, you can populate this list through the String Collection Editor window. At runtime, you can access and manipulate the items through the methods and properties of the `Items` collection, which are described shortly.

MultiColumn

A ListBox control can display its items in multiple columns if you set its `MultiColumn` property to True. The problem with multicolumn ListBoxes is that you can't specify the column in which each item will appear. ListBoxes with many items and their `MultiColumn` property set to True expand horizontally, not vertically. A horizontal scroll bar will be attached to a multicolumn ListBox, so that users can bring any column into view. This property does not apply to the ComboBox control.

SelectionMode

This property, which applies to the ListBox and CheckedListBox controls only, determines how the user can select the list's items. The possible values of this property — members of the `SelectionMode` enumeration — are shown in Table 6.3.

Table 6.3: The *SelectionMode* Enumeration

Value	Description
None	No selection at all is allowed.
One	(Default) Only a single item can be selected.
MultiSimple	Simple multiple selection: A mouse click (or pressing the spacebar) selects or deselects an item in the list. You must click all the items you want to select.
MultiExtended	Extended multiple selection: Press Shift and click the mouse (or press one of the arrow keys) to expand the selection. This process highlights all the items between the previously selected item and the current selection. Press Ctrl and click the mouse to select or deselect single items in the list.

Sorted

When this property is True, the items remain sorted at all times. The default is False, because it takes longer to insert new items in their proper location. This property's value can be set at design time as well as runtime.

The items in a sorted ListBox control are sorted in ascending and case-sensitive order. Uppercase characters appear before the equivalent lowercase characters, but both upper- and lowercase characters appear together. All words beginning with *B* appear after the words beginning with *A* and before the words beginning with *C*. Within the group of words beginning with *B*, those beginning with a capital *B* appear before those beginning with a lowercase *b*. This sorting order is known as *phone book order*.

Moreover, the ListBox control won't sort numeric data. The number 10 will appear in front of the number 5 because the string ***10*** is smaller than the string ***5***. If the numbers are formatted as ***010*** and ***005***, they will be sorted correctly.

Text

The `Text` property returns the selected text on the control. Although you can set the `Text` property for the ComboBox control at design time, this property is available only at runtime for the other two controls. Notice that the items need not be strings. By default, each item is an object. For each object, however, the control displays a string, which is the same string returned by the object's `ToString` method.

Manipulating the *Items* Collection

To manipulate a ListBox control from within your application, you should be able to do the following:

- Add items to the list
- Remove items from the list
- Access individual items in the list

The items in the list are represented by the `Items` collection. You use the members of the `Items` collection to access the control's items and to add or remove items. The `Items` property exposes the standard members of a collection, which are described later in this section.

Each member of the `Items` collection is an object. In most cases, we use ListBox controls to store strings, but it's possible to store objects. When you add an object to a ListBox control, a string is displayed on the corresponding line of the control. This is the string returned by the object's `ToString` method. This is the property of the object that will be displayed by default. You can display any other property of the object by setting the control's `ValueMember` property to the name of the property.

If you add a Color object and a Rectangle object to the `Items` collection with the following statements:

```
ListBox1.Items.Add(New Font("Verdana", 12, _
FontStyle.Bold)
ListBox1.Items.Add(New Rectangle(0, 0, 100, 100))
```

then the following strings appear on the first two lines of the control:

```
 [Font: Name=Verdana, Size=12, Units=3, GdiCharSet=1, gdiVerticalFont=False]
{X=0, Y=0, Width=100, Height=100}
```

However, you can access the members of the two objects because the ListBox stores objects, not their descriptions. The following statement prints the width of the Rectangle object (the output produced by the statement is highlighted):

```
Debug.WriteLine(ListBox1.Items.Item(1).Width)
100
```

The expression in the preceding statement is *late-bound*, which means that the compiler doesn't know whether the first object in the Items collection is a Rectangle object and it can't verify the member Width. If you attempt to call the Width property of the first item in the collection, you'll get an exception at runtime indicating that the code has attempted to access a missing member. The missing member is the Width property of the Font object.

The proper way to read the objects stored in a ListBox control is to examine the type of the object first and then attempt to retrieve a property (or call a method) of the object, only if it's of the appropriate type. Here's how you would read the Width property of a Rectangle object:

```
If ListBox1.Items.Item(0).GetType Is _
       GetType(Rectangle) Then
   Debug.WriteLine( _
       CType(ListBox1.Items.Item(0), Rectangle).Width)
End If
```

The *Add* Method

To add items to the list, use the Items.Add or Items.Insert method. The syntax of the Add method is as follows:

```
ListBox1.Items.Add(item)
```

The item parameter is the object to be added to the list. You can add any object to the ListBox control, but items are usually strings. The Add method appends new items to the end of the list, unless the Sorted property has been set to True.

The following loop adds the elements of the array words to a ListBox control, one at a time:

```
Dim words(100) As String
{ statements to populate array }
Dim i As Integer
For i = 0 To 99
   ListBox1.Items.Add(words(i))
Next
```

Similarly, you can iterate through all the items on the control by using a loop such as the following:

```
Dim i As Integer
For i = 0 To ListBox1.Items.Count - 1
   { statements to process item ListBox1.Items(i) }
Next
```

You can also use the For Each . . . Next statement to iterate through the Items collection, as shown here:

```
Dim itm As Object
For Each itm In ListBox1.Items
   { process the current item, represented by the itm variable }
Next
```

When you populate a ListBox control with a large number of items, call the `BeginUpdate` method before starting the loop and call the `EndUpdate` method when you're done. These two methods turn off the visual update of the control while you're populating it and they speed up the process considerably. When the `EndUpdate` method is called, the control is redrawn with all the items.

The *Insert* Method

To insert an item at a specific location, use the `Insert` method, whose syntax is as follows:

```
ListBox1.Items.Insert(index, item)
```

The `item` parameter is the object to be added, and `index` is the location of the new item. The first item's index in the list is zero. Note that you need not insert items at specific locations when the list is sorted. If you do, the items will be inserted at the specified locations, but the list will no longer be sorted.

The *Clear* Method

The `Clear` method removes all the items from the control. Its syntax is quite simple:

```
List1.Items.Clear
```

The *Count* Property

This is the number of items in the list. If you want to access all the items with a `For` . . . `Next` loop, the loop's counter must go from 0 to `ListBox.Items.Count - 1`, as shown in the example of the `Add` method.

The *CopyTo* Method

The `CopyTo` method of the `Items` collection retrieves all the items from a ListBox control and stores them in the array passed to the method as an argument. The syntax of the `CopyTo` method is

```
ListBox.CopyTo(destination, index)
```

where `destination` is the name of the array that will accept the items, and `index` is the index of an element in the array where the first item will be stored. The array that will hold the items of the control must be declared explicitly and must be large enough to hold all the items.

The *Remove* and *RemoveAt* Methods

To remove an item from the list, you can simply call the `Items` collection's `Remove` method, passing the object to be removed as an argument. If the control contains strings, pass the string to be removed. If the same string appears multiple times on the control, only the first instance will be removed.

You can also remove an item by specifying its position in the list via the `RemoveAt` method, which accepts as argument the position of the item to be removed:

```
ListBox1.Items.RemoveAt(index)
```

The `index` parameter is the order of the item to be removed, and the first item's order is 0.

The *Contains* Method

The `Contains` method of the `Items` collection — not to be confused with the control's `Contains` method — accepts an object as an argument and returns a True/False value that indicates whether the collection contains this object. Use the `Contains` method to avoid the insertion of identical objects into the ListBox control. The following statements add a string to the `Items` collection, only if the string isn't already part of the collection:

```
Dim itm As String = "Remote Computing"
If Not ListBox1.Items.Contains(itm) Then
    ListBox1.Items.Add(itm)
End If
```

Selecting Items

The ListBox control allows the user to select either one or multiple items, depending on the setting of the `SelectionMode` property. In a single-selection ListBox control, you can retrieve the selected item by using the `SelectedItem` property, and its index by using the `SelectedIndex` property. `SelectedItem` returns the selected item, which is an object. The text of the selected item is reported by the `Text` property.

If the control allows the selection of multiple items, they're reported with the `SelectedItems` property. This property is a collection of objects and exposes the same members as the `Items` collection. Because the ComboBox does not allow the selection of multiple items, it provides only the `SelectedIndex` and `SelectedItem` properties.

To iterate through all the selected items in a multiselection ListBox control, use a loop such as the following:

```
Dim itm As Object
For Each itm In ListBox1.SelectedItems
   Debug.WriteLine(itm)
Next
```

The *`itm`* variable should be declared as `Object` because the items in the ListBox control are objects. If they're all of the same type, you can convert them to the specific type and then call their methods. If all the items are of the Rectangle type, you can use a loop like the following to print the area of each rectangle:

```
Dim itm As Rectangle
For Each itm In ListBox1.SelectedItems
   Debug.WriteLine(itm.Width * itm.Height)
Next
```

VB 2008 at Work: The ListBox Demo Project

The ListBox Demo application (shown in Figure 6.6) demonstrates the basic operations of the ListBox control. The two ListBox controls on the form operate slightly differently. The first has the default configuration: Only one item can be selected at a time, and new items are appended after the existing item. The second ListBox control has its `Sorted` property set to True and its `MultiSelect` property set according to the values of the two RadioButton controls at the bottom of the form.

The code for the ListBox Demo application contains much of the logic you'll need in your ListBox manipulation routines. It shows you how to do the following:

- Add and remove items at runtime
- Transfer items between lists at runtime
- Handle multiple selected items
- Maintain sorted lists

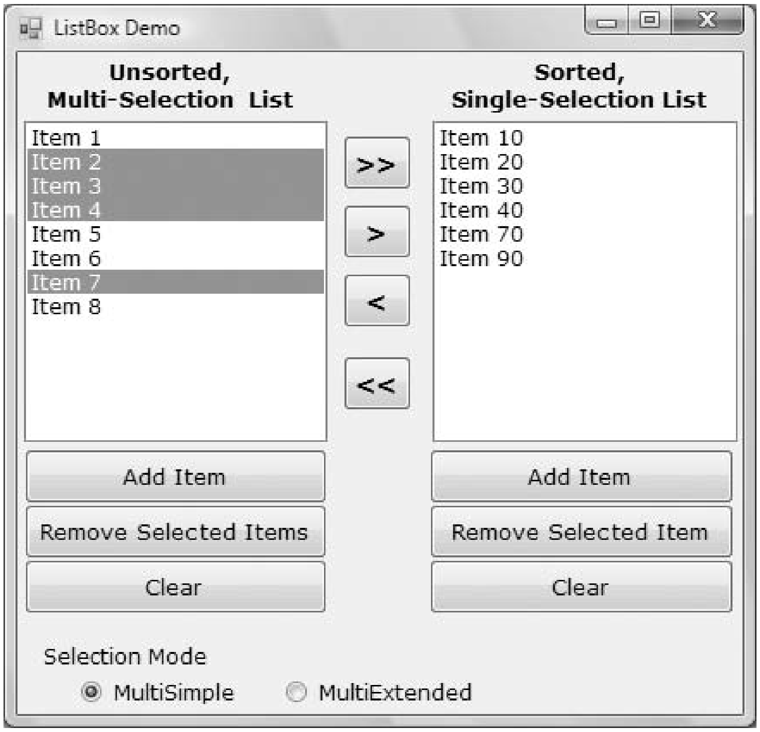


FIGURE 6.6 ListBox Demo demonstrates most of the operations you'll perform with ListBoxes.

THE ADD ITEM BUTTONS

The Add Item buttons use the `InputBox()` function to prompt the user for input, and then they add the user-supplied string to the ListBox control. The code is identical for both buttons (see Listing 6.10).

LISTING 6.10: The Add New Element Buttons

```
Private Sub bttnSourceAdd_Click(...) _
          Handles bttnSourceAdd.Click
    Dim ListItem As String
    ListItem = InputBox("Enter new item's name")
```

```
        If ListItem.Trim <> "" Then
            sourceList.Items.Add(ListItem)
        End If
    End Sub
```

Notice that the subroutine examines the data entered by the user to avoid adding blank strings to the list. The code for the Clear buttons is also straightforward; it simply calls the `Clear` method of the `Items` collection to remove all entries from the corresponding list.

Removing Items from the Two Lists

The code for the Remove Selected Item button is different from that for the Remove Selected Items button (both are presented in Listing 6.11). The code for the Remove Selected Item button removes the selected item, while the Remove Selected Items buttons must scan all the items of the left list and remove the selected one(s).

Listing 6.11: The Remove Buttons

```
    Private Sub bttnDestinationRemove_Click(...) _
                  Handles bttnDestinationRemove.Click
        destinationList.Items.Remove( destinationList.SelectedItem)
    End Sub

    Private Sub bttnSourceRemove_Click(...) _
                  Handles bttnSourceRemove.Click
        Dim i As Integer
        For i = 0 To sourceList.SelectedIndices.Count - 1
            sourceList.Items.RemoveAt( sourceList.SelectedIndices(0))
        Next
    End Sub
```

Even if it's possible to remove an item by its value, this is not a safe approach. If two items have the same name, the `Remove` method will remove the first one. Unless you've provided the code to make sure that no identical items can be added to the list, remove them by their index, which is unique.

Notice that the code always removes the first item in the `SelectedIndices` collection. If you attempt to remove the item `SelectedIndices(i)`, you will remove the first selected item, but after that you will not remove all the selected items. After removing an item from the selection, the remaining items are no longer at the same locations. (In effect, you have to refresh the `SelectedIndices` collection.) The second selected item will take the place of the first selected item, which was just deleted, and so on. By removing the first item in the `SelectedIndices` collection, we make sure that all selected items, and only those items, will be eventually removed.

Moving Items between Lists

The two single-arrow buttons that are between the ListBox controls shown in Figure 6.6 transfer selected items from one list to another. The button with the single arrow pointing to the right transfers the items selected in the left list, after it ensures that the list contains at least one selected item. Its code is presented in Listing 6.12. First, it adds the item to the second list, and then it removes the item from the original list. Notice that the code removes an item by passing it as an argument to the `Remove` method because it doesn't make any difference which one of two identical objects will be removed.

LISTING 6.12: Moving the Selected Items

```
Private Sub bttnSourceMove_Click(...) _
              Handles bttnSourceMove.Click
    While sourceList.SelectedIndices.Count > 0
        destinationList.Items.Add(sourceList.Items( _
                    sourceList.SelectedIndices(0)))
        sourceList.Items.Remove(sourceList.Items( _
                    sourceList.SelectedIndices(0)))
    End While
End Sub
```

The second single-arrow button transfers items in the opposite direction. The destination control (the one on the right) doesn't allow the selection of multiple items, so you could use the `SelectedIndex` and `SelectedItem` properties. Because the single selected element is also part of the `SelectedItems` collection, you need not use a different approach. The statements that move a single item from the right to the left ListBox are shown next:

```
sourceList.Items.Add(destinationList.SelectedItem)
destinationList.Items.RemoveAt( _
              destinationList.SelectedIndex)
```

Searching the ListBox

Two of the most useful methods of the ListBox control are the `FindString` and `FindStringExact` methods, which allow you to quickly locate any item in the list. The `FindString` method locates a string that partially matches the one you're searching for; `FindStringExact` finds an exact match. If you're searching for *Man*, and the control contains a name such as *Mansfield*, `FindString` matches the item, but `FindStringExact` does not.

Both the `FindString` and `FindStringExact` methods perform case-insensitive searches. If you're searching for *visual*, and the list contains the item *Visual*, both methods will locate it. Their syntax is the same:

```
itemIndex = ListBox1.FindString(searchStr As String)
```

where `searchStr` is the string you're searching for. An alternative form of both methods allows you to specify the order of the item at which the search will begin:

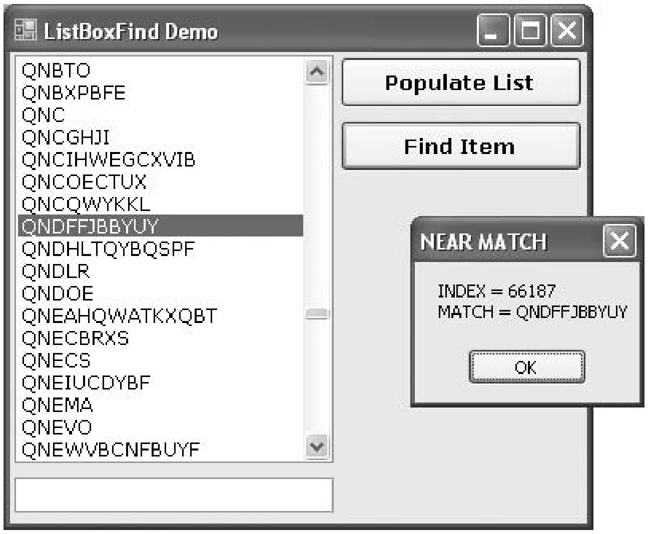
```
itemIndex = ListBox1.FindString(searchStr As String, _
                                 startIndex As Integer)
```

The `startIndex` argument allows you to specify the beginning of the search, but you can't specify where the search will end.

The `FindString` and `FindStringExact` methods work even if the ListBox control is not sorted. You need not set the `Sorted` property to True before you call one of the searching methods on the control. Sorting the list will help the search operation, but it takes the control less than 100 milliseconds to find an item in a list of 100,000 items, so time spent to sort the list isn't worth it. Before you load thousands of items in a ListBox control, however, you should probably consider a more-functional interface.

VB 2008 at Work: The ListBoxFind Application

The application you'll build in this section (seen in Figure 6.7) populates a list with a large number of items and then locates any string you specify. Click the button Populate List to populate the ListBox control with 10,000 random strings. This process will take a few seconds and will populate the control with different random strings every time. Then, you can enter a string in the TextBox control at the bottom of the form. As you type characters (or even delete characters in the TextBox), the program will locate the closest match in the list and select (highlight) this item.



```
QNBTO
QNBXPBFE
QNC
QNCGHJI
QNCIHWEGCXVIB
QNCOECTUX
QNCQWYKKL
QNDFFJBBYUY
QNDHLTQYBQSPF
QNDLR
QNDOE
QNEAHQWATKXQBT
QNECBRXS
QNECS
QNEIUCDYBF
QNEMA
QNEVO
QNEWVBCNFBUYF
```

```
INDEX = 66187
MATCH = QNDFFJBBYUY
```

Figure 6.7
The ListBoxFind application

The sample application reacts to each keystroke in the TextBox control and locates the string you're searching for instantly. The Find Item button does the same, but I thought I should demonstrate the efficiency of the ListBox control and the type of functionality you'd expect in a rich client application.

The code (shown in Listing 6.13) attempts to locate an exact match via the `FindStringExact` method. If it succeeds, it reports the index of the matching element. If not, it attempts to locate a near match with the `FindString` method. If it succeeds, it reports the index of the near match (which is the first item on the control that partially matches the search argument) and terminates. If it fails to find an exact match, it reports that the string wasn't found in the list.

LISTING 6.13: Searching the List

```
Private Sub TextBox1_TextChanged(...) Handles TextBox1.TextChanged
    Dim srchWord As String = TextBox1.Text.Trim
    If srchWord.Length = 0 Then Exit Sub
    Dim wordIndex As Integer
    wordIndex = ListBox1.FindStringExact(srchWord)
    If wordIndex >= 0 Then
        ListBox1.TopIndex = wordIndex
        ListBox1.SelectedIndex = wordIndex
    Else
        wordIndex = ListBox1.FindString(srchWord)
        If wordIndex >= 0 Then
            ListBox1.TopIndex = wordIndex
            ListBox1.SelectedIndex = wordIndex
        Else
            Debug.WriteLine("Item " & srchWord & _
                            " is not in the list")
        End If
    End If
End Sub
```

If you search for ***SAC***, for example, and the control contains a string such as "***SAC***" or "***sac***" or "***sAc***", the program will return the index of the item in the list and will report an exact match. If no exact match can be found, the program will return something like "***SACDEF***", if such a string exists on the control, as a near match. If none of the strings on the control starts with the characters ***SAC***, the search will fail.

Populating the List

The Populate List button creates 10,000 random items with the help of the Random class. First, it generates a random value in the range 1 through 20, which is the length of the string (not all strings have the same length). Then the program generates as many random characters as the length of the string and builds the string by appending each character to it. These random numbers are in the range of 65 to 91 and they're the ANSI values of the uppercase characters.

The ComboBox Control

The ComboBox control is similar to the ListBox control in the sense that it contains multiple items and the user may select one, but it typically occupies less space onscreen. The ComboBox is practically an expandable ListBox control, which can grow when the user wants to make a selection and retract after the selection is made. Normally, the ComboBox control displays one line with the selected item, as this control doesn't allow multiple item selection. The essential difference, however, between ComboBox and ListBox controls is that the ComboBox allows the user to specify items that don't exist in the list.

There are three types of ComboBox controls. The value of the control's `Style` property determines which box is used; these values are shown in Table 6.4.

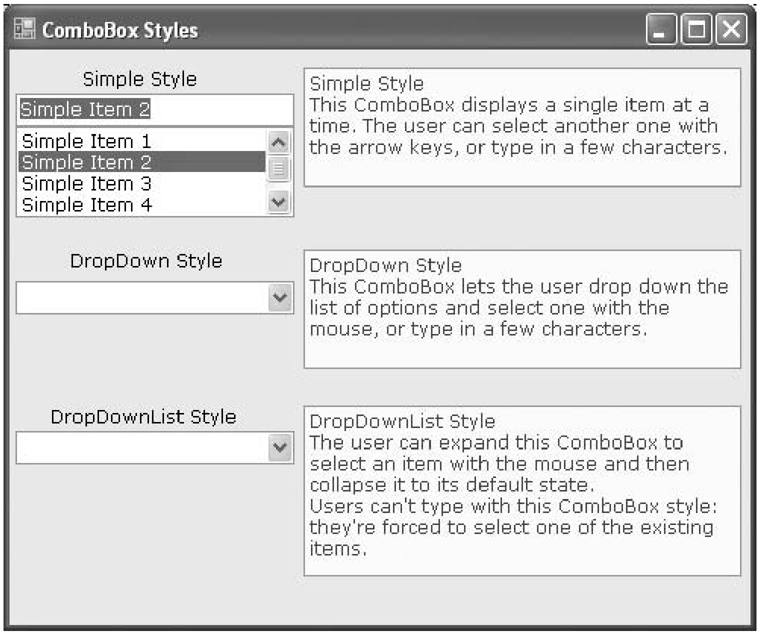
TABLE 6.4: Styles of the ComboBox Control

Value	Effect
DropDown	(Default) The control is made up of a drop-down list, which is visible at all times, and a text box. The user can select an item from the list or type a new one in the text box.
DropDownList	This style is a drop-down list from which the user can select one of its items but can't enter a new one. The control displays a single item, and the list is expanded as needed.
Simple	The control includes a text box and a list that doesn't drop down. The user can select from the list or type in the text box.

The ComboBox Styles project, shown in Figure 6.8, demonstrates the three styles of the ComboBox control. This is another common element of the Windows interface, and its properties and methods are identical to those of the ListBox control. Load the ComboBox Styles project in the Visual Basic IDE and experiment with the three styles of the ComboBox control.

The DropDown and Simple ComboBox controls allow the user to select an item from the list or enter a new one in the edit box of the control. Moreover, they're collapsed by default and they display a single item, unless the user expands the list of items to make a selection. The DropDownList ComboBox is similar to a ListBox control in the sense that it restricts the user to selecting an item (the user cannot enter a new one). However, it takes much less space on the form than a ListBox does, because normally it displays a single item. When the user wants to make a selection, the DropDownList expands to display more items. After the user has made a selection, the list contracts to a single line again.

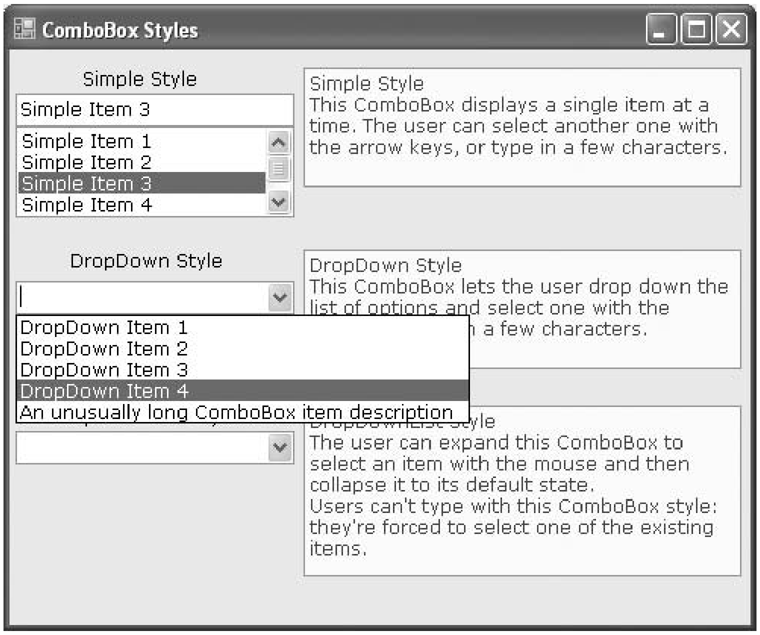
FIGURE 6.8A
The Simple ComboBox displays a fixed number of items at all times.



Simple Style
This ComboBox displays a single item at a time. The user can select another one with the arrow keys, or type in a few characters.

DropDown Style
This ComboBox lets the user drop down the list of options and select one with the mouse, or type in a few characters.

DropDownList Style
The user can expand this ComboBox to select an item with the mouse and then collapse it to its default state.
Users can't type with this ComboBox style: they're forced to select one of the existing items.

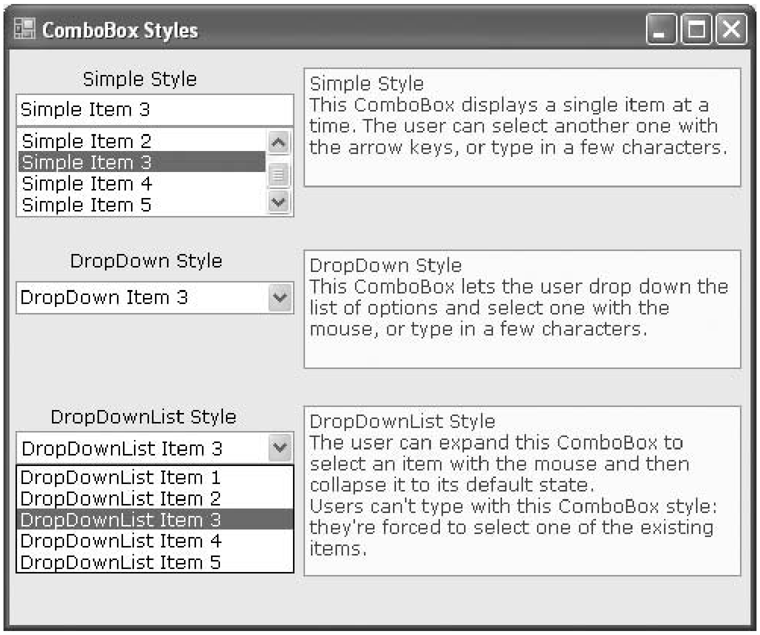


Simple Style
This ComboBox displays a single item at a time. The user can select another one with the arrow keys, or type in a few characters.

DropDown Style
This ComboBox lets the user drop down the list of options and select one with the … a few characters.

…yle
The user can expand this ComboBox to select an item with the mouse and then collapse it to its default state.
Users can't type with this ComboBox style: they're forced to select one of the existing items.

FIGURE 6.8B
The DropDown ComboBox displays a single item, and users can either expand the items or type something in the edit box.



Simple Style
This ComboBox displays a single item at a time. The user can select another one with the arrow keys, or type in a few characters.

DropDown Style
This ComboBox lets the user drop down the list of options and select one with the mouse, or type in a few characters.

DropDownList Style
The user can expand this ComboBox to select an item with the mouse and then collapse it to its default state.
Users can't type with this ComboBox style: they're forced to select one of the existing items.

FIGURE 6.8C
The DropDownList ComboBox expands to display its items, but doesn't allow users to type anything in the edit box.

Most of the properties and methods of the ListBox control also apply to the ComboBox control. The `Items` collection gives you access to the control's items, and the `SelectedIndices` and `SelectedItems` collections give you access to the items in the current selection. If the control allows only a single item to be selected, use the properties `SelectedIndex` and `SelectedItem`. You can also use the `FindString` and `FindStringExact` methods to locate any item in the control.

There's one aspect worth mentioning regarding the operation of the control. Although the edit box at the top allows you to enter a new string, the new string doesn't become a new item in the list. It remains there until you select another item or you clear the edit box. You can provide some code to make any string entered by the user in the control's edit box be added to the list of existing items.

The most common use of the ComboBox control is as a lookup table. The ComboBox control takes up very little space on the form, but it can be expanded at will. You can save even more space when the ComboBox is contracted by setting it to a width that's too small for the longest item. Use the `DropDownWidth` property, which is the width of the segment of the drop-down list. By default, this property is equal to the control's `Width` property. The second ComboBox control in Figure 6.8 contains an unusually long item. The control is wide enough to display the default selection. When the user clicks the arrow to expand the control, the drop-down section of the control is wider than the default width, so that the long items can be read.

Adding Items to a ComboBox at Runtime

Although the ComboBox control allows users to enter text in the control's edit box, it doesn't provide a simple mechanism for adding new items at runtime. Let's say you provide a ComboBox with city names. Users can type the first few characters and quickly locate the desired item. But what if you want to allow users to add new city names? You can provide this feature with two simple techniques. The simpler one is to place a button with an ellipsis (three periods) right next to the control. When users want to add a new item to the control, they can click the button and be prompted for the new item.

A more-elegant approach is to examine the control's `Text` property as soon as the control loses focus, or the user presses the Enter key. If the string entered by the user doesn't match an item on the control, you must add a new item to the control's `Items` collection and select the new item from within your code. The FlexComboBox project demonstrates how to use both techniques in your code. The main form of the project, which is shown in Figure 6.9, is a simple data-entry screen. It's not the best data-entry form, but it's meant for demonstration purposes.

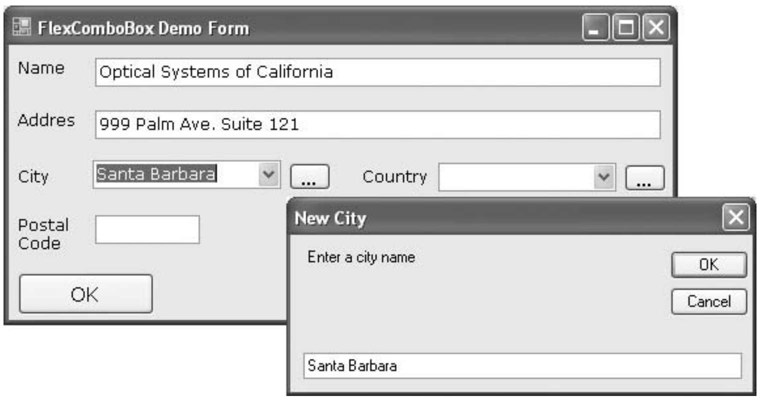


Figure 6.9
The FlexComboBox project demonstrates two techniques for adding new items to a ComboBox at runtime.

You can either enter a city name (or country name) and press the Tab key to move to another control or click the button next to the control to be prompted for a new city/country name. The application will let you enter any city/country combination. You should provide code to limit the cities within the selected country, but this is a nontrivial task. You also need to store the new city names entered on the first ComboBox control to a file (or a database table), so users will find them there the next time they execute the application. I haven't made the application elaborate; I've added the code only to demonstrate how to add new items to a ComboBox control at runtime.

VB 2008 At Work: The FlexCombo Project

The ellipsis button next to the *City* ComboBox control prompts the user for the new item via the `InputBox()` function. Then it searches the `Items` collection of the control via the `FindString` method, and if the new item isn't found, it's added to the control. Then the code selects the new item in the list. To do so, it sets the control's `SelectedIndex` property to the value returned by the `Items.Add` method, or the value returned by the `FindString` method, depending on whether the item was located or added to the list. Listing 6.14 shows the code behind the ellipsis button.

Listing 6.14: Adding a New Item to the ComboBox Control at Runtime

```
Private Sub Button1_Click(...) Button1.Click
   Dim itm As String
   itm = InputBox("Enter new item", "New Item")
   If itm.Trim <> "" Then AddElement(itm)
End Sub
```

The `AddElement()` subroutine, which accepts a string as an argument and adds it to the control, is shown in Listing 6.15. If the item doesn't exist in the control, it's added to the `Items` collection. If the item is a member of the `Items` collection, it's selected. As you will see, the same subroutine will be used by the second method for adding items to the control at runtime.

Listing 6.15: The *AddElement()* Subroutine

```
Sub AddElement(ByVal newItem As String)
    Dim idx As Integer
    If ComboBox1.FindString(newItem) > 0 Then
        idx = ComboBox1.FindString(newItem)
    Else
        idx = ComboBox1.Items.Add(newItem)
    End If
    ComboBox1.SelectedIndex = idx
End Sub
```

You can also add new items at runtime by adding the same code in the control's `LostFocus` event handler:

```
Private Sub ComboBox1_LostFocus(...) Handles ComboBox1.LostFocus
   Dim newItem As String = ComboBox1.Text
   AddElement(newItem)
End Sub
```

The ScrollBar and TrackBar Controls

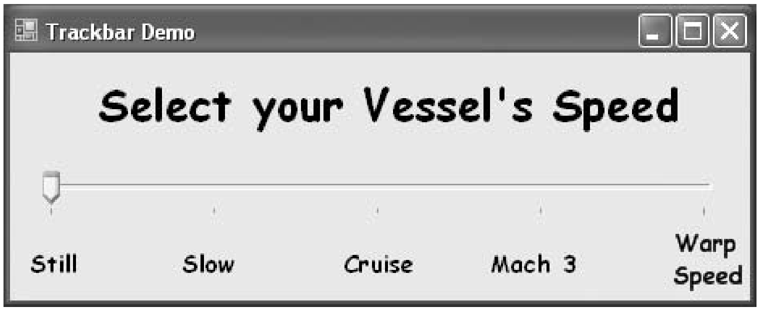
The ScrollBar and TrackBar controls let the user specify a magnitude by scrolling a selector between its minimum and maximum values. In some situations, the user doesn't know in advance the exact value of the quantity to specify (in which case, a text box would suffice), so your application must provide a more-flexible mechanism for specifying a value, along with some type of visual feedback.

The vertical scroll bar that lets a user move up and down a long document is a typical example of the use of the ScrollBar control. The scroll bar and visual feedback are the prime mechanisms for repositioning the view in a long document or in a large picture that won't fit entirely in its window.

The TrackBar control is similar to the ScrollBar control, but it doesn't cover a continuous range of values. The TrackBar control has a fixed number of tick marks, which the developer can label (for example, Still, Slow, and Warp Speed, as shown in Figure 6.10). Users can place the slider's indicator to the desired value. Whereas the ScrollBar control relies on some visual feedback outside the control to help the user position the indicator to the desired value, the TrackBar control forces the user to select from a range of valid values.

In short, the ScrollBar control should be used when the exact value isn't as important as the value's effect on another object or data element. The TrackBar control should be used when the user can type a numeric value and the value your application expects is a number in a specific range; for example, integers between 0 and 100, or a value between 0 and 5 inches in steps of 0.1 inches (0.0, 0.1, 0.2 . . . 5.0). The TrackBar control is preferred to the TextBox control in similar situations because there's no need for data validation on your part. The user can specify only valid numeric values with the mouse.

FIGURE 6.10
The TrackBar control lets the user select one of several discrete values.



The ScrollBar Control

There's no ScrollBar control per se in the Toolbox; instead, there are two versions of it: the HScrollBar and VScrollBar controls. They differ only in their orientation, but because they share the same members, I will refer to both controls collectively as ScrollBar controls. Actually, both controls inherit from the ScrollBar control, which is an abstract control: It can be used to implement vertical and horizontal scroll bars, but it can't be used directly on a form. Moreover, the HScrollBar and VScrollBar controls are not displayed in the Common Controls tab of the Toolbox. You have to open the All Windows Forms tab to locate these two controls.

The ScrollBar control is a long stripe with an indicator that lets the user select a value between the two ends of the control. The left (or bottom) end of the control corresponds to its minimum value; the other end is the control's maximum value. The current value of the control is determined by the position of the indicator, which can be scrolled between the minimum and maximum values. The basic properties of the ScrollBar control, therefore, are properly named `Minimum`, `Maximum`, and `Value`.

Minimum The control's minimum value. The default value is 0, but because this is an Integer value, you can set it to negative values as well.

Maximum The control's maximum value. The default value is 100, but you can set it to any value that you can represent with the Integer data type.

Value The control's current value, specified by the indicator's position.

The `Minimum` and `Maximum` properties are Integer values. To cover a range of nonintegers, you must supply the code to map the actual values to Integer values. For example, to cover a range from 2.5 to 8.5, set the `Minimum` property to 25, set the `Maximum` property to 85, and divide the control's value by 10. If the range you need is from –2.5 to 8.5, do the same but set the `Minimum` property to –25 and the `Maximum` value to 85, and divide the `Value` property by 10.

There are two more properties that allow you to control the movement of the indicator: the `SmallChange` and `LargeChange` properties. The first property is the amount by which the indicator changes when the user clicks one of the arrows at the two ends of the control. The `LargeChange` property is the displacement of the indicator when the user clicks somewhere in the scroll bar itself. You can manipulate a scroll bar by using the keyboard as well. Press the arrow keys to move the indicator in the corresponding direction by `SmallChange`, and the PageUp/PageDown keys to move the indicator by `LargeChange`.

VB 2008 AT WORK: THE COLORS PROJECT

Figure 6.11 shows the main form of the Colors sample project, which lets the user specify a color by manipulating the value of its basic colors (red, green, and blue) through scroll bars. Each basic color is controlled by a scroll bar and has a minimum value of 0 and a maximum value of 255. If you aren't familiar with color definition in the Windows environment, see the section ''Specifying Colors'' in Chapter 19, ''Manipulating Images and Bitmaps.''

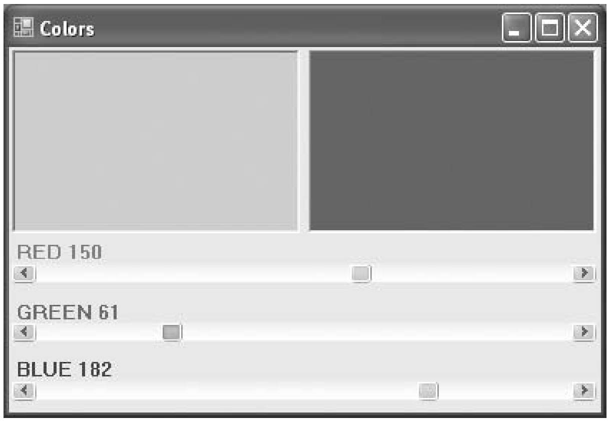


FIGURE 6.11
The Colors application demonstrates the use of the ScrollBar control.

As the scroll bar is moved, the corresponding color is displayed, and the user can easily specify a color without knowing the exact values of its primary components. All the user needs to know is whether the desired color contains, for example, too much red or too little green. With the help of the scroll bars and the immediate feedback from the application, the user can easily pinpoint the desired color. Notice that the exact values of the color's basic components are of no practical interest; only the final color counts.

The ScrollBar Control's Events

The user can change the ScrollBar control's value in three ways: by clicking the two arrows at its ends, by clicking the area between the indicator and the arrows, and by dragging the indicator with the mouse. You can monitor the changes of the ScrollBar's value from within your code by using two events: `ValueChanged` and `Scroll`. Both events are fired every time the indicator's position is changed. If you change the control's value from within your code, only the `ValueChanged` event will be fired.

The `Scroll` event can be fired in response to many different actions, such as the scrolling of the indicator with the mouse, a click on one of the two buttons at the ends of the scroll bars, and so on. If you want to know the action that caused this event, you can examine the `Type` property of the second argument of the event handler. The settings of the `e.Type` property are members of the `ScrollEventType` enumeration (`LargeDecrement`, `SmallIncrement`, `Track`, and so on).

Handling the Events in the Colors Application

The Colors application demonstrates how to program the two events of the ScrollBar control. The two PictureBox controls display the color designed with the three scroll bars. The left PictureBox is colored from within the `Scroll` event, whereas the other one is colored from within the `ValueChanged` event. Both events are fired as the user scrolls the scrollbar's indicator, but in the `Scroll` event handler of the three scroll bars, the code examines the value of the `e.Type` property and reacts to it only if the event was fired because the scrolling of the indicator has ended. For all other actions, the event handler doesn't update the color of the left PictureBox.

If the user attempts to change the Color value by clicking the two arrows of the scroll bars or by clicking in the area to the left or to the right of the indicator, both PictureBox controls are updated. While the user slides the indicator or keeps pressing one of the end arrows, only the PictureBox to the right is updated.

The conclusion from this experiment is that you can program either event to provide continuous feedback to the user. If this feedback requires too many calculations, which would slow down the reaction of the corresponding event handler, you can postpone the reaction until the user has stopped scrolling the indicator. You can detect this condition by examining the value of the `e.Type` property. When it's `ScrollEventType.EndScroll`, you can execute the appropriate statements. Listing 6.16 shows the code behind the `Scroll` and `ValueChanged` events of the scroll bar that controls the red component of the color. The code of the corresponding events of the other two controls is identical.

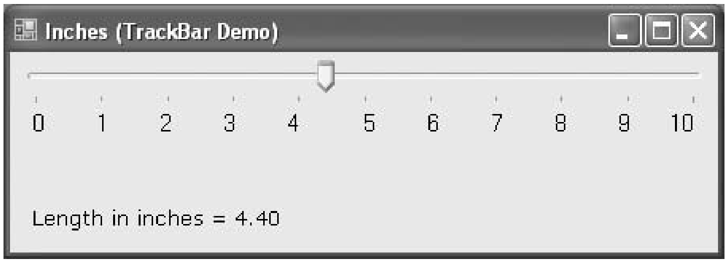
Listing 6.16: Programming the ScrollBar Control's *Scroll* Event

```
Private Sub redBar_Scroll(...) Handles redBar.Scroll
    If e.Type = ScrollEventType.EndScroll Then
        ColorBox1()
        lblRed.Text = "RED " & redBar.Value.ToString("###")
    End If
End Sub

Private Sub redBar_ValueChanged(...) Handles redBar.ValueChanged
    ColorBox2()
End Sub
```

The `ColorBox1()` and `ColorBox2()` subroutines update the color of the two PictureBox controls by setting their background colors. You can open the Colors project in Visual Studio and examine the code of these two routines.

The TrackBar Control

The TrackBar control is similar to the ScrollBar control, but it lacks the granularity of ScrollBar. Suppose that you want the user of an application to supply a value in a specific range, such as the speed of a moving object. Moreover, you don't want to allow extreme precision; you need only a few settings, as shown in the examples of Figures 6.10 and 6.12. The user can set the control's value by sliding the indicator or by clicking on either side of the indicator.



Length in inches = 4.40

FIGURE 6.12 The Inches application demonstrates the use of the TrackBar control in specifying an exact value in a specific range.

Granularity is how specific you want to be in measuring. In measuring distances between towns, a granularity of a mile is quite adequate. In measuring (or specifying) the dimensions of a building, the granularity could be on the order of a foot or an inch. The TrackBar control lets you set the type of granularity that's necessary for your application.

Similar to the ScrollBar control, `SmallChange` and `LargeChange` properties are available. `SmallChange` is the smallest increment by which the Slider value can change. The user can change the slider by the `SmallChange` value only by sliding the indicator. (Unlike the ScrollBar control, there are no arrows at the two ends of the Slider control.) To change the Slider's value by `LargeChange`, the user can click on either side of the indicator.

VB 2008 at Work: The Inches Project

Figure 6.12 demonstrates a typical use of the TrackBar control. The form in the figure is an element of a program's user interface that lets the user specify a distance between 0 and 10 inches in increments of 0.2 inches. As the user slides the indicator, the current value is displayed on a Label control below the TrackBar. If you open the Inches application, you'll notice that there are more stops than there are tick marks on the control. This is made possible with the `TickFrequency` property, which determines the frequency of the visible tick marks.

You might specify that the control has 50 stops (divisions), but that only 10 of them will be visible. The user can, however, position the indicator on any of the 40 invisible tick marks. You can think of the visible marks as the major tick marks, and the invisible ones as the minor tick marks. If the `TickFrequency` property is 5, only every fifth mark will be visible. The slider's indicator, however, will stop at all tick marks.

When using the TrackBar control on your interfaces, you should set the `TickFrequency` property to a value that helps the user select the desired setting. Too many tick marks are confusing and difficult to read. Without tick marks, the control isn't of much help. You might also consider placing a few labels to indicate the value of selected tick marks, as I have done in this example.

The properties of the TrackBar control in the Inches application are as follows:

`Minimum` = 0
`Maximum` = 50
`SmallChange` = 1
`LargeChange` = 5
`TickFrequency` = 5

The TrackBar needs to cover a range of 10 inches in increments of 0.2 inches. If you set the `SmallChange` property to 1, you have to set `LargeChange` to 5. Moreover, the `TickFrequency` is set to 5, so there will be a total of five divisions in every inch. The numbers below the tick marks were placed there with properly aligned Label controls.

The label at the bottom needs to be updated as the TrackBar's value changes. This is signaled to the application with the `Change` event, which occurs every time the value of the control changes, either through scrolling or from within your code. The `ValueChanged` event handler of the TrackBar control is shown next:

```
Private Sub TrackBar1_ValueChanged(...) _
              Handles TrackBar1.ValueChanged
   lblInches.Text = "Length in inches = " & _
              Format(TrackBar1.Value / 5, "#.00")
End Sub
```

The Label controls below the tick marks can also be used to set the value of the control. Every time you click one of the labels, the following statement sets the TrackBar control's value. Notice that all the Label controls' `Click` events are handled by a common handler:

```
Private Sub Label_Click(...) _
                Handles Label1.Click, Label9.Click
    TrackBar1.Value = sender.text * 5
End Sub
```

The Bottom Line

Use the TextBox control as a data-entry and text-editing tool. The TextBox control is the most common element of the Windows interface, short of the Button control, and it's used to display and edit text. You can use a TextBox control to prompt users for a single line of text (such as a product name) or a small document (a product's detailed description).

Master It What are the most important properties of the TextBox control? Which ones would you set in the Properties windows at design-time?

Master It How will you implement a control that suggests lists of words matching the characters entered by the user?

Use the ListBox, CheckedListBox, and ComboBox controls to present lists of items. The ListBox control contains a list of items from which the user can select one or more, depending on the setting of the `SelectionMode` property.

Master It How will you locate an item in a ListBox control?

Use the ScrollBar and TrackBar controls to enable users to specify sizes and positions with the mouse. The ScrollBar and TrackBar controls let the user specify a magnitude by scrolling a selector between its minimum and maximum values. The ScrollBar control uses some visual feedback to display the effects of scrolling on another entity, such as the current view in a long document.

Master It Which event of the ScrollBar control would you code to provide visual feedback to the user?

Chapter 7

Working with Forms

In Visual Basic, the *form* is the container for all the controls that make up the user interface. When a Visual Basic application is executing, each window it displays on the desktop is a form. The terms *form* and *window* describe the same entity. A window is what the user sees on the desktop when the application is running. A form is the same entity at design time. The proper term is a *Windows form*, as opposed to a *web form*, but I will refer to them as *forms*. This term includes both the regular forms and dialog boxes, which are simple forms you use for very specific actions, such as to prompt the user for a particular piece of data or to display critical information. A *dialog box* is a form with a small number of controls, no menus, and usually an OK and a Cancel button to close it.

Forms have a built-in functionality that is always available without any programming effort on your part. You can move a form around, resize it, and even cover it with other forms. You do so with the mouse or with the keyboard through the Control menu.

In previous chapters, you concentrated on placing the elements of the user interface on forms, setting their properties, and adding code behind selected events. Now, you'll look at forms themselves and at a few related topics. In this chapter, you'll learn how to do the following:

- Use forms' properties
- Design applications with multiple forms
- Design dynamic forms
- Design menus

Forms have many trivial properties that won't be discussed here. Instead, let's jump directly to the properties that are unique to forms and then look at how to manipulate forms from within an application's code.

The Appearance of Forms

Applications are made up of one or more forms — usually more than one. You should craft your forms carefully, make them functional, and keep them simple and intuitive. You already know how to place controls on the form, but there's more to designing forms than populating them with controls. The main characteristic of a form is the title bar on which the form's caption is displayed (see Figure 7.1).

Clicking the icon on the left end of the title bar opens the Control menu, which contains the commands shown in Table 7.1. On the right end of the title bar are three buttons: Minimize, Maximize, and Close. Clicking these buttons performs the associated function. When a form is maximized, the Maximize button is replaced by the Restore button. When clicked, the Restore button resets the form to the size and position before it was maximized, and it's replaced by the Maximize button. To access the Control menu without a mouse, press Alt and then the down arrow key.

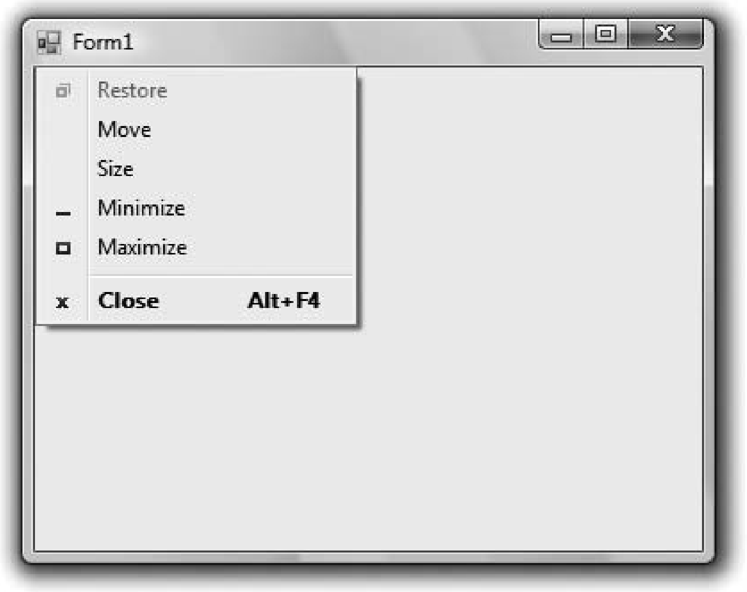


Figure 7.1
The elements of the form

Table 7.1: Commands of the Control Menu

Command	**Effect**
Restore	Restores a maximized form to the size it was before it was maximized; available only if the form has been maximized.
Move	Lets the user move the form around with the arrow keys.
Size	Lets the user resize the form with the arrow keys.
Minimize	Minimizes the form.
Maximize	Maximizes the form.
Close	Closes the current form. (Closing the application's main form terminates the application.)

Properties of the Form Object

You're familiar with the appearance of forms, even if you haven't programmed in the Windows environment in the past; you have seen nearly all types of windows in the applications you're using every day. The floating toolbars used by many graphics applications, for example, are actually forms with a narrow title bar. The dialog boxes that display critical information or prompt you to select the file to be opened are also forms. You can duplicate the look of any window or dialog box through the following properties of the Form object.

AcceptButton, CancelButton

These two properties let you specify the default Accept and Cancel buttons. The Accept button is the one that's automatically activated when you press Enter, no matter which control has the focus at the time, and is usually the button with the OK caption. Likewise, the Cancel button is the one that's automatically activated when you hit the Esc key and is usually the button with the Cancel caption. To specify the Accept and Cancel buttons on a form, locate the `AcceptButton` and `CancelButton` properties of the form and select the corresponding controls from a drop-down list, which contains the names of all the buttons on the form. For more information on these two properties, see the section ''Forms versus Dialog Boxes,'' later in this chapter.

AutoScaleMode

This property determines how the control is scaled, and its value is a member of the `AutoScaleMode` enumeration: *`None`* (automatic scaling is disabled), *`Font`* (the controls on the form are scaled relative to the size of their font), *`Dpi`*, which stands for dots per inch (the controls on the form are scaled relative to the display resolution), and *`Inherit`* (the controls are scaled according to the *`AutoScaleMode`* property of their parent class). The default value is *`Font`*; if you change the form's font size, the controls on it are scaled to the new font size.

AutoScroll

The `AutoScroll` property is a True/False value that indicates whether scroll bars will be automatically attached to the form (as seen in Figure 7.2) if the form is resized to a point that not all its controls are visible. Use this property to design large forms without having to worry about the resolution of the monitor on which they'll be displayed. The `AutoScroll` property is used in conjunction with two other properties (described a little later in this section): `AutoScrollMargin` and `AutoScrollMinSize`. Note that the `AutoScroll` property applies to a few controls as well, including the Panel and SplitContainer controls. For example, you can create a form with a fixed and a scrolling pane by placing two Panel controls on it and setting the `AutoScroll` property of one of them (the Panel you want to scroll) to True.

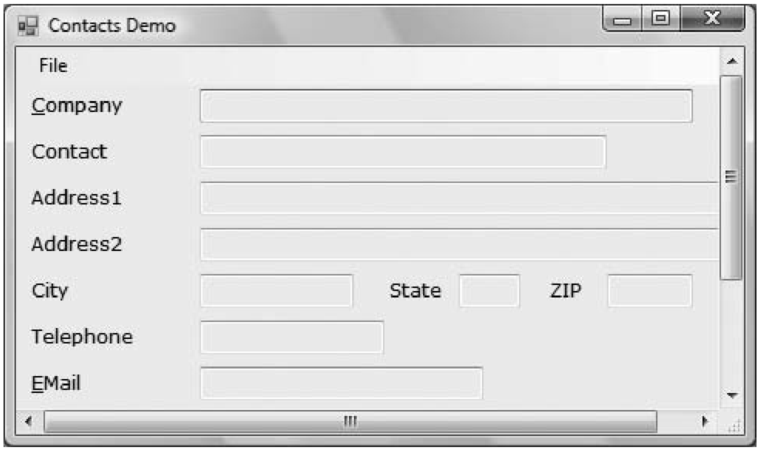
AutoScrollPosition

This property is available from within your code only (you can't set this property at design time), and it indicates the number of pixels that the form was scrolled up or down. Its initial value is zero, and it assumes a value when the user scrolls the form (provided that the form's `AutoScroll` property is True). Use this property to find out the visible controls from within your code, or scroll the form programmatically to bring a specific control into view.

AutoScrollMargin

This is a margin, expressed in pixels, that's added around all the controls on the form. If the form is smaller than the rectangle that encloses all the controls adjusted by the margin, the appropriate scroll bar(s) will be displayed automatically.

FIGURE 7.2
If the controls don't fit in the form's visible area, scroll bars can be attached automatically.



AUTOSCROLLMINSIZE

This property lets you specify the minimum size of the form before the scroll bars are attached. If your form contains graphics that you want to be visible at all times, set the `Width` and `Height` members of the `AutoScrollMinSize` property to the dimensions of the graphics. (Of course, the graphics won't be visible at all times, but the scroll bars indicate that there's more to the form than can fit in the current window.) Notice that this isn't the form's minimum size; users can make the form even smaller. To specify a minimum size for the form, use the `MinimumSize` property, described later in this section.

Let's say the `AutoScrollMargin` property of the form is 180 × 150. If the form is resized to fewer than 180 pixels horizontally or 150 pixels vertically, the appropriate scroll bars will appear automatically, as long as the `AutoScroll` property is True. If you want to enable the `Auto-Scroll` feature when the form's width is reduced to anything fewer than 250 pixels, set the `AutoScrollMinSize` property to (250, 0). In this example, setting `AutoScrollMinSize.Width` to anything less than 180, or `AutoScrollMinSize.Height` to anything less than 150, will have no effect on the appearance of the form and its scroll bars.

BRINGING SELECTED CONTROLS INTO VIEW

In addition to the `Autoscroll` properties, the Form object provides the `Scroll` method, which allows you to scroll a form programmatically, and `ScrollControlIntoView`, which scrolls the form until the specified control comes into view. The `Scroll` method accepts as arguments the horizontal and vertical displacements of the scrolling operation, whereas `ScrollControlIntoView` accepts as an argument the control you want to bring into view. Notice that activating a control with the Tab key automatically brings the control into view if it's not already visible on the form. Finally, the `Scroll` event is fired every time a form is scrolled.

FORMBORDERSTYLE

The `FormBorderStyle` property determines the style of the form's border; its value is one of the `FormBorderStyle` enumeration's members, which are shown in Table 7.2. You can make the form's title bar disappear altogether by setting the form's `FormBorderStyle` property to

`FixedToolWindow`, the `ControlBox` property to False, and the `Text` property (the form's caption) to an empty string. However, a form like this can't be moved around with the mouse and will probably frustrate users.

TABLE 7.2: The *FormBorderStyle* Enumeration

VALUE	EFFECT
None	A borderless window that can't be resized. This setting is rarely used.
Sizable	(default) A resizable window that's used for displaying regular forms.
Fixed3D	A window with a fixed visible border, ''raised'' relative to the main area. Unlike the None setting, this setting allows users to minimize and close the window.
FixedDialog	A fixed window used to implement dialog boxes.
FixedSingle	A fixed window with a single-line border.
FixedToolWindow	A fixed window with a Close button only. It looks like a toolbar displayed by drawing and imaging applications.
SizableToolWindow	Same as the FixedToolWindow, but is resizable. In addition, its caption font is smaller than the usual.

ControlBox

This property is also True by default. Set it to False to hide the control box icon and disable the Control menu. Although the Control menu is rarely used, Windows applications don't disable it. When the `ControlBox` property is False, the three buttons on the title bar are also disabled. If you set the `Text` property to an empty string, the title bar disappears altogether.

MinimizeBox, MaximizeBox

These two properties, which specify whether the Minimize and Maximize buttons will appear on the form's title bar, are True by default. Set them to False to hide the corresponding buttons on the form's title bar.

MinimumSize, MaximumSize

These two properties read or set the minimum and maximum size of a form. When users resize the form at runtime, the form won't become any smaller than the dimensions specified by the `MinimumSize` property and no larger than the dimensions specified by the `MaximumSize` property. The `MinimumSize` property is a Size object, and you can set it with a statement like the following:

```
Me.MinimumSize = New Size(400, 300)
```

Or you can set the width and height separately:

```
Me.MinimumSize.Width = 400
Me.MinimumSize.Height = 300
```

The `MinimumSize.Height` property includes the height of the form's title bar; you should take that into consideration. If the minimum usable size of the form is 400 × 300, use the following statement to set the `MinimumSize` property:

```
Me.MinimumSize = New Size(400, 300 + SystemInformation.CaptionHeight)
```

The default value of both properties is (0, 0), which means that no minimum or maximum size is imposed on the form, and the user can resize it as desired.

Use the SystemInformation Class to Read System Information

The height of the caption is not a property of the Form object, even though it's used to determine the useful area of the form (the total height minus the caption bar). Keep in mind that the height of the caption bar is given by the `CaptionHeight` property of the SystemInformation object. You should look up the SystemInformation object, which exposes a lot of useful properties — such as `BorderSize` (the size of the form's borders), `Border3DSize` (the size of three-dimensional borders), `CursorSize` (the cursor's size), and many more.

KeyPreview

This property enables the form to capture all keystrokes before they're passed to the control that has the focus. Normally, when you press a key, the `KeyPress` event of the control with the focus is triggered (as well as the `KeyUp` and `KeyDown` events), and you can handle the keystroke from within the control's appropriate handler. In most cases, you let the control handle the keystroke and don't write any form code for that.

If you want to use ''universal'' keystrokes in your application, you must set the `KeyPreview` property to True. Doing so enables the form to intercept all keystrokes, so you can process them from within the form's keystroke event handlers. To handle a specific keystroke at the form's level, set the form's `KeyPreview` property to True and insert the appropriate code in the form's `KeyDown` or `KeyUp` event handler (the `KeyPress` event isn't fired for the function keys).

The same keystrokes are then passed to the control with the focus, unless you ''kill'' the keystroke by setting its `SuppressKeystroke` property to True when you process it on the form's level. For more information on processing keystrokes at the form level and using special keystrokes throughout your application, see the Contacts project later in this chapter.

SizeGripStyle

This property gets or sets the style of the sizing handle to display in the bottom-right corner of the form. You can set it to a member of the `SizeGripStyle` enumeration: *`Auto`* (the size grip is displayed as needed), *`Show`* (the size grip is displayed at all times), or *`Hide`* (the size grip is not displayed, but users can still resize the form with the mouse).

StartPosition, Location

The `StartPosition` property, which determines the initial position of the form when it's first displayed, can be set to one of the members of the `FormStartPosition` enumeration: *`CenterParent`* (the form is centered in the area of its parent form), *`CenterScreen`* (the form is centered on the monitor), *`Manual`* (the position of the form is determined by the `Location` property), *`WindowsDefaultLocation`* (the form is positioned at the Windows default location), and

WindowsDefaultBound (the form's location and bounds are determined by Windows defaults). The Location property allows you to set the form's initial position at design time or to change the form's location at runtime.

TopMost

This property is a True/False value that lets you specify whether the form will remain on top of all other forms in your application. Its default property is False, and you should change it only on rare occasions. Some dialog boxes, such as the Find & Replace dialog box of any text-processing application, are always visible, even when they don't have the focus. For more information on using the TopMost property, see the discussion of the TextPad project in Chapter 6, ''Basic Windows Controls.'' You can also add a professional touch to your application by providing a CheckBox control that determines whether a form should remain on top of all other forms of the application.

Size

Use the Size property to set the form's size at design time or at runtime. Normally, the form's width and height are controlled by the user at runtime. This property is usually set from within the form's Resize event handler to maintain a reasonable aspect ratio when the user resizes the form. The Form object also exposes the Width and Height properties for controlling its size.

Placing Controls on Forms

The first step in designing your application's interface is, of course, the analysis and careful planning of the basic operations you want to provide through your interface. The second step is to design the forms. Designing a form means placing Windows controls on it, setting the controls' properties, and then writing code to handle the events of interest. Visual Studio 2008 is a *rapid application development (RAD)* environment. This doesn't mean that you're expected to develop applications rapidly. It has come to mean that you can rapidly prototype an application and show something to the customer. And this is made possible through the visual tools that come with VS 2008, especially the new Form Designer.

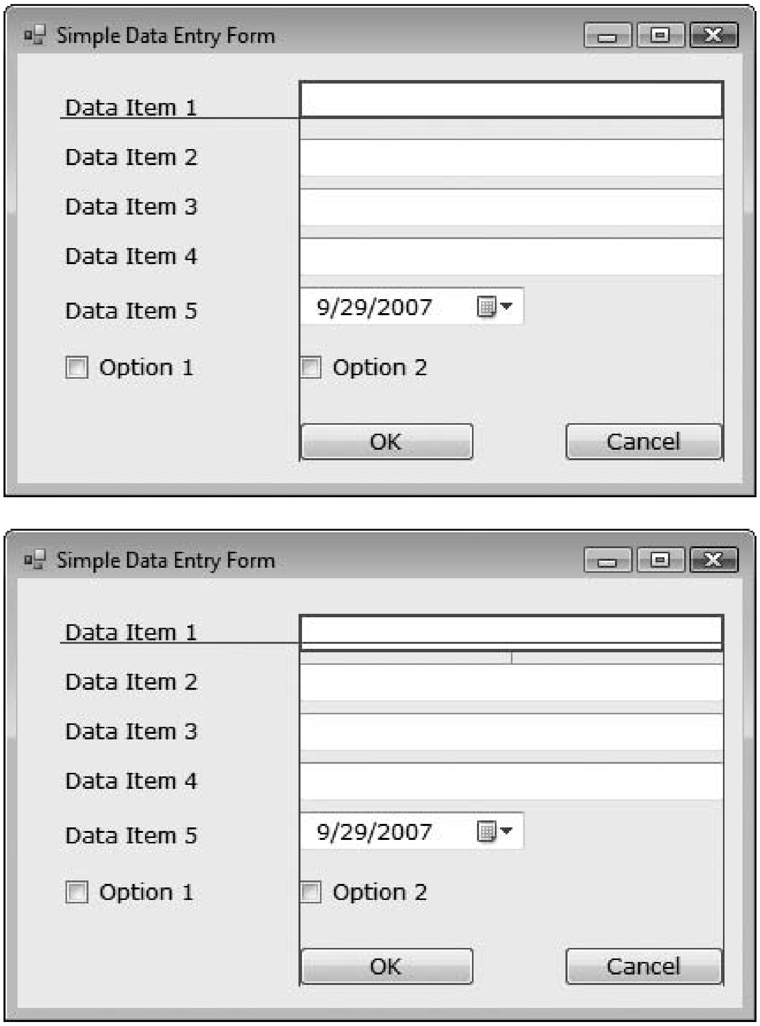
To place controls on your form, you select them in the Toolbox and then draw, on the form, the rectangle in which the control will be enclosed. Or you can double-click the control's icon to place an instance of the control on the form. All controls have a default size, and you can resize the control on the form by using the mouse.

Each control's dimensions can also be set in the Properties window through the Size property. The Size property exposes the Width and Height components, which are expressed in pixels. Likewise, the Location property returns (or sets) the coordinates of the top-left corner of the control. In the section ''Building Dynamic Forms at Runtime,'' later in this chapter, you'll see how to create new controls at runtime and place them in a specific location on a form from within your code.

As you place controls on the form, you can align them in groups by using the commands of the Format menu. Select multiple controls on the form by using the mouse and the Shift (or Ctrl) key, and then align their edges or their middles with the appropriate command of the Format menu. To align the left edges of a column of TextBoxes, choose the Format ➢ Align ➢ Left command. You can also use the commands of the Format ➢ Make Same Size command to adjust the dimensions of the selected controls. (To make them equal in size, make their widths or heights equal.)

As you move controls around with the mouse, a blue snap line appears when the controls become nearly aligned with another control. Release the mouse while the snap line is visible to leave the control aligned with the one indicated by the snap lines. The blue snap lines indicate edge alignment. Most of the time, we need to align not the edges of two controls, but their baselines (the baseline of the text on the control). The snap lines that indicate baseline alignment are red. Figure 7.3 shows both types of snap lines. When we're aligning a Label control with its matching TextBox control on a form, we want to align their baselines, not their frames (especially if you consider that the Label controls are always displayed without borders). If the control is aligned with other controls in both directions, two snap lines will appear — a horizontal one and a vertical one.

Figure 7.3
Edge alignment (top) and baseline alignment (bottom)

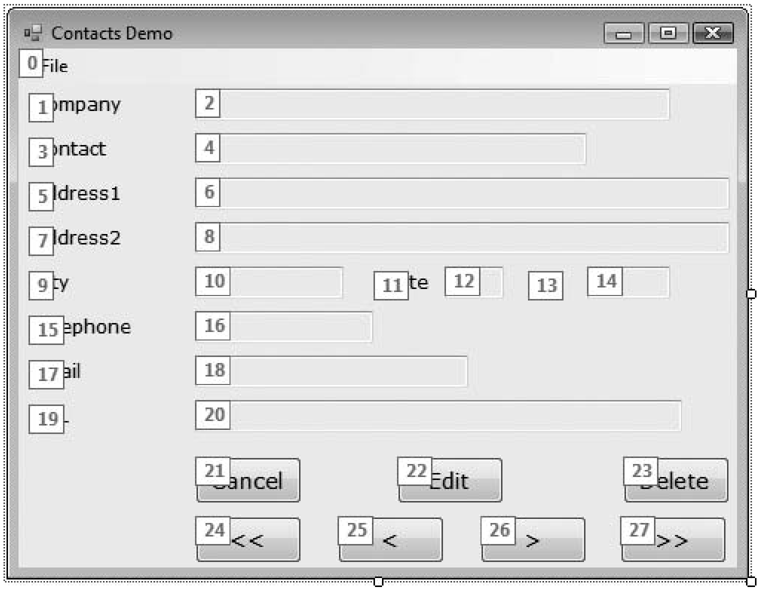


Setting the *TabOrder* Property

Another important issue in form design is the tab order of the controls on the form. As you know, pressing the Tab key at runtime takes you to the next control on the form. The order of the controls is the order in which they were placed on the form, but this is never what we want. When you design the application, you can specify in which order the controls receive the focus (the *tab order*, as it is known) with the help of the `TabOrder` property. Each control has its own `TabOrder` setting, which is an integer value. When the Tab key is pressed, the focus is moved to the control whose tab order immediately follows the tab order of the current control. The values of the `TabOrder` properties of the various controls on the form need not be consecutive.

To specify the tab order of the various controls, you can set their `TabOrder` property in the Properties window or you can choose the Tab Order command from the View menu. The tab order of each control will be displayed on the corresponding control, as shown in Figure 7.4. (The form shown in the figure is the Contacts application, which is discussed shortly.)

FIGURE 7.4
Setting the tab order of the controls on the main form of the Contacts project



To set the tab order of the controls, click each control in the order in which you want them to receive the focus. You must click all of them in the desired order, starting with the first control in the tab order. Each control's index in the tab order appears in the upper-left corner of the control. When you're finished, choose the Tab Order command from the View menu again to hide these numbers.

As you place controls on the form, don't forget to lock them, so that you won't move them around by mistake as you work with other controls. You can lock the controls in their places either by setting each control's `Locked` property to True or by locking all the controls on the form at once via the Format ➢ Lock Controls command.

Real World Scenario

DESIGN WITH THE USER IN MIND

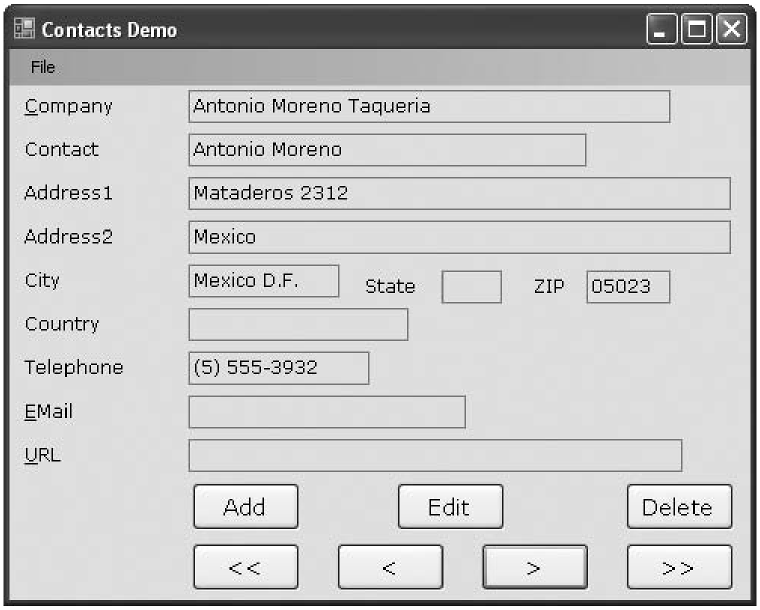
Designing functional forms is a crucial step in the process of developing Windows applications. Most data-entry operators don't work with the mouse, and you must make sure that all the actions (such as switching to another control, opening a menu, clicking a button, and so on) can be performed with the keyboard. This requirement doesn't apply to graphics applications, of course, but most applications developed with VB are business applications, and users should be able to perform most of the tasks with the keyboard, not with the mouse.

In my experience, the most important aspect of the user interface of a business application is the handling of the Enter keystroke. When a TextBox control has the focus, the Enter keystroke should advance the focus to the next control in the tab order; when a list control (such as the ListBox or ListView control) has the focus, the Enter keystroke should invoke the same action as double-clicking the current item. The sample project in the following section demonstrates many of the features you'd expect from a data-entry application.

If you're developing a data-entry form, you must take into consideration the needs of the users. Make a prototype and ask the people who will use the application to test-drive it. Listen to their objections carefully, collect all the information, and then use it to refine your application's user interface. Don't defend your design — just learn from the users. They will uncover all the flaws of the application and they'll help you design the most functional interface. In addition, they will accept the finished application with fewer objections and complaints if they know what to expect.

VB 2008 at Work: The Contacts Project

I want to conclude this section with a simple data-entry application that demonstrates many of the topics discussed here, as well as a few techniques for designing easy-to-use forms. Figure 7.5 shows a data-entry form for maintaining contact information, and I'm sure you will add your own fields to make this application more useful.



Company	Antonio Moreno Taqueria
Contact	Antonio Moreno
Address1	Mataderos 2312
Address2	Mexico
City	Mexico D.F.
State	
ZIP	05023
Country	
Telephone	(5) 555-3932
EMail	
URL	

FIGURE 7.5
A simple data-entry screen

You can navigate through the contacts by clicking the buttons with the arrows, as well as add new contacts or delete existing ones by clicking the appropriate buttons. When you're entering a new contact, the buttons shown in Figure 7.5 are replaced by the usual OK and Cancel buttons. The action of adding a new contact, or editing an existing one, must end by clicking one of these two buttons. After committing a new contact or canceling the action, the usual navigation buttons appear again.

Place the controls you see in Figure 7.5 on the form and align them appropriately. After the controls are on the form, the next step is to set their tab order. You must specify a `TabOrder` even for controls that never receive focus, such as the Label controls. In addition to the tab order of the controls, we'll also use shortcut keys to give the user quick access to the most common fields. The shortcut keys are displayed as underlined characters on the corresponding labels. Notice that the Label controls have shortcut keys, even though they don't receive the focus. When you press the shortcut key of a Label, the focus is moved to the following control in the tab order, which is the TextBox control next to it.

If you run the application now, you'll see that the focus moves from one TextBox to the next and that the labels are skipped. After the last TextBox control, the focus is moved to the buttons and then back to the first TextBox control. To add a shortcut key for the most common fields, determine which fields will have shortcut keys and then which keys will be used for that purpose. Being the Internet buffs that we all are, let's assign shortcut keys to the Company, EMail, and URL fields. Locate each label's `Text` property in the Properties window and insert the & symbol in front of the character you want to act as a shortcut for each Label. The `Text` properties of the three controls should be **`&Company`**, **`&EMail`**, and **`&URL`**.

Shortcut keys are activated at runtime by pressing the shortcut character while holding down the Alt key. The shortcut key will move the focus to the corresponding Label control, but because labels can't receive the focus, the focus is moved immediately to the next control in the tab order, which is the adjacent TextBox control.

The contacts are stored in an ArrayList object, which is similar to an array but a little more convenient. We'll discuss ArrayLists in Chapter 14, ''Storing Data in Collections''; for now, you can ignore the parts of the application that manipulate the contacts and focus on the design issues.

Start by loading the sample data included with the application. Open the File menu and choose Load. You won't be prompted for a filename; the application always opens the same file in its root folder. After reading about the OpenFileDialog and SaveFileDialog controls, you can modify the code so that it prompts the user about the file to read from or write to. Then enter a new contact by clicking the Add button or edit an existing contact by clicking the Edit button. Both actions must end with the OK or Cancel button. In other words, we require users to explicitly end the operation, and we won't allow them to switch to another contact while adding or editing one.

The code behind the various buttons is straightforward. The Add button hides all the navigational buttons at the bottom of the form and clears the TextBoxes. The OK button saves the new contact to an ArrayList structure and redisplays the navigational buttons. The Cancel button ignores the data entered by the user and likewise displays the navigational buttons. In all cases, when the user switches back to the view mode, the TextBoxes are also locked, by setting their `ReadOnly` properties to True.

Handling Keystrokes

Although the Tab key is the Windows method of moving to the next control on the form, most users will find it more convenient to use the Enter key. The Enter key is the most important one on the keyboard, and applications should handle it intelligently. When the user presses Enter in a single-line TextBox, for example, the obvious action is to move the focus to the following control. I included a few statements in the `KeyDown` event handlers of the TextBox controls to move the focus to the following one:

```
Private Sub txtAddress1_KeyDown(...) Handles txtAddress1.KeyDown
    If e.KeyData = Keys.Enter Then
```

```
            e.SuppressKeyPress = True
            txtAddress2.Focus()
        End If
    End Sub
```

If you use the KeyUp event handler instead, the result won't be any different, but an annoying beeping sound will be emitted with each keystroke. The beep occurs when the button is depressed, so we must intercept the Enter key as soon as it happens, and not after the control receives the notification for the KeyDown event. The control will still catch the KeyUp event and it will beep because it's a single-line TextBox control (an audible warning that the specific key shouldn't be used in a single-line TextBox control). To avoid the beep sound, the code ''kills'' the keystroke by setting the SuppressKeystroke property to True.

Processing Keys from within Your Code

The code shown in the preceding KeyDown event handler will work, but you must repeat it for every TextBox control on the form. A more convenient approach is to capture the Enter keystroke in the form's KeyDown event handler and process it for all TextBox controls. First, we must figure out whether the control with the focus is a TextBox control. The property Me.ActiveControl returns a reference to the control with the focus. To find out the type of the active control and compare it to the TextBox control's type, use the following If statement:

```
If Me.ActiveControl.GetType Is GetType(TextBox) Then
' process the Enter key
End If
```

An interesting method of the Form object is the ProcessTabKey method, which imitates the Tab keystroke. Calling the ProcessTabKey method is equivalent to pressing the Tab key from within your code. The method accepts a True/False value as an argument, which indicates whether it will move the focus to the next control in the tab order (if True), or to the previous control in the tab order. Once you can figure out the active control's type and you have a method of simulating the Tab keystroke from within your code, you don't have to code every TextBox control's KeyDown event.

Start by setting the form's KeyPreview property to True and then insert the following statements in the form's KeyDown event handler:

```
If e.KeyCode = Keys.Enter Then
    If Me.ActiveControl.GetType Is GetType(TextBox) Then
        e.SuppressKeyPress = True
        If e.Shift Then
            Me.ProcessTabKey(False)
        Else
            Me.ProcessTabKey(True)
        End If
    End If
End If
```

The last topic demonstrated in this example is how to capture certain keystrokes, regardless of the control that has the focus. We'll use the F10 keystroke to display the total number of contacts entered so far. Assuming that you have already set the form's `KeyPreview` property to True, enter the following code in the form's `KeyDown` event:

```
If e.Keycode = keys.F10 Then
   MsgBox("There are " & Contacts.Count.ToString & _
          " contacts in the database")
   e.Handled = True
End If
```

Listing 7.1 shows the complete handler for the form's `KeyDown` event, which also allows you to move to the next or previous contact by using the Alt+Plus or Alt+Minus keys, respectively.

Listing 7.1: Handling Keystrokes in the Form's *KeyDown* Event Handler

```
Public Sub Form1_KeyDown(ByVal sender As Object, _
           ByVal e As System.WinForms.KeyEventArgs) _
                        Handles Form1.KeyUp
   If e.Keycode = Keys.F10 Then
      MsgBox("There are " & Contacts.Count.ToString & _
             " contacts in the database")
      e.Handled = True
   End If
   If e.KeyCode = Keys.Subtract And e.Modifiers = Keys.Alt Then
      bttnPrevious.PerformClick
   End If
   If e.KeyCode = Keys.Add And e.Modifiers = Keys.Alt Then
      bttnNext.PerformClick
   End If
End Sub
```

The `KeyCode` property of the `e` argument returns the code of the key that was pressed. All key codes are members of the `Keys` enumeration, so you need not memorize them. The name of the key with the plus symbol is `Keys.Add`. The `Modifiers` property of the same argument returns the modifier key(s) that were held down while the key was pressed. Also, all possible values of the `Modifiers` property are members of the `Keys` enumeration and will appear as soon as you type the equal sign.

Anchoring and Docking

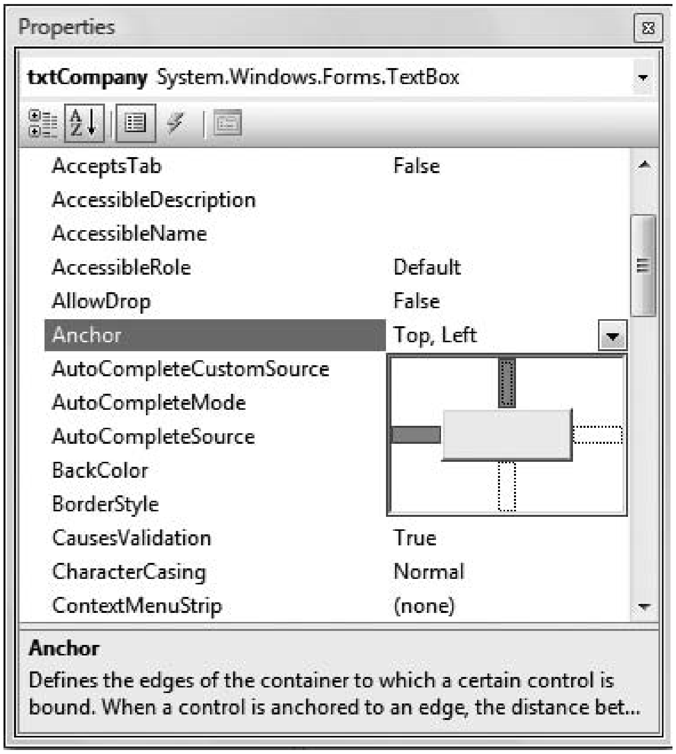
A common issue in form design is the design of forms that are properly resized. For instance, you might design a nice form for a given size, but when it's resized at runtime, the controls are all clustered in the top-left corner. Or a TextBox control that covers the entire width of the form at design time suddenly ''cringes'' on the left when the user drags out the window. If the user makes the form smaller than the default size, part of the TextBox could be invisible because it's outside the form. You can attach scroll bars to the form, but that doesn't really help — who wants to type text and have to scroll the form horizontally? It makes sense to scroll vertically because you get to see many lines at once, but if the TextBox control is wider than the form, you can't read entire lines.

Anchoring Controls

The `Anchor` property lets you attach one or more edges of the control to corresponding edges of the form. The anchored edges of the control maintain the same distance from the corresponding edges of the form.

Place a TextBox control on a new form, set its `MultiLine` property to True, and then open the control's `Anchor` property in the Properties window. You will see a rectangle within a larger rectangle and four pegs that connect the small control to the sides of the larger box (see Figure 7.6). The large box is the form, and the small one is the control. The four pegs are the anchors, which can be either white or gray. The gray anchors denote a fixed distance between the control and the form. By default, the control is placed at a fixed distance from the top-left corner of the form. When the form is resized, the control retains its size and its distance from the top-left corner of the form.

Figure 7.6
The settings of the `Anchor` property



Properties

txtCompany System.Windows.Forms.TextBox

AcceptsTab	False
AccessibleDescription	
AccessibleName	
AccessibleRole	Default
AllowDrop	False
Anchor	Top, Left
AutoCompleteCustomSource	
AutoCompleteMode	
AutoCompleteSource	
BackColor	
BorderStyle	
CausesValidation	True
CharacterCasing	Normal
ContextMenuStrip	(none)

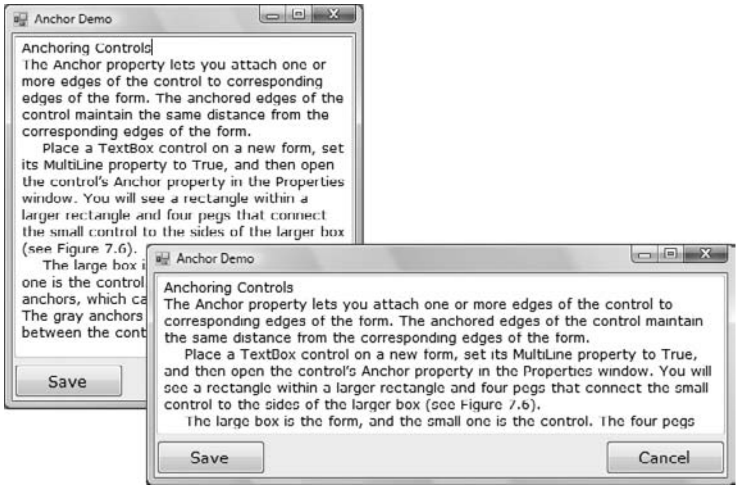
Anchor
Defines the edges of the container to which a certain control is bound. When a control is anchored to an edge, the distance bet...

We want our TextBox control to fill the width of the form, be aligned to the top of the form, and leave some space for a few buttons at the bottom. We also want our form to maintain this arrangement, regardless of its size. Make the TextBox control as wide as the form (allowing, perhaps, a margin of a few pixels on either side). Then place a couple of buttons at the bottom of the form and make the TextBox control tall enough that it stops above the buttons. This is the form of the Anchor sample project.

Now open the TextBox control's `Anchor` property and make all four anchors gray by clicking them. This action tells the Form Designer to resize the control accordingly at runtime, so that the distances between the sides of the control and the corresponding sides of the form are the same as those you set at design time. Select each button on the form and set their `Anchor` properties in the Properties window: Anchor the left button to the left and bottom of the form, and the right button to the right and bottom of the form.

Resize the form at design time without running the project, and you'll see that all the controls are resized and rearranged on the form at all times. Figure 7.7 shows the Anchor project's main form in two different sizes.

FIGURE 7.7
Use the `Anchor` property of the various controls to design forms that can be resized gracefully at runtime.



Anchoring Controls
The Anchor property lets you attach one or
more edges of the control to corresponding
edges of the form. The anchored edges of the
control maintain the same distance from the
corresponding edges of the form.
Place a TextBox control on a new form, set
its MultiLine property to True, and then open
the control's Anchor property in the Properties
window. You will see a rectangle within a
larger rectangle and four pegs that connect
the small control to the sides of the larger box
(see Figure 7.6).
The large box i
one is the control
anchors, which ca
The gray anchors
between the cont

Anchoring Controls
The Anchor property lets you attach one or more edges of the control to
corresponding edges of the form. The anchored edges of the control maintain
the same distance from the corresponding edges of the form.
Place a TextBox control on a new form, set its MultiLine property to True,
and then open the control's Anchor property in the Properties window. You will
see a rectangle within a larger rectangle and four pegs that connect the small
control to the sides of the larger box (see Figure 7.6).
The large box is the form, and the small one is the control. The four pegs

Yet, there's a small problem: If you make the form very narrow, there will be no room for both buttons across the form's width. The simplest way to fix this problem is to impose a minimum size for the form. To do so, you must first decide the form's minimum width and height and then set the `MinimumSize` property to these values. You can also use the `AutoScroll` properties, but it's not recommended that you add scroll bars to a small form like ours.

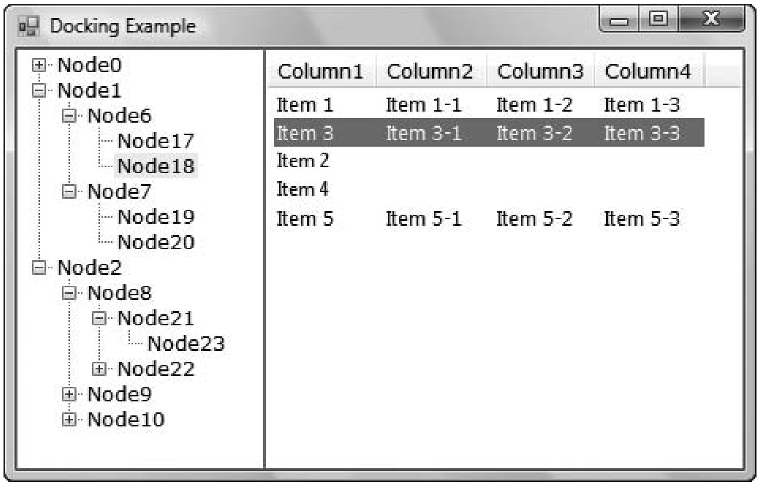
DOCKING CONTROLS

In addition to the `Anchor` property, most controls provide the `Dock` property, which determines how a control will dock on the form. The default value of this property is None.

Create a new form, place a multiline TextBox control on it, and then open the control's `Dock` property. The various rectangular shapes are the settings of the property. If you click the middle rectangle, the control will be docked over the entire form: It will expand and shrink both horizontally and vertically to cover the entire form. This setting is appropriate for simple forms that contain a single control, usually a TextBox, and sometimes a menu. Try it out.

Let's create a more complicated form with two controls (see the Docking sample project). The form shown in Figure 7.8 contains a TreeView control on the left and a ListView control on the right. The two controls display folder and file data on an interface that's very similar to that of Windows Explorer. The TreeView control displays the directory structure, and the ListView control displays the selected folder's files.

Place a TreeView control on the left side of the form and a ListView control on the right side of the form. Then dock the TreeView to the left and the ListView to the right. If you run the application now, as you resize the form, the two controls remain docked to the two sides of the form — but their sizes don't change. If you make the form wider, there will be a gap between the two controls. If you make the form narrower, one of the controls will overlap the other.



Column1	Column2	Column3	Column4
Item 1	Item 1-1	Item 1-2	Item 1-3
Item 3	Item 3-1	Item 3-2	Item 3-3
Item 2			
Item 4			
Item 5	Item 5-1	Item 5-2	Item 5-3

FIGURE 7.8
Filling a form with two controls

End the application, return to the Form Designer, select the ListView control, and set its `Dock` property to Fill. This time, the ListView will change size to take up all the space to the right of the TreeView. The ListView control will attempt to fill the form, but it won't take up the space of another control that has been docked already. The TreeView and ListView controls are discussed in Chapter 9, ''The TreeView and ListView Controls''; that's why I've populated them with some fake data at design time. In Chapter 9, you'll learn how to populate these two controls at runtime with folder names and filenames, respectively, and build a custom Windows Explorer.

Splitting Forms into Multiple Panes

The form behaves better, but it's not what you really expect from a Windows application. The problem with the form in Figure 7.8 is that users can't change the relative widths of the controls. In other words, they can't make one of the controls narrower to make room for the other, which is a fairly common concept in the Windows interface.

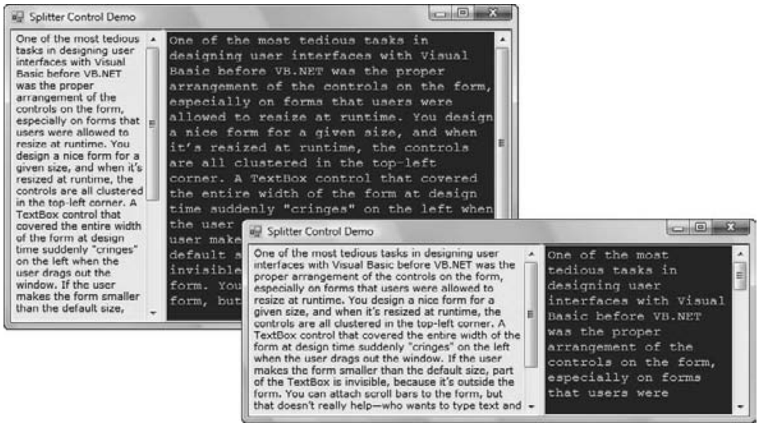
The narrow bar that allows users to control the relative sizes of two controls is a *splitter*. When the cursor hovers over a splitter, it changes to a double arrow to indicate that the bar can be moved. By moving the splitter, you can enlarge one of the two controls while shrinking the other. The Form Designer provides a special control for placing a splitter between two controls: the SplitContainer control. We'll design a new form with two TextBoxes and a splitter between them so that users can change the relative size of the two controls.

First, place a SplitContainer control on the form. The SplitContainer consists of two Panels, the *`Panel1`* and *`Panel2`* controls, and a vertical splitter between them. This is the default configuration; you can change the orientation of the splitter by using the control's `Orientation` property. Also by default, the two panels of the Splitter control are resized proportionally as you resize the form. If you want to keep one of the panels fixed and have the other take up the rest of the form, set the control's `FixedPanel` property to the name of the panel you want to retain its size.

Next, place a TextBox control in the left panel of the SplitControl and set its `Multiline` property to True. You don't need to do anything about its size because we'll dock it in the panel to which it belongs. With the TextBox control selected, locate its `Dock` property and set it to *`Fill`*. The TextBox control will fill the left panel of the SplitContainer control. Do the same with another TextBox control, which will fill the right panel of the SplitContainer control. Set this control's `Multiline` property to True and its `Dock` property to *`Fill`*.

Now run the project and check out the functionality of the SplitContainer. Paste some text on the two controls and then change their relative sizes by sliding the splitter between them, as shown in Figure 7.9. You will find this project, called Splitter1, among the sample projects of this chapter.

FIGURE 7.9
The SplitContainer control lets you change the relative size of the controls on either side.



One of the most tedious tasks in designing user interfaces with Visual Basic before VB.NET was the proper arrangement of the controls on the form, especially on forms that users were allowed to resize at runtime. You design a nice form for a given size, and when it's resized at runtime, the controls are all clustered in the top-left corner. A TextBox control that covered the entire width of the form at design time suddenly "cringes" on the left when the user drags out the window. If the user makes the form smaller than the default size,

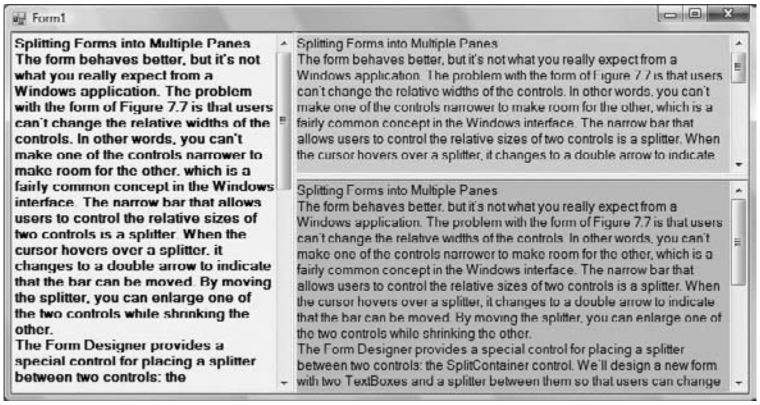
```
One of the most tedious tasks in
designing user interfaces with Visual
Basic before VB.NET was the proper
arrangement of the controls on the form,
especially on forms that users were
allowed to resize at runtime. You design
a nice form for a given size, and when
it's resized at runtime, the controls
are all clustered in the top-left
corner. A TextBox control that covered
the entire width of the form at design
time suddenly "cringes" on the left when
the user
user make
default s
invisible
form. You
form, but
```

One of the most tedious tasks in designing user interfaces with Visual Basic before VB.NET was the proper arrangement of the controls on the form, especially on forms that users were allowed to resize at runtime. You design a nice form for a given size, and when it's resized at runtime, the controls are all clustered in the top-left corner. A TextBox control that covered the entire width of the form at design time suddenly "cringes" on the left when the user drags out the window. If the user makes the form smaller than the default size, part of the TextBox is invisible, because it's outside the form. You can attach scroll bars to the form, but that doesn't really help—who wants to type text and

```
One of the most
tedious tasks in
designing user
interfaces with Visual
Basic before VB.NET
was the proper
arrangement of the
controls on the form,
especially on forms
that users were
```

Let's design a more elaborate form with two SplitContainer controls, such as the one shown in Figure 7.10. (It's the form in the Splitter2 sample project.) This form, which resembles the interface of Microsoft Office Outlook, consists of a TreeView control on the left (where the folders are displayed), a ListView control (where the selected folder's items are displayed), and a TextBox control (where the selected item's details are displayed). Because we haven't discussed the ListView and TreeView controls yet, I'm using three TextBox controls with different background colors; the process of designing the form is identical, regardless of the controls you put on it.

FIGURE 7.10
An elaborate form with two splitter controls



Splitting Forms into Multiple Panes
The form behaves better, but it's not what you really expect from a Windows application. The problem with the form of Figure 7.7 is that users can't change the relative widths of the controls. In other words, you can't make one of the controls narrower to make room for the other, which is a fairly common concept in the Windows interface. The narrow bar that allows users to control the relative sizes of two controls is a splitter. When the cursor hovers over a splitter, it changes to a double arrow to indicate that the bar can be moved. By moving the splitter, you can enlarge one of the two controls while shrinking the other.
The Form Designer provides a special control for placing a splitter between two controls: the

Splitting Forms into Multiple Panes
The form behaves better, but it's not what you really expect from a Windows application. The problem with the form of Figure 7.7 is that users can't change the relative widths of the controls. In other words, you can't make one of the controls narrower to make room for the other, which is a fairly common concept in the Windows interface. The narrow bar that allows users to control the relative sizes of two controls is a splitter. When the cursor hovers over a splitter, it changes to a double arrow to indicate

Splitting Forms into Multiple Panes
The form behaves better, but it's not what you really expect from a Windows application. The problem with the form of Figure 7.7 is that users can't change the relative widths of the controls. In other words, you can't make one of the controls narrower to make room for the other, which is a fairly common concept in the Windows interface. The narrow bar that allows users to control the relative sizes of two controls is a splitter. When the cursor hovers over a splitter, it changes to a double arrow to indicate that the bar can be moved. By moving the splitter, you can enlarge one of the two controls while shrinking the other.
The Form Designer provides a special control for placing a splitter between two controls: the SplitContainer control. We'll design a new form with two TextBoxes and a splitter between them so that users can change

Start by placing a SplitContainer control on the form. Then place a multiline TextBox control on the left panel of the SplitContainer control and set the TextBox control's `Dock` property to *`Fill`*. The TextBox control will fill the left panel of the SplitContainer control. Place another SplitContainer in the right panel of the first SplitContainer control. This control will be automatically docked in its panel and will fill it. Its orientation, however, is vertical, and the splitter will separate the panel into two smaller vertical panes. Select the second SplitContainer control, locate its `Orientation` property in the Properties window, and set it to *`Horizontal`*.

Now you can fill each of the panels with a TextBox control. Set each TextBox control's `BackgroundColor` to a different color, its `MultiLine` property to True, and its `Dock` property to *`Fill`*. The TextBox controls will fill their containers, which are the panels of the two SplitContainer controls, not the form. If you look up the properties of a SplitContainer control, you'll see that it's made up of two Panel controls, which are exposed as properties of the SplitContainer control, the `Panel1` and `Panel2` controls. You can set many of the properties of these two constituent controls, such as their font and color, their minimum size, and so on. They even expose an `AutoScroll` property, so that users can scroll the contents of each one independently of the other. You can also set other properties of the SplitContainer control, such as the `SplitterWidth` property, which is the width of the splitter bar between the two panels in pixels, and the `SplitterIncrement` property, which is the smallest number of pixels that the splitter bar can be moved in either direction.

So far, you've seen what the Form Designer and the Form object can do for your application. Let's switch our focus to programming forms and explore the events triggered by the Form object.

The Form's Events

The Form object triggers several events. The most important are `Activated`, `Deactivate`, `FormClosing`, `Resize`, and `Paint`.

The *Activated* and *Deactivate* Events

When more than one form is displayed, the user can switch from one to the other by using the mouse or by pressing Alt+Tab. Each time a form is activated, the `Activated` event takes place. Likewise, when a form is activated, the previously active form receives the `Deactivate` event. Insert in these two event handlers the code you want to execute when a form is activated (set certain control properties, for example) and when a form loses the focus or is deactivated. These two events are the form's equivalents of the `Enter` and `Leave` events of the various controls. Notice an inconsistency in the names of the two events: the `Activated` event takes place after the form has been activated, whereas the `Deactivate` event takes place right before the form is deactivated.

The *FormClosing* and *FormClosed* Events

The `FormClosing` event is fired when the user closes the form by clicking its Close button. If the application must terminate because Windows is shutting down, the same event will be fired as well. Users don't always quit applications in an orderly manner, and a professional application should behave gracefully under all circumstances. The same code you execute in the application's Exit command must also be executed from within the closing event. For example, you might display a warning if the user has unsaved data, you might have to update a database, and so on. Place the code that performs these tasks in a subroutine and call it from within your menu's Exit command, as well as from within the `FormClosing` event's handler.

You can cancel the closing of a form by setting the `e.Cancel` property to True. The event handler in Listing 7.2 displays a message box informing the user that the data hasn't been saved and gives him a chance to cancel the action and return to the application.

Listing 7.2: Cancelling the Closing of a Form

```
Public Sub Form1_FormClosing(...) Handles Me.FormClosing
   Dim reply As MsgBoxResult
   reply = MsgBox("Document has been edited. " & _
            "OK to terminate application, Cancel to " &   _
            "return to your document.", MsgBoxStyle.OKCancel)
   If reply = MsgBoxResult.Cancel Then
      e.Cancel = True
   End If
End Sub
```

The `e` argument of the `FormClosing` event provides the `CloseReason` property, which reports how the form is closing. Its value is one of the following members of the `CloseReason` enumeration: *`FormOwnerClosing`*, *`MdiFormClosing`*, *`None`*, *`TaskManagerClosing`*, *`WindowsShutDown`*. The names of the members are self-descriptive, and you can query the `CloseReason` property to determine how the window is closing.

The `FormClosed` event fires after the form has been closed. You can find out the action that caused the form to be closed through the `e.CloseReason` property, but it's too late to cancel the closing of the form.

The *Resize*, *ResizeBegin*, and *ResizeEnd* Events

The `Resize` event is fired every time the user resizes the form by using the mouse. With previous versions of VB, programmers had to insert quite a bit of code in the `Resize` event's handler to resize the controls and possibly rearrange them on the form. With the `Anchor` and `Dock` properties, much of this overhead can be passed to the form itself. If you want the two sides of the form to maintain a fixed ratio, however, you have to resize one of the dimensions from within the `Resize` event handler. Let's say the form's width-to-height ratio must be 3:4. Assuming that you're using the form's height as a guide, insert the following statement in the `Resize` event handler to make the width equal to three-fourths of the height:

```
Private Form1_Resize(...) Handles Me.Resize
   Me.Width = (0.75 * Me.Height)
End Sub
```

The `Resize` event is fired continuously while the form is being resized. If you want to keep track of the initial form's size and perform all the calculations after the user has finished resizing the form, you can use the `ResizeBegin` and `ResizeEnd` events, which are fired at the beginning and after the end of a resize operation, respectively. Store the form's width and height to two global variables in the `ResizeBegin` event and use these two variables in the `ResizeEnd` event handler.

The *Scroll* Event

The `Scroll` event is fired by forms that have their `AutoScroll` property set to True when the user scrolls the form. The second argument of the `Scroll` event handler exposes the `OldValue` and `NewValue` properties, which are the displacements of the form before and after the scroll operation. This event can be used to keep a specific control in view when the form's contents are scrolled.

The AutoScroll property is handy for large forms, but it has a serious drawback: It scrolls the entire form. In most cases, we want to keep certain controls in view at all times. Instead of a scrollable form, you can create forms with scrollable sections by exploiting the AutoScroll properties of the Panel and/or the SplitContainer controls. You can also reposition certain controls from within the form's Scroll event handler. Let's say you have placed a few controls on a Panel container and you want to keep this Panel at the top of a scrolling form. The following statements in the form's Scroll event handler reposition the Panel at the top of the form every time the user scrolls the form:

```
Private Sub Form1_Scroll(...) Handles Me.Scroll
    Panel1.Top = Panel1.Top + (e.NewValue - e.OldValue)
End Sub
```

THE *PAINT* EVENT

This event takes place every time the form must be refreshed, and we use its handler to execute code for any custom drawing on the form. When you switch to another form that partially or totally overlaps the current one and then switch back to the first form, the Paint event will be fired to notify your application that it must redraw the form. The form will refresh its controls automatically, but any custom drawing on the form won't be refreshed automatically. We'll discuss this event in more detail in Chapter 18, ''Drawing and Painting with Visual Basic 2008,'' in the presentation of the Framework's drawing methods.

Loading and Showing Forms

Most practical applications are made up of multiple forms and dialog boxes, and one of the operations you'll have to perform with multiform applications is to load and manipulate forms from within other forms' code. For example, you might want to display a second form to prompt the user for data specific to an application. You must explicitly load the second form and read the information entered by the user when the auxiliary form is closed. Or you might want to maintain two forms open at once and let the user switch between them. A text editor and its Find & Replace dialog box is a typical example.

You can access a form from within another form by its name. Let's say that your application has two forms, named Form1 and Form2, and that Form1 is the project's startup form. To show Form2 when an action takes place on Form1, call the Show method of the auxiliary form:

```
Form2.Show
```

This statement brings up Form2 and usually appears in a button's or menu item's Click event handler. To exchange information between two forms, use the techniques described in the ''Controlling One Form from within Another,'' section later in this chapter.

The Show method opens a form in a *modeless* manner: The two forms are equal in stature on the desktop, and the user can switch between them. You can also display the second form in a *modal* manner, which means that users can't return to the form from which they invoked it without closing the second form. While a modal form is open, it remains on top of the desktop, and you can't move the focus to any other form of the same application (but you can switch to another application). To open a modal form, use the ShowDialog method:

```
Form2.ShowDialog
```

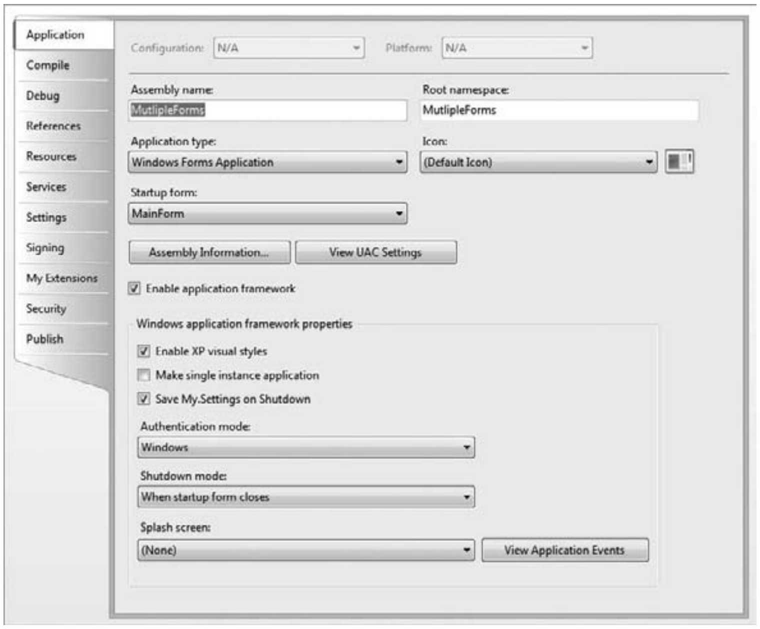
The modal form is, in effect, a dialog box like the Open File dialog box. You must first select a file on this form and click the Open button, or click the Cancel button to close the dialog box and return to the form from which the dialog box was invoked. This brings up the topic of forms and dialog boxes.

A dialog box is simply a modal form. When we display forms as dialog boxes, we change the border of the forms to the setting `FixedDialog` and invoke them with the `ShowDialog` method. Modeless forms are more difficult to program because the user may switch among them at any time. Moreover, the two forms that are open at once must interact with one another. When the user acts on one of the forms, it might necessitate some changes in the other, and you'll see shortly how this is done. If the two active forms don't need to interact, display one of them as a dialog box.

When you're finished with the second form, you can either close it by calling its `Close` method or hide it by calling its `Hide` method. The `Close` method closes the form, and its resources are returned to the system. The `Hide` method sets the form's `Visible` property to False; you can still access a hidden form's controls from within your code, but the user can't interact with it. Forms that are displayed often, such as the Find & Replace dialog box of a text-processing application, should be hidden — not closed. To the user, it makes no difference whether you hide or close a form. If you hide a form, however, the next time you bring it up with the `Show` method, its controls are in the state they were the last time.

The Startup Form

A typical application has more than a single form. When an application starts, the main form is loaded. You can control which form is initially loaded by setting the startup object in the project Properties window, shown in Figure 7.11. To open this dialog box, right-click the project's name in the Solution Explorer and select Properties. In the project's Properties pages, select the Application tab and select the appropriate item in the Startup Form combo box.



Configuration: N/A Platform: N/A

Assembly name: MutlipleForms

Root namespace: MutlipleForms

Application type: Windows Forms Application

Icon: (Default Icon)

Startup form: MainForm

☑ 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: (None)

FIGURE 7.11
In the Properties window, you can specify the form that's displayed when the application starts.

By default, the IDE suggests the name of the first form it created, which is `Form1`. If you change the name of the form, Visual Basic will continue using the same form as the startup form with its new name.

You can also start an application by using a subroutine, without loading a form. This subroutine is the `MyApplication_Startup` event handler, which is fired automatically when the application starts. To display the AuxiliaryForm object from within the `Startup` event handler, use the following statement:

```
Private Sub MyApplication_Startup(...) Handles Me.Startup
      System.Windows.Forms.Application.Run(New AuxiliaryForm())
End Sub
```

To view the `MyApplication_Startup` event handler, click the View Application Events button at the bottom of the Application pane, as shown in Figure 7.11. This action will take you to the MyApplication code window, where you can select the MyApplication Events item in the object list and the Startup item in the events list.

Controlling One Form from within Another

Loading and displaying a form from within another form's code is fairly trivial. In some situations, this is all the interaction you need between forms. Each form is designed to operate independently of the others, but they can communicate via public variables (see the following section). In most situations, however, you need to control one form from within another's code. Controlling the form means accessing its controls, and setting or reading values from within another form's code.

In Chapter 6, you developed the TextPad application, which is a basic text editor and consists of the main form and an auxiliary form for the Find & Replace operations. All other operations on the text are performed with menu commands on the main form. When the user wants to search for and/or replace a string, the program displays another form on which the user specifies the text to find, the type of search, and so on. When the user clicks one of the Find & Replace form's buttons, the corresponding code must access the text on the main form of the application and search for a word or replace a string with another. The Find & Replace dialog box not only interacts with the TextBox control on the main form, it also remains visible at all times while it's open, even if it doesn't have the focus, because its `TopMost` property was set to True.

Sharing Variables between Forms

The preferred method for two forms to communicate with each other is via *public variables*. These variables are declared in the form's declarations section, outside any procedure, with the keyword `Public`. If the following declarations appear in `Form1`, the variable *`NumPoints`* and the array *`DataValues`* can be accessed by any procedure in `Form1`, as well as from within the code of any form belonging to the same project:

```
Public NumPoints As Integer
Public DataValues(100) As Double
```

To access a public variable declared in `Form1` from within another form's code, you must prefix the variable's name by the name of the form, as in the following:

```
Form1.NumPoints = 99
Form1.DataValues(0) = 0.3395022
```

You can use the same notation to access the controls on another form. If `Form1` contains the `TextBox1` control, you can use the following statement to read its text:

```
Form1.TextBox1.Text
```

If a button on `Form1` opens the auxiliary form `Form2`, you can set selected controls to specific values before showing the auxiliary form. The following statements should appear in a button's or menu item's `Click` event handler:

```
Form2.TextBox1.Text = "some text"
Form2.DateTimePicker1.Value = Today
Form2.Show()
```

You can also create a variable to represent another form and access the auxiliary form through this variable. Let's say you want to access the resources of `Form2` from within the code of `Form1`. Declare the variable `auxForm` to represent `Form2` and then access the resources of `Form2` with the following statements:

```
Dim auxForm As Form2
auxForm.TextBox1.Text = "some text"
auxForm.DateTimePicker1.Value = Today
auxForm.Show
```

Multiple Instances of a Single Form

Note that the variable that represents an auxiliary form is declared without the New keyword. The *auxForm* variable represents an existing form. If we used the New keyword, we'd create a new instance of the corresponding form. This technique is used when we want to display multiple instances of the same form, as in an application that allows users to open multiple documents of the same type.

Let's say you're designing an image-processing application, or a simple text editor. Each new document should be opened in a separate window. Obviously, we can't design many identical forms and use them as needed. The solution is to design a single form and create new instances of it every time the user opens an existing document or creates a new one. These instances are independent of one another and they may interact with the main form. Usually they don't, because they aren't auxiliary forms; they contain the necessary interface elements, such as menus, for processing the specific document type, and users can arrange them any way they like on the desktop.

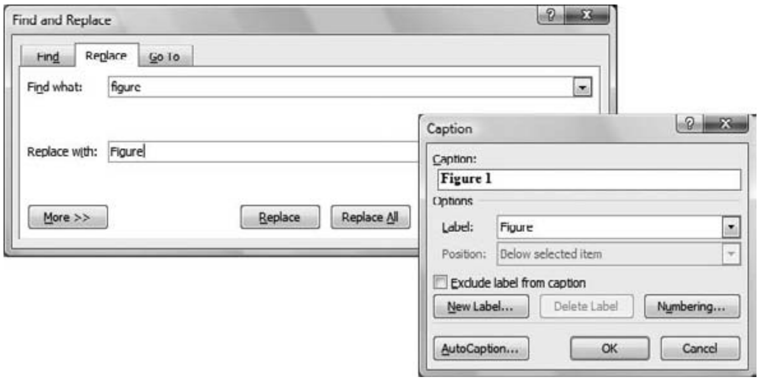
The approach described here is reminiscent of Multiple Document Interface (MDI) applications. The MDI interface requires that all windows be contained within a parent window and, although once very popular, it's going slowly out of style. The new interfaces open multiple independent windows on the desktop. Each window is an instance of a single form and it's declared with the New keyword. I've used this style of interface to redesign the TextPad application of Chapter 6, and I've included the revised application in this chapter's projects for your reference. Open the project in Visual Studio and examine its code, which contains a lot of comments.

Forms versus Dialog Boxes

Dialog boxes are special types of forms with very specific functionality, which we use to prompt the user for data. The Open and Save dialog boxes are two of the most familiar dialog boxes in Windows. They're so common that they're actually known as *common dialog boxes*. Technically, a dialog box is a good old form with its `FormBorderStyle` property set to FixedDialog. Like forms, dialog boxes might contain a few simple controls, such as Labels, TextBoxes, and Buttons. You can't overload a dialog box with controls and functionality, because you'll end up with a regular form.

Figure 7.12 shows a few dialog boxes you have certainly seen while working with Windows applications. The Insert Caption dialog box of Word is a modal dialog box: You must close it before switching to your document. The Find & Replace dialog box is modeless: It allows you to switch to your document, yet it remains visible while open even if it doesn't have the focus.

FIGURE 7.12
Typical dialog boxes used by Word



Find what: figure
Replace with: Figure
Caption: Figure 1
Label: Figure
Position: Below selected item

Notice that some dialog boxes, such as Open, Color, and even the humble MessageBox, come with the Framework, and you can incorporate them in your applications without having to design them.

A characteristic of dialog boxes is that they provide an OK and a Cancel button. The OK button tells the application that you're finished using the dialog box, and the application can process the information in it. The Cancel button tells the application that it should ignore the information in the dialog box and cancel the current operation. As you will see, dialog boxes allow you to quickly find out which buttons were clicked to close them, so that your application can take a different action in each case.

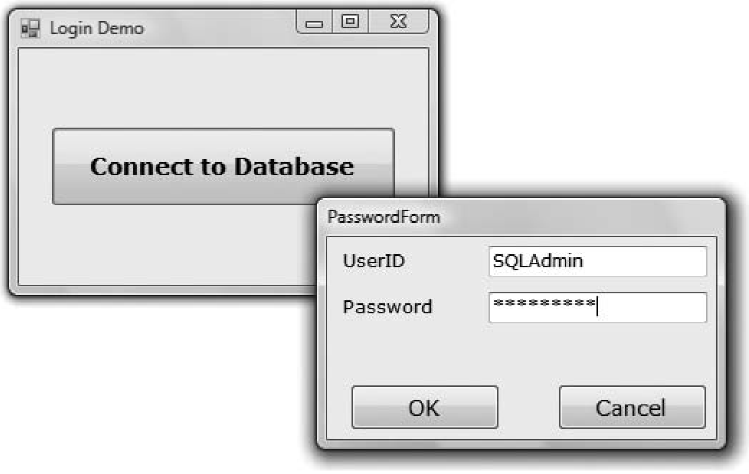
In short, the difference between forms and dialog boxes is artificial. If it were really important to distinguish between the two, they'd be implemented as two different objects — but they're the same object. So, without any further introduction, let's look at how to create and use dialog boxes.

To create a dialog box, start with a Windows form, set its `FormBorderStyle` property to Fixed-Dialog, and set the `ControlBox`, `MinimizeBox`, and `MaximizeBox` properties to False. Then add the necessary controls on the form and code the appropriate events, as you would do with a regular Windows form.

Figure 7.13 shows a simple dialog box that prompts the user for an ID and a password (see the Password sample project). The dialog box contains two TextBox controls, next to the appropriate labels, and the usual OK and Cancel buttons. The Cancel button signifies that the user wants to cancel the operation, which was initiated in the form that displayed the dialog box.

Start a new project, rename the form to **`MainForm`**, and place a button on the form. This is the application's main form, and we'll invoke the dialog box from within the button's `Click` event handler. Then add a new form to the project, name it **`PasswordForm`**, and place on it the controls shown in Figure 7.13.

FIGURE 7.13
A simple dialog box that prompts users for a username and password



To display a modal form, you call the `ShowDialog` method, instead of the `Show` method. You already know how to read the values entered on the controls of the dialog box. You also need to know which button was clicked to close the dialog box. To convey this information from the dialog box back to the calling application, the Form object provides the `DialogResult` property. This property can be set to one of the values shown in Table 7.3 to indicate which button was clicked. The *`DialogResult.OK`* value indicates that the user has clicked the OK button on the form. There's no need to place an OK button on the form; just set the form's `DialogResult` property to DialogResult.OK.

TABLE 7.3: The *DialogResult* Enumeration

VALUE	**DESCRIPTION**
Abort	The dialog box was closed with the Abort button.
Cancel	The dialog box was closed with the Cancel button.
Ignore	The dialog box was closed with the Ignore button.
No	The dialog box was closed with the No button.
None	The dialog box hasn't been closed yet. Use this option to find out whether a modeless dialog box is still open.
OK	The dialog box was closed with the OK button.
Retry	The dialog box was closed with the Retry button.
Yes	The dialog box was closed with the Yes button.

The dialog box need not contain any of the buttons mentioned here. It's your responsibility to set the value of the `DialogResult` property from within your code to one of the settings shown in the table. This value can be retrieved by the calling application. The code behind the two buttons in the dialog box is quite short:

```
Private Sub bttnOK_Click(...) Handles bttnOK.Click
    Me.DialogResult = DialogResult.OK
    Me.Close
End Sub

Private Sub bttnCancel_Click(...) Handles bttnCancel.Click
    Me.DialogResult = DialogResult.Cancel
    Me.Close
End Sub
```

The event handler of the button that displays this dialog box should contain an `If` statement that examines the value returned by the `ShowDialog` method:

```
If PasswordForm.ShowDialog = DialogResult.OK Then
   { process the user selection }
End If
```

Depending on your application, you might allow the user to close the dialog box by clicking more than two buttons. Some of them must set the `DialogResult` property to *`DialogResult.OK`*, others to *`DialogResult.Cancel`*.

If the form contains an Accept and a Cancel button, you don't have to enter a single line of code in the modal form. The user can enter values on the various controls and then close the dialog box by pressing the Enter or Cancel key. The dialog box will close and will return the *`Dialo-gResult.OK`* or *`DialogResult.Cancel`* value. The Accept button sets the form's `DialogResult` property to *`DialogResult.OK`* automatically, and the Cancel button sets the same property to *`DialogResult.Cancel`*. Any other button must set the `DialogResult` property explicitly. Listing 7.3 shows the code behind the Log In button on the sample project's main form.

LISTING 7.3: Prompting the User for an ID and a Password

```
Private Sub Button1_Click(...) Handles Button1.Click
   If PasswordForm.ShowDialog() = DialogResult.OK Then
      If PasswordForm.txtUserID.Text = "" Or _
              PasswordForm.txtPassword.Text = "" Then
         MsgBox("Please specify a user ID and a password to connect")
         Exit Sub
      End If
      MsgBox("You were connected as " & _
              passwordForm.txtUserID.Text)
   Else
```

```
            MsgBox("Connection failed for user " & _
                    password.txtPassword.Text)
        End If
    End Sub
```

VB 2008 at Work: The MultipleForms Project

It's time to write an application that puts together the most important topics discussed in this section. The MultipleForms project consists of a main form, an auxiliary form, and a dialog box. All three components of the application's interface are shown in Figure 7.14. The buttons on the main form display both the auxiliary form and the dialog box.

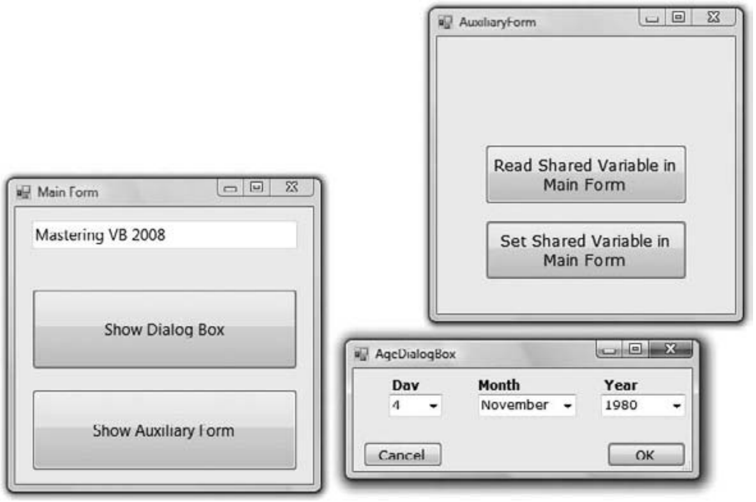


Figure 7.14
The MultipleForms project's interface

Let's review the various operations we want to perform — they're typical for many situations, not for only this application. At first, we must be able to invoke both the auxiliary form and the dialog box from within the main form; the Show Auxiliary Form and Show Dialog Box buttons do this. The main form contains a variable declaration: *strProperty*. This variable is, in effect, a property of the main form and is declared as public with the following statement:

```
Public strProperty As String = "Mastering VB 2008"
```

The main form calls the auxiliary form's Show method to display it in a modeless manner. The auxiliary form button named Read Shared Variable In Main Form reads the *strProperty* variable of the main form with the following statement:

```
Private Sub bttnReadShared_Click(...) Handles bttnReadShared.Click
   MsgBox(MainForm.strProperty, MsgBoxStyle.OKOnly, _
               "Public Variable Value")
End Sub
```

Using the same notation, you can set this variable from within the auxiliary form. The following event handler prompts the user for a new value and assigns it to the shared variable of the main form:

```
Private Sub bttnSetShared_Click(...) Handles bttnSetShared.Click
   Dim str As String
   str = InputBox("Enter a new value for strProperty")
   MainForm.strProperty = str
End Sub
```

The two forms communicate with each other through public variables. Let's make this communication a little more elaborate by adding an event. Every time the auxiliary form sets the value of the *strProperty* variable, it will raise an event to notify the main form. The main form, in turn, will use this event to display the new value of the string on the TextBox control as soon as the code in the auxiliary form changes the value of the variable and before it's closed.

To raise an event, you must declare the event's name in the form's declaration section. Insert the following statement in the auxiliary form's declarations section:

```
Event strPropertyChanged()
```

Now add a statement that fires the event. To raise an event, we call the `RaiseEvent` statement, passing the name of the event as an argument. This statement must appear in the `Click` event handler of the Set Shared Variable In Main Form button, right after setting the value of the shared variable. Listing 7.4 shows the revised event handler.

LISTING 7.4: Raising an Event

```
Private Sub bttnSetShared_Click(...) Handles bttnSetShared.Click
   Dim str As String
   str = InputBox("Enter a new value for strProperty")
   MainForm.strProperty = str
   RaiseEvent strPropertyChanged
End Sub
```

The event will be raised, but it will go unnoticed if we don't handle it from within the main form's code. To handle the event, you must create a variable that represents the auxiliary form with the `WithEvents` keyword:

```
Dim WithEvents FRM As New AuxiliaryForm()
```

The `WithEvents` keyword tells VB that the variable is capable of raising events. If you expand the drop-down list with the objects in the code editor, you will see the name of the `FRM` variable, along with the other controls you can program. Select `FRM` in the list and then expand the list of events for the selected item. In this list, you will see the `strPropertyChanged` event.

Select it, and the definition of an event handler will appear. Enter these statements in this event's handler:

```
Private Sub FRM_strPropertyChanged() Handles FRM.strPropertyChanged
   TextBox1.Text = strProperty
   Beep()
End Sub
```

It's a simple handler, but it's adequate for demonstrating how to raise and handle custom events on the form level. If you want, you can pass arguments to the event handler by including them in the declaration of the event. To pass the original and the new value through the `strPropertyChanged` event, use the following declaration:

```
Event strPropertyChanged(ByVal oldValue As String, _
                         ByVal newValue As String)
```

If you run the application now, you'll see that the value of the TextBox control in the main form changes as soon as you change the property's value in the auxiliary form.

Of course, you can update the TextBox control on the main form directly from within the auxiliary form's code. Use the expression *`MainForm.TextBox1`* to access the control and then manipulate it as usual. Events are used to perform some actions on a form when an action takes place in one of the other forms of the application. The benefit of using events, as opposed to accessing members of another form from within our code, is that the auxiliary form need not know anything about the form that called it. The auxiliary form raises the event, and it's the other form's responsibility to handle it.

Let's see now how the main form interacts with the dialog box. What goes on between a form and a dialog box is not exactly *interaction*; it's a more timid type of behavior. The form displays the dialog box and waits until the user closes the dialog box. Then it looks at the value of the `DialogResult` property to find out whether it should even examine the values passed back by the dialog box. If the user has closed the dialog box with the Cancel (or an equivalent) button, the application ignores the dialog box settings. If the user closed the dialog box with the OK button, the application reads the values and proceeds accordingly.

Before showing the dialog box, the code of the Show Dialog Box button sets the values of certain controls in it. In the course of the application, it usually makes sense to suggest a few values in the dialog box, so that the user can accept the default values by pressing the Enter key. The main form selects a date on the dialog box's controls and then displays the dialog box with the statements given in Listing 7.5.

LISTING 7.5: Displaying a Dialog Box and Reading Its Values

```
Protected Sub Button3_Click(...) Handles Button3.Click
' Preselects the date 4/11/1980
   AgeDialog.cmbMonth.Text = "4"
   AgeDialog.cmbDay.Text = "11"
   AgeDialog.CmbYear.Text = "1980"
```

```
        AgeDialog.ShowDialog()
        If AgeDialog.DialogResult = DialogResult.OK Then
            MsgBox(AgeDialog.cmbMonth.Text & " " & _
                   AgeDialog.cmbDay.Text & ", " & _
                   AgeDialog.cmbYear.Text)
        Else
            MsgBox("OK, we'll protect your vital personal data")
        End If
    End Sub
```

To close the dialog box, you can click the OK or Cancel button. Each button sets the `DialogResult` property to indicate the action that closed the dialog box. The code behind the two buttons is shown in Listing 7.6.

Listing 7.6: Setting the Dialog Box's *DialogResult* Property

```
    Protected Sub bttnOK_Click(...) Handles bttnOK.Click
        Me.DialogResult = DialogResult.OK
    End Sub

    Protected Sub bttnCancel_Click(...) Handles bttnCancel.Click
        Me.DialogResult = DialogResult.Cancel
    End Sub
```

Because the dialog box is modal, the code in the Show Dialog Box button is suspended at the line that shows the dialog box. As soon as the dialog box is closed, the code in the main form resumes with the statement following the one that called the `ShowDialog` method of the dialog box. This is the `If` statement in Listing 7.5 that examines the value of the `DialogResult` property and acts accordingly.

Building Dynamic Forms at Runtime

Sometimes you won't know in advance how many instances of a given control might be required on a form. Let's say you're designing a form for displaying the names of all tables in a database. It's practically impossible to design a form that will accommodate every database users might throw at your application. Another typical example is a form for entering family related data, which includes the number of children in the family and their ages. As soon as the user enters (or changes) the number of children, you should display as many `TextBox` controls as there are children to collect their ages.

For these situations, it is possible to design *dynamic forms*, which are populated at runtime. The simplest approach is to create more controls than you'll ever need and set their `Visible` properties to False at design time. At runtime, you can display the controls by switching their `Visible` properties to True. As you know already, quick-and-dirty methods are not the most efficient ones. You must still rearrange the controls on the form to make it look nice at all times. The proper method to create dynamic forms at runtime is to add controls to and remove them from your form as needed, using the techniques discussed in this section.

Just as you can create new instances of forms, you can also create new instances of any control and place them on a form. The Form object exposes the `Controls` collection, which contains all the controls on the form. This collection is created automatically as you place controls on the form at design time, and you can access the members of this collection from within your code. It is also possible to add new members to the collection, or remove existing members, with the `Add` and `Remove` statements accordingly.

The Form's *Controls* Collection

All the controls on a form are stored in the `Controls` collection, which is a property of the Form object. The `Controls` collection exposes members for accessing and manipulating the controls at runtime, and they're the usual members of a collection:

***Add* method** The `Add` method adds a new element to the `Controls` collection. In effect, it adds a new control on the current form. The `Add` method accepts a reference to a control as an argument and adds it to the collection. Its syntax is the following, where *`controlObj`* is an instance of a control:

```
Controls.Add(controlObj)
```

To place a new Button control on the form, declare a variable of the Button type, set its properties, and then add it to the `Controls` collection:

```
Dim bttn As New System.WinForms.Button
bttn.Text = "New Button"
bttn.Left = 100
bttn.Top = 60
bttn.Width = 80
Me.Controls.Add(bttn)
```

***Remove* method** The `Remove` method removes an element from the `Controls` collection. It accepts as an argument either the index of the control to be removed or a reference to the control to be removed (a variable of the Control type that represents one of the controls on the form). The syntax of these two forms of the `Remove` method is the following:

```
Me.Controls.Remove(index)
Me.Controls.Remove(controlObj)
```

***Count* property** This property returns the number of elements in the `Controls` collection. Notice that if there are container controls, the controls in the containers are not included in the count. For example, if your form contains a Panel control, the controls on the panel won't be included in the value returned by the `Count` property. The Panel control, however, has its own `Controls` collection.

***All* method** This method returns all the controls on a form (or on a container control) as an array of the `System.WinForms.Control` type. You can iterate through the elements of this array with the usual methods exposed by the Array class.

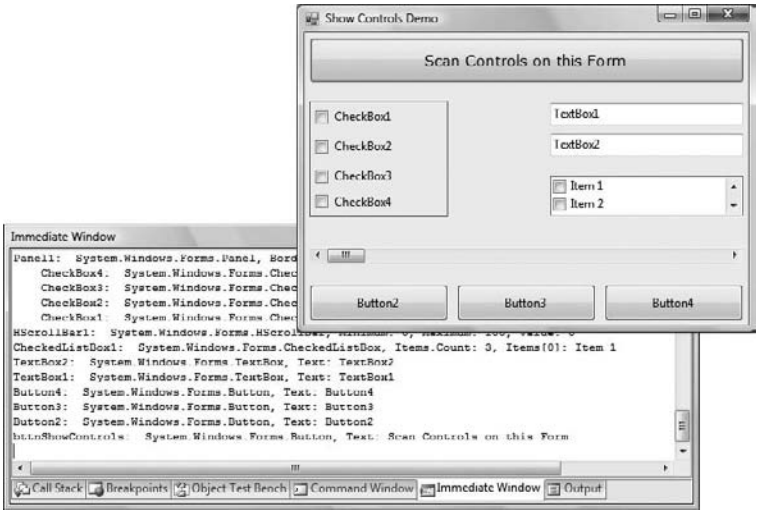
***Clear* method** The `Clear` method removes all the elements of the `Controls` array and effectively clears the form.

The `Controls` collection is also a property of any control that can host other controls. Many of the controls that come with VB 2008, such as the Panel control, can host other controls. As you recall from our discussion of the `Anchor` and `Dock` properties, it's customary to place controls on a panel and handle them collectively as a section of the form. They are moved along with the panel at design time, and they're rearranged as a group at runtime. The panel belongs to the form's `Controls` collection, and it provides its own `Controls` collection, which lets you access the controls on the panel.

VB 2008 at Work: The ShowControls Project

The ShowControls project (Figure 7.15) demonstrates the basic methods of the `Controls` array. Open the project and add any number of controls on its main form. You can place a panel to act as a container for other controls as well. Just don't remove the button at the top of the form (the Scan Controls On This Form button), which contains the code to list all the controls.

Figure 7.15
Accessing the controls on a form at runtime



```
Panel1:  System.Windows.Forms.Panel, Bord
    CheckBox4:  System.Windows.Forms.Chec
    CheckBox3:  System.Windows.Forms.Chec
    CheckBox2:  System.Windows.Forms.Chec
    CheckBox1:  System.Windows.Forms.Chec
HScrollBar1:  System.Windows.Forms.HScrollBar, Minimum: 0, Maximum: 100, Value: 0
CheckedListBox1:  System.Windows.Forms.CheckedListBox, Items.Count: 3, Items[0]: Item 1
TextBox2:  System.Windows.Forms.TextBox, Text: TextBox2
TextBox1:  System.Windows.Forms.TextBox, Text: TextBox1
Button4:  System.Windows.Forms.Button, Text: Button4
Button3:  System.Windows.Forms.Button, Text: Button3
Button2:  System.Windows.Forms.Button, Text: Button2
bttnShowControls:  System.Windows.Forms.Button, Text: Scan Controls on this Form
```

The code behind the Scan Controls On This Form button enumerates the elements of the form's `Controls` collection. The code doesn't take into consideration containers within containers. This would require a *recursive* routine, which would scan for controls at any depth. The code that iterates through the form's `Controls` collection and prints the names of the controls in the Output window is shown in Listing 7.7.

Listing 7.7: Iterating the Controls Collection

```
Private Sub Button1_Click(...) Handles Button1.Click
    Dim Control As Windows.Forms.Control
    For Each Control In Me.Controls
        Debug.WriteLine(Control.ToString)
        If Control.GetType Is GetType(System.Windows.Forms.Panel) Then
```

```
                Dim nestedControl As Windows.Forms.Control
                For Each nestedControl In Control.Controls
                    Debug.WriteLine("    " & nestedControl.ToString)
                Next
            End If
        Next
    End Sub
```

The form shown in Figure 7.15 produced the following (partial) output (the controls on the Panel are indented to stand out in the listing):

```
Panel1:          System.Windows.Forms.Panel,
BorderStyle:
System.Windows.Forms.BorderStyle.FixedSingle
    CheckBox4:   System.Windows.Forms.CheckBox, CheckState: 0
    CheckBox3:   System.Windows.Forms.CheckBox, CheckState: 0
HScrollBar1:     System.Windows.Forms.HScrollBar,
                 Minimum: 0, Maximum: 100, Value: 0
CheckedListBox1: System.Windows.Forms.CheckedListBox,
                 Items.Count: 3, Items[0]: Item 1
TextBox2:        System.Windows.Forms.TextBox,
                 Text: TextBox2
TextBox1:        System.Windows.Forms.TextBox,
                 Text: TextBox1
Button4:         System.Windows.Forms.Button,
                 Text: Button4
```

To find out the type of individual controls, call the GetType method. The following statement examines whether the control in the first element of the Controls collection is a TextBox:

```
If Me.Controls(0).GetType Is GetType(system.WinForms.TextBox) Then
   MsgBox("It's a TextBox control")
End If
```

Notice the use of the Is operator in the preceding statement. The equals operator will cause an exception because objects can be compared only with the Is operator. (You're comparing instances, not values.)

If you know the type's exact name, you can use a statement like the following:

```
If Me.Controls(i).GetType.Name = "TextBox" Then ...
```

To access other properties of the control represented by an element of the Controls collection, you must first cast it to the appropriate type. If the first control of the collection is a TextBox control, use the CType() function to cast it to a TextBox variable and then request its Text property:

```
If Me.Controls(0).GetType Is GetType(system.WinForms.TextBox) Then
   Debug.WriteLine(CType(Me.Controls(0), TextBox).Text)
End If
```

The `If` statement is necessary, unless you can be sure that the first control is a TextBox control. If you omit the `If` statement and attempt to convert the control to a TextBox, a runtime exception will be thrown if the object `Me.Controls(0)` isn't a TextBox control.

VB 2008 at Work: The DynamicForm Project

To demonstrate how to handle controls at runtime from within your code, I included the DynamicForm project (Figure 7.16), a simple data-entry window for a small number of data points. The user can specify at runtime the number of data points she wants to enter, and the number of TextBoxes on the form is adjusted automatically.

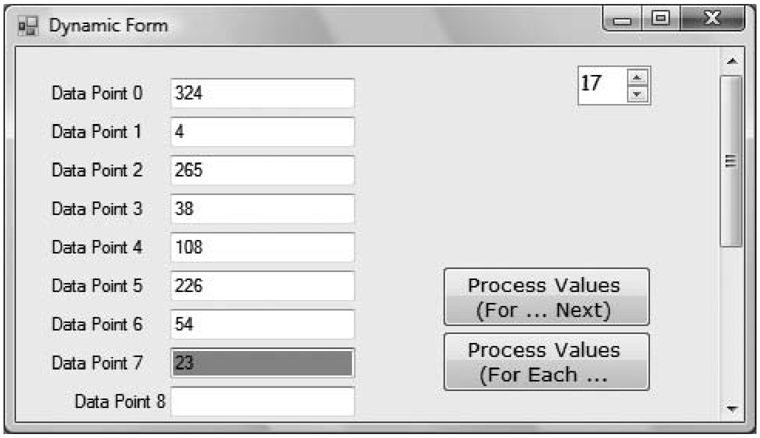


FIGURE 7.16 The DynamicForm project

The control you see at the top of the form is the NumericUpDown control. All you really need to know about this control is that it displays an integer in the range specified by its `Minimum` and `Maximum` properties and allows users to select a value. It also fires the `ValueChanged` event every time the user clicks one of the two arrows or types another value in its edit area. This event handler's code adds or removes controls on the form, so that the number of text boxes (as well as the number of corresponding labels) matches the value on the control. Listing 7.8 shows the handler for the `ValueChanged` event of the `NumericUpDown1` control. The `ValueChanged` event is fired when the user clicks one of the two arrows on the control or types a new value in the control's edit area.

LISTING 7.8: Adding and Removing Controls at Runtime

```
Private Sub NumericUpDown1_ValueChanged(...)  Handles NumericUpDown1.ValueChanged
    Dim TB As New TextBox()
    Dim LBL As New Label()
    Dim i, TBoxes As Integer
    '   Count all TextBox controls on the Form
    For i = 0 To Me.Controls.Count - 1
        If Me.Controls(i).GetType Is _
                    GetType(System.Windows.Forms.TextBox) Then
            TBoxes = TBoxes + 1
        End If
    Next
    ' Add new controls if number of controls on the Form is less
    ' than the number specified with the NumericUpDown control
    If TBoxes < NumericUpDown1.Value Then
        TB.Left = 100
        TB.Width = 120
        TB.Text = ""
        For i = TBoxes To CInt(NumericUpDown1.Value) - 1
            TB = New TextBox()
            LBL = New Label()
            If NumericUpDown1.Value = 1 Then
                TB.Top = 20
                TB.TabIndex = 0
            Else
                TB.Top = Me.Controls(Me.Controls.Count - 2).Top + 25
            End If
    ' Set the trivial properties of the new controls
    LBL.Left = 20
    LBL.Width = 80
    LBL.Text = "Data Point " & i
    LBL.Top = TB.Top + 3
    TB.Left = 100
    TB.Width = 120
    TB.Text = ""
    ' add controls to the form
    Me.Controls.Add(TB)
        Me.Controls.Add(LBL)
        TB.TabIndex = Convert.ToInt32(NumericUpDown1.Value)
        ' and finally connect their GotFocus/LostFocus events
        ' to the appropriate handler
        AddHandler TB.Enter, _
                     New System.EventHandler(AddressOf TBox_Enter)
        AddHandler TB.Leave, _
                     New System.EventHandler(AddressOf TBox_Leave)
            Next
    Else
        For i = Me.Controls.Count - 1 To Me.Controls.Count - _
                    2 * (TBoxes - CInt(NumericUpDown1.Value)) Step -2
            Me.Controls.Remove(Controls(i))
            Me.Controls.Remove(Controls(i - 1))
        Next
    End If
End Sub
```

Ignore the `AddHandler` statements for now; they're discussed in the following section. First, the code counts the number of TextBoxes on the form; then it figures out whether it should add or remove elements from the `Controls` collection. To remove controls, the code iterates through the last *n* controls on the form and removes them. The number of controls to be removed is the following, where *TBoxes* is the total number of controls on the form minus the value specified in the NumericUpDown control:

```
2 * (TBoxes - NumericUpDown1.Value)
```

If the value entered in the NumericUpDown control is less than the number of TextBox controls on the form, the code removes the excess controls from within a loop. At each step, it removes two controls, one of them a TextBox and the other a Label control with the matching caption. (That's why the loop variable is decreased by two.) The code also assumes that the first two controls on the form are the Button and the NumericUpDown controls. If the value entered by the user exceeds the number of TextBox controls on the form, the code adds the necessary pairs of TextBox and Label controls to the form.

To add controls, the code initializes a TextBox (*TB*) and a Label (*LBL*) variable. Then, it sets their locations and the label's caption. The left coordinate of all labels is 20, their width is 80, and their `Text` property (the label's caption) is the order of the data item. The vertical coordinate is 20 pixels for the first control, and all other controls are 3 pixels below the control on the previous row. After a new control is set up, it's added to the `Controls` collection with one of the following statements:

```
Me.Controls.Add(TB)     ' adds a TextBox control
Me.Controls.Add(LBL)    ' adds a Label control
```

The code contains a few long lines, but it isn't really complicated. It's based on the assumption that except for the first few controls on the form, all others are pairs of Label and TextBox controls used for data entry. You can simplify the code a little by placing the Label and TextBox controls on a Panel and manipulate the Panel's `Controls` collection. This collection contains only the data-entry controls, and the form may contain any number of additional controls.

To use the values entered by the user on the dynamic form, we must iterate the `Controls` collection, extract the values in the TextBox controls, and use them. Listing 7.9 shows how the top Process Values button scans the TextBox controls on the form and performs some basic calculations with them (counting the number of data points and summing their values).

LISTING 7.9: Reading the Controls on the Form

```
Private Sub Button1_Click(...) Handles Button1.Click
    Dim TBox As TextBox
    Dim Sum As Double = 0, points As Integer = 0
    Dim iCtrl As Integer
    For iCtrl = 0 To Me.Controls.Count - 1
        If Me.Controls(iCtrl).GetType Is _
                GetType(System.Windows.Forms.TextBox) Then
            TBox = CType(Me.Controls(iCtrl), TextBox)
            If IsNumeric(TBox.Text) Then
                Sum = Sum + Val(TBox.Text)
                points = points + 1
            End If
        End If
    Next
    MsgBox("The sum of the " & points.ToString & _
           " data points is " & Sum.ToString)
End Sub
```

You can add more statements to calculate the mean and other vital statistics, or you can process the values in any other way. You can even dump all the values into an array and then use the array notation to manipulate them.

The project's form has its `AutoScroll` property set to True, so that users can scroll it up and down if they specify a number of data points that exceeds the vertical dimension of the form. The two controls on the top-right side of the form, however, must remain at their location at all times. I placed them on a Panel control and added some code to the form's `Scroll` event handler, so that every time the user scrolls the form, the Panel control maintains its distance from the top and right edges of the form (otherwise, the two controls would scroll out of view). A single statement is all it takes to keep the Panel control in view at all times:

```
Private Sub Form1_Scroll(ByVal sender As Object, _
            ByVal e As System.Windows.Forms.ScrollEventArgs) _
            Handles Me.Scroll
    Panel1.Top = Panel1.Top + (e.NewValue - e.OldValue)
End Sub
```

You should try to redesign this application and place the data-entry controls on a Panel with its `AutoSize` and `AutoScroll` properties set to True.

The second button on the form does the exact same thing as the top one, only this one uses a `For Each ... Next` loop structure to iterate through the form's controls.

Creating Event Handlers at Runtime

You saw how to add controls on your forms at runtime and how to access the properties of these controls from within your code. In many situations, this is all you need: a way to access the properties of the controls (the text on a TextBox control or the status of a CheckBox or RadioButton control). What good is a Button control, however, if it can't react to the `Click` event? The only problem with the controls you add to the `Controls` collection at runtime is that they don't react to events. It's possible, though, to create event handlers at runtime, and this is what you'll learn in this section.

To create an event handler at runtime, create a subroutine that accepts two arguments — the usual `sender` and `e` arguments — and enter the code you want to execute when a specific control receives a specific event. The type of the `e` argument must match the definition of the second argument of the event for which you want to create a handler. Let's say that you want to add one or more buttons at runtime on your form, and these buttons should react to the `Click` event. Create the `ButtonClick()` subroutine and enter the appropriate code in it. The name of the subroutine can be anything; you don't have to make up a name that includes the control's or the event's name.

After the subroutine is in place, you must connect it to an event of a specific control. The `ButtonClick()` subroutine, for example, must be connected to the `Click` event of a Button control. The statement that connects a control's event to a specific event handler is the `AddHandler` statement, whose syntax is as follows:

```
AddHandler control.event, New System.EventHandler(AddressOf ButtonClick)
```

For example, to connect the ProcessNow() subroutine to the Click event of the Calculate button, use the following statement:

```
AddHandler Calculate.Click, _
     New System.EventHandler(AddressOf ProcessNow)
```

Let's add a little more complexity to the DynamicForm application. We'll program the Enter and Leave events of the TextBox controls added at runtime through the Me.Controls.Add method. When a TextBox control receives the focus, we'll change its background color to a light yellow, and when it loses the focus, we'll restore the background to white, so the user knows which box has the focus at any time. We'll use the same handlers for all TextBox controls. (The code of the two handlers is shown in Listing 7.10.)

Listing 7.10: Event Handlers Added at Runtime

```
Private Sub TBox_Enter(ByVal sender As Object, _
                       ByVal e As System.EventArgs)
   CType(sender, TextBox).BackColor = color.LightCoral
End Sub

Private Sub TBox_Leave(ByVal sender As Object, _
                       ByVal e As System.EventArgs)
   CType(sender, TextBox).BackColor = color.White
End Sub
```

The two subroutines use the sender argument to find out which TextBox control received or lost the focus, and they set the appropriate control's background color. (These subroutines are not event handlers yet, because they're not followed by the Handles keyword — at least, not before we associate them with an actual control and a specific event.) This process is done in the same segment of code that sets the properties of the controls we create dynamically at runtime. After adding the control to the Me.Controls collection, call the following statements to connect the new control's Enter and Leave events to the appropriate handlers:

```
AddHandler TB.Enter, New System.EventHandler(AddressOf TBox_Enter)
AddHandler TB.Leave, New System.EventHandler(AddressOf TBox_Leave)
```

Run the DynamicForm application and see how the TextBox controls handle the focus-related events. With a few statements and a couple of subroutines, we were able to create event handlers at runtime from within our code.

Designing an Application Generator

In the preceding sections of this chapter, you learned how to create new forms from within your code and how to instantiate them. In effect, you have the basic ingredients for designing applications from within your code. Designing an application programmatically is not a trivial task, but now you have a good understanding of how an application generator works. You can even design a wizard that prompts the user for information about the appearance of the form and then design the form from within your code.

Designing Menus

Menus are among the most common and most characteristic elements of the Windows user interface. Even in the old days of character-based displays, menus were used to display methodically organized choices and guide the user through an application. Despite the visually rich interfaces of Windows applications and the many alternatives, menus are still the most popular means of organizing a large number of options. Many applications duplicate some or all of their menus in the form of toolbar icons, but the menu is a standard fixture of a form. You can turn the toolbars on and off, but not the menus.

The Menu Editor

Menus can be attached only to forms, and they're implemented through the MenuStrip control. The items that make up the menu are ToolStripMenuItem objects. As you will see, the MenuStrip control and ToolStripMenuItem objects give you absolute control over the structure and appearance of the menus of your application. The MenuStrip control is a variation of the Strip control, which is the base of menus, toolbars, and status bars.

You can design menus visually and then program their `Click` event handlers. In principle, that's all there is to a menu: You specify its items (the menu's commands) and then you program each command's actions. Depending on the needs of your application, you might want to enable and disable certain commands, add context menus to some of the controls on your form, and so on. Because each item in a menu is represented by a ToolStripMenuItem object, you can control the application's menus from within your code by manipulating the properties of the ToolStripMenuItem objects. Let's start by designing a simple menu, and I'll show you how to manipulate the menu objects from within your code as we go along.

Double-click the MenuStrip icon in the Toolbox. (You'll find the MenuStrip control in the Menus & Toolbars tab of the Toolbox.) An instance of the MenuStrip control will be added to the form, and a single menu command will appear on your form. Its caption will be Type Here. If you don't see the first menu item on the form right away, select the MenuStrip control in the Components tray below the form. Do as the caption says: Click it and enter the first command's caption, **File**, as seen in Figure 7.17. To add items under the File menu, press Enter. To enter another command in the main menu, press Tab. Depending on your action, another box will be added, in which you can type the caption of the next command. Press Enter to move to the next item vertically, and Tab to move to the next item horizontally. To insert a separator, enter a hyphen (-) as the item's caption.

When you hover the pointer over a menu item, a drop-down button appears to the right of the item. Click this button to select the type of item you'll place on the menu. This item can be a MenuItem object, a separator, a ComboBox, or a TextBox. In this chapter, I'll focus on menu items, which are by far the most common elements on a menu. The last two options, however, allow you to build elaborate menus, reminiscent of the Office menus.

Enter the items of the File menu — **New**, **Open**, **Save**, **SaveAs**, and **Exit** — and then click somewhere on the form. All the temporary items (the ones with the Type Here caption) will disappear, and the menu will be finalized on the form.

To add the Edit menu, select the MenuStrip icon to activate the visual menu editor and then click the File item. In the new item that appears next to the File item on the control, enter the string **Edit**. Press Enter and you'll switch to the first item of the Edit menu. Fill the Edit menu with the usual editing commands. Table 7.4 shows the captions (property `Text`) and names (property `Name`) for each menu and each command. You can also insert a standard menu with the Insert Standard Items command of the MenuStrip object's context menu.

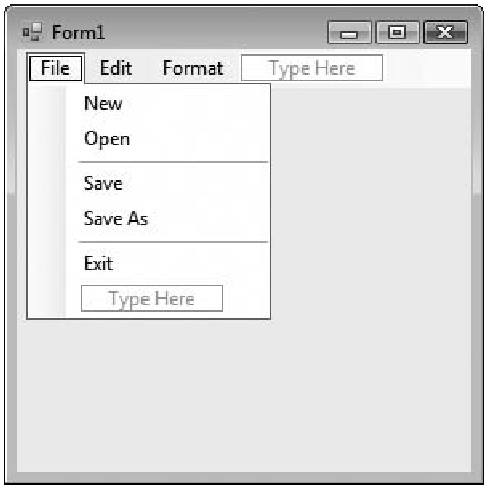


FIGURE 7.17
Designing a menu on the form

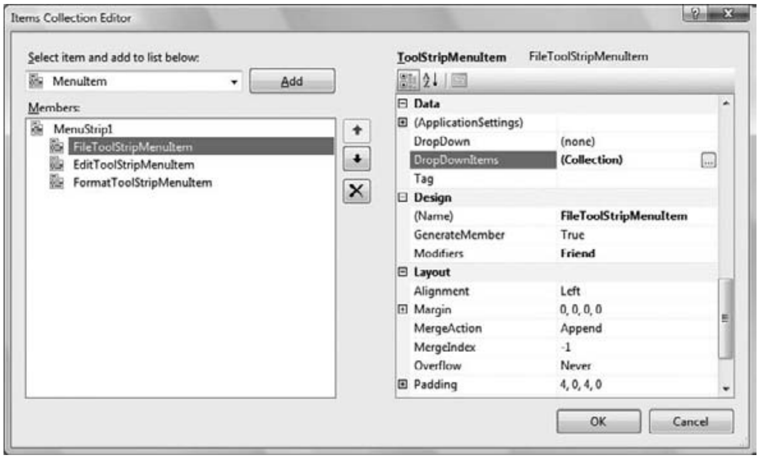
TABLE 7.4: The Captions and Names of the File and Edit Menus

CAPTION	**NAME**
File	FileMenu
New	FileNew
Open	FileOpen
Save	FileSave
Save As	FileSaveAs
Exit	FileExit
Edit	EditMenu
Copy	EditCopy
Cut	EditCut
Paste	EditPaste

The leftmost items in Table 7.4 are the names of the first-level menus (File and Edit); the captions that are indented in the table are the commands on these two menus. Each menu item has a name, which allows you to access its properties from within your code. The same name is also used in naming the `Click` event handler of the item. The default names of the menu items you add visually to the application's menu are based on the item's caption followed by the suffix ToolStripMenuItem (FileToolStripMenuItem, NewToolStripMenuItem, and so on). You'll probably want to change the default names to something less redundant. To do so, change the `Name` property in the Properties window. To view the properties of a menu item, right-click it and select Properties from the context menu.

The most convenient method of editing a menu is to use the Items Collection Editor window, which is shown in Figure 7.18. This isn't a visual editor, but you can set all the properties of each menu item without having to switch to the Properties window.

FIGURE 7.18
Editing a menu with the Items Collection Editor



```
MenuStrip1
    FileToolStripMenuItem
    EditToolStripMenuItem
    FormatToolStripMenuItem
```

```
ToolStripMenuItem   FileToolStripMenuItem

⊟ Data
⊞ (ApplicationSettings)
  DropDown            (none)
  DropDownItems       (Collection)
  Tag
⊟ Design
  (Name)              FileToolStripMenuItem
  GenerateMember      True
  Modifiers           Friend
⊟ Layout
  Alignment           Left
⊞ Margin              0, 0, 0, 0
  MergeAction         Append
  MergeIndex          -1
  Overflow            Never
⊞ Padding             4, 0, 4, 0
```

The Add button adds to the menu an item of the type specified in the combo box next to it (a menu item, combo box, or text box). To insert an item at a different location, add it to the menu and then use the arrow buttons to move it up or down. As you add new items, you can set their `Text` and `Name` properties on the right pane of the editor. You can also set their font, set the alignment and orientation of the text, and specify an image to be displayed along with the text. To add an image to a menu item, locate the `Image` property and click the ellipsis button. A dialog box will appear, in which you can select the appropriate resource. Notice that all the images you use on your form are stored as resources of the project. You can add all the images and icons you might need in a project to the same resource file and reuse them at will. The `TextImageRelation` property allows you to specify the relative positions of the text and the image. You can also select to display text only, images only, or text and images for each menu item with the `DisplayStyle` property.

If the menu item leads to a submenu, you must also specify the submenu's items. Locate the `DropDownItems` property and click the ellipsis button. An identical window will appear, in which you can enter the drop-down items of the current menu item. Notice that the menu on the form is continuously updated while you edit it in the Items Collection Editor window, so you can see the effects of your changes on the form. Personally, I'm more productive with the editor than with the visual tools, mainly because all the properties are right there, and I don't have to switch between the design surface and the Properties window.

The ToolStripMenuItem Properties

The ToolStripMenuItem class represents a menu command, at any level. If a command leads to a submenu, it's still represented by a ToolStripMenuItem object, which has its own collection of

ToolStripMenuItem objects: the `DropDownItems` collection, which is made up of ToolStripMenu-Item objects. The ToolStripMenuItem class provides the following properties, which you can set in the Properties window at design time or manipulate from within your code:

Checked Some menu commands act as toggles, and they are usually selected (checked) to indicate that they are on, or are deselected (unchecked) to indicate that they are off. To initially display a check mark next to a menu command, set its `Checked` property to True. You can also access this property from within your code to change the checked status of a menu command at runtime. For example, to toggle the status of a menu command called `FntBold`, use this statement:

```
FntBold.Checked = Not FntBold.Checked
```

Enabled Some menu commands aren't always available. The Paste command, for example, has no meaning if the Clipboard is empty (or if it contains data that can't be pasted in the current application). To indicate that a command can't be used at the time, you set its `Enabled` property to False. The command then appears grayed out in the menu, and although it can be highlighted, it can't be activated. The following statements enable and disable the Undo command depending on whether the *`TextBox1`* control can undo the most recent operation:

```
If TextBox1.CanUndo Then
   cmdUndo.Enabled = True
Else
   cmdUndo.Enabled = False
End  If
```

`cmdUndo` is the name of the Undo command in the application's Edit menu. The `CanUndo` property of the TextBox control returns a True/False value that indicates whether the last action can be undone or not.

IsOnDropDown If the menu command, represented by a ToolStripMenuItem object, belongs to a submenu, its `IsOnDropDown` property is True; otherwise, it's False. The `IsOnDropDown` property is read-only and False for the items on the first level of the menu.

Visible To remove a command temporarily from the menu, set the command's `Visible` property to False. The `Visible` property isn't used frequently in menu design. In general, you should prefer to disable a command to indicate that it can't be used at the time (some other action is required to enable it). Making a command invisible frustrates users, who might spend time trying to locate the command in another menu.

Programming Menu Commands

When a menu item is selected by the user, it triggers a `Click` event. To program a menu item, insert the appropriate code in the item's `Click` event handler. The Exit command's code would be something like the following:

```
Sub menuExit(...) Handles menuExit.Click
   End
End Sub
```

If you need to execute any cleanup code before the application ends, place it in the `CleanUp()` subroutine and call this subroutine from within the Exit item's `Click` event handler:

```
Sub menuExit(...) Handles menuExit.Click
   CleanUp()
   End
End Sub
```

The same subroutine must also be called from within the `FormClosing` event handler of the application's main form because some users might terminate the application by clicking the form's Close button.

An application's Open menu command contains the code that prompts the user to select a file and then open it. You will see many examples of programming menu commands in the following chapters. All you really need to know now is that each menu item is a ToolStripMenuItem object, and it fires the `Click` event every time it's selected with the mouse or the keyboard. In most cases, you can treat the `Click` event handler of a ToolStripMenuItem object just like the `Click` event handler of a Button.

Another interesting event of the ToolStripMenuItem is the `DropDownOpened` event, which is fired when the user opens a menu or submenu (in effect, when the user clicks a menu item that leads to a submenu). In this event's handler, you can insert code to modify the submenu. The Edit menu of just about any application contains the ubiquitous Cut/Copy/Paste commands. These commands are not meaningful at all times. If the Clipboard doesn't contain text, the Paste command should be disabled. If no text is selected, the Copy and Cut commands should also be disabled. Here's how you could change the status of the Paste command from within the `Drop-DownOpened` event handler of the Edit menu:

```
If My.Computer.Clipboard.ContainsText Then
    PasteToolStripMenuItem.Enabled = True
Else
    PasteToolStripMenuItem.Enabled = True
End If
```

Likewise, to change the status of the Cut and Copy commands, use the following statements in the `DropDownOpened` event of the ToolStripMenuItem that represents the Edit menu:

```
If txtEditor.SelectedText.Trim.Length > 0 Then
    CopyToolStripMenuItem.Enabled = True
    CutToolStripMenuItem.Enabled = True
Else
    CopyToolStripMenuItem.Enabled = False
    CutToolStripMenuItem.Enabled = False
End If
```

Using Access and Shortcut Keys

Menus provide a convenient way to display a large number of choices to the user. They allow you to organize commands in groups, according to their functions, and are available at all times. Opening menus and selecting commands with the mouse, however, can be an inconvenience. When using a word processor, for example, you don't want to have to take your hands off the keyboard and reach for the mouse. To simplify menu access, Windows forms support access keys and shortcut keys.

Access Keys

Access keys allow the user to open a menu by pressing the Alt key and a letter key. To open the Edit menu in all Windows applications, for example, you can press Alt+E. E is the Edit menu's access key. After the menu is open, the user can select a command with the arrow keys or by pressing another key, which is the command's access key, without holding down the Alt key.

Access keys are designated by the designer of the application and are marked with an underline character. To assign an access key to a menu item, insert the ampersand symbol (&) in front of the character you want to use as an access key in the ToolStripMenuItem's `Text` property.

Default Access Keys Are Based on Item Captions

If you don't designate access keys, Visual Basic will use the first character in each top-level menu as its access key. The user won't see the underline character under the first character, but can open the menu by pressing the first character of its caption while holding down the Alt key. If two or more menu captions begin with the same letter, the first (leftmost and topmost) menu will open.

Because the & symbol has a special meaning in menu design, you can't use it in a menu item's caption. To actually display the & symbol in a caption, prefix it with another & symbol. For example, the caption &Drag produces a command with the caption Drag (the first character is underlined because it's the access key). The caption Drag && Drop will create another command whose caption will be Drag & Drop. Finally, the string &Drag && Drop will create another command with the caption Drag & Drop.

Shortcut Keys

Shortcut keys are similar to access keys, but instead of opening a menu, they run a command when pressed. Assign shortcut keys to frequently used menu commands, so that users can reach them with a single keystroke. Shortcut keys are combinations of the Ctrl key and a function or character key. For example, the usual *access* key for the Close command (after the File menu is opened with Alt+F) is C, but the usual *shortcut* key for the Close command is Ctrl+W.

To assign a shortcut key to a menu command, drop down the `ShortcutKeys` list in the ToolStripMenuItem's Properties window and select a keystroke. Specify a modifier (Shift, Ctrl, or Alt) and a key. You don't have to insert any special characters in the command's caption, nor do you have to enter the keystroke next to the caption. It will be displayed next to the command automatically. When assigning access and shortcut keys, take into consideration the well-established Windows standards. Users expect Alt+F to open the File menu, so don't use Alt+F for the Format menu. Likewise, pressing Ctrl+C universally performs the Copy command; don't use Ctrl+C as a shortcut for the Cut command.

Manipulating Menus at Runtime

Dynamic menus change at runtime to display more or fewer commands, depending on the current status of the program. This section explores two techniques for implementing dynamic menus:

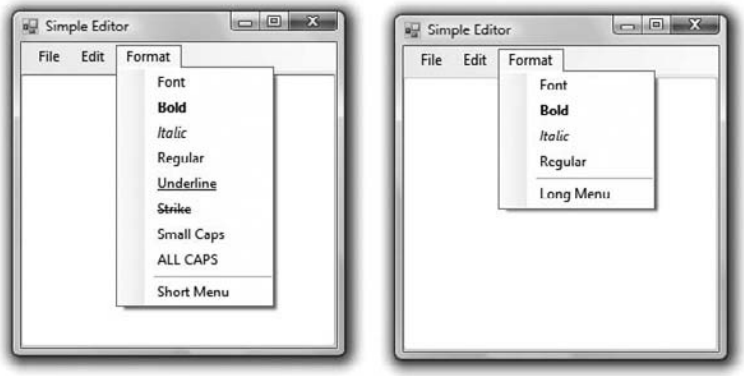
- ◆ Creating short and long versions of the same menu
- ◆ Adding and removing menu commands at runtime

Creating Short and Long Menus

A common technique in menu design is to create long and short versions of a menu. If a menu contains many commands, and most of the time only a few of them are needed, you can create one menu with all the commands and another with the most common ones. The first menu is the long one, and the second is the short one. The last command in the long menu should be Short Menu, and when selected, it should display the short version. The last command in the short menu should be Long Menu, and it should display the long version.

Figure 7.19 shows a long and a short version of the same menu from the LongMenu project. The short version omits infrequently used commands and is easier to handle.

Figure 7.19
The two versions of the Format menu of the LongMenu application



To implement the LongMenu command, start a new project and create a menu with the options shown in Figure 7.19. Listing 7.11 is the code that shows/hides the long menu in the MenuSize command's `Click` event.

Listing 7.11: The MenuSize Menu Item's *Click* Event

```
Private Sub mnuSize_Click(...) Handles mnuSize.Click
    If mnuSize.Text = "Short Menu" Then
        mnuSize.Text = "Long Menu"
    Else
        mnuSize.Text = "Short Menu"
    End If
    mnuUnderline.Visible = Not mnuUnderline.Visible
    mnuStrike.Visible = Not mnuStrike.Visible
    mnuSmallCaps.Visible = Not mnuSmallCaps.Visible
    mnuAllCaps.Visible = Not mnuAllCaps.Visible
End Sub
```

The subroutine in Listing 7.11 doesn't do much. It simply toggles the `Visible` property of certain menu commands and changes the command's caption to **Short Menu** or **Long Menu**, depending on the menu's current status. Notice that because the `Visible` property is a True/False value, we don't care about its current status; we simply toggle the current status with the `Not` operator.

Adding and Removing Commands at Runtime

I conclude the discussion of menu design with a technique for building dynamic menus, which grow and shrink at runtime. Many applications maintain a list of the most recently opened files in the File menu. When you first start the application, this list is empty, and as you open and close files, it starts to grow.

The RunTimeMenu project demonstrates how to add items to and remove items from a menu at runtime. The main menu of the application's form contains the Run Time Menu submenu, which is initially empty.

The two buttons on the form add commands to and remove commands from the Run Time Menu. Each new command is appended at the end of the menu, and the commands are removed from the bottom of the menu first (the most recently added commands are removed first). To change this order and display the most recent command at the beginning of the menu, use the `Insert` method instead of the `Add` method to insert the new item. Listing 7.12 shows the code behind the two buttons that add and remove menu items.

Listing 7.12: Adding and Removing Menu Items at Runtime

```
Private Sub bttnAddItem_Click(...) Handles bttnAddItem.Click
   Dim Item As New ToolStripMenuItem
   Item.Text = "Run Time Option" & _
             RunTimeMenuToolStripMenuItem. _
             DropDownItems.Count.ToString
   RunTimeMenuToolStripMenuItem.DropDownItems.Add(Item)
   AddHandler Item.Click, _
             New System.EventHandler(AddressOf OptionClick)
End Sub

Private Sub bttnRemoveItem_Click(...) Handles bttnRemoveItem.Click
   If RunTimeMenuToolStripMenuItem.DropDownItems.Count > 0 Then
       Dim mItem As ToolStripItem
       Dim items As Integer = _
                 RunTimeMenuToolStripMenuItem.DropDownItems.Count
       mItem = RunTimeMenuToolStripMenuItem.DropDownItems(items - 1)
   RunTimeMenuToolStripMenuItem.DropDownItems.Remove(mItem)
   End If
End Sub
```

The Remove button's code uses the `Remove` method to remove the last item in the menu by its index, after making sure the menu contains at least one item. The Add button adds a new item, sets its caption to Run Time Option *n*, where *n* is the item's order in the menu. In addition, it assigns an event handler to the new item's `Click` event. This event handler is the same for all the items added at runtime; it's the `OptionClick()` subroutine.

All the runtime options invoke the same event handler — it would be quite cumbersome to come up with a separate event handler for different items. In the single event handler, you can examine the name of the ToolStripMenuItem object that invoked the event handler and act accordingly. The `OptionClick()` subroutine used in Listing 7.13 displays the name of the menu item that invoked it. It doesn't do anything, but it shows you how to figure out which item of the Run Time Menu was clicked.

Listing 7.13: Programming Dynamic Menu Items

```
Private Sub OptionClick(...)
    Dim itemClicked As New ToolStripMenuItem
    itemClicked = CType(sender, ToolStripMenuItem)
    MsgBox("You have selected the item " & _
                   itemClicked.Text)
End Sub
```

Creating Context Menus

Nearly every Windows application provides a *context menu* that the user can invoke by right-clicking a form or a control. (It's sometimes called a *shortcut menu* or *pop-up menu*.) This is a regular menu, but it's not anchored on the form. It can be displayed anywhere on the form or on specific controls. Different controls can have different context menus, depending on the operations you can perform on them at the time.

To create a context menu, place a ContextMenuStrip control on your form. The new context menu will appear on the form just like a regular menu, but it won't be displayed there at runtime. You can create as many context menus as you need by placing multiple instances of the ContextMenuStrip control on your form and adding the appropriate commands to each one. To associate a context menu with a control on your form, set the *control's* ContextMenuStrip property to the name of the corresponding context menu.

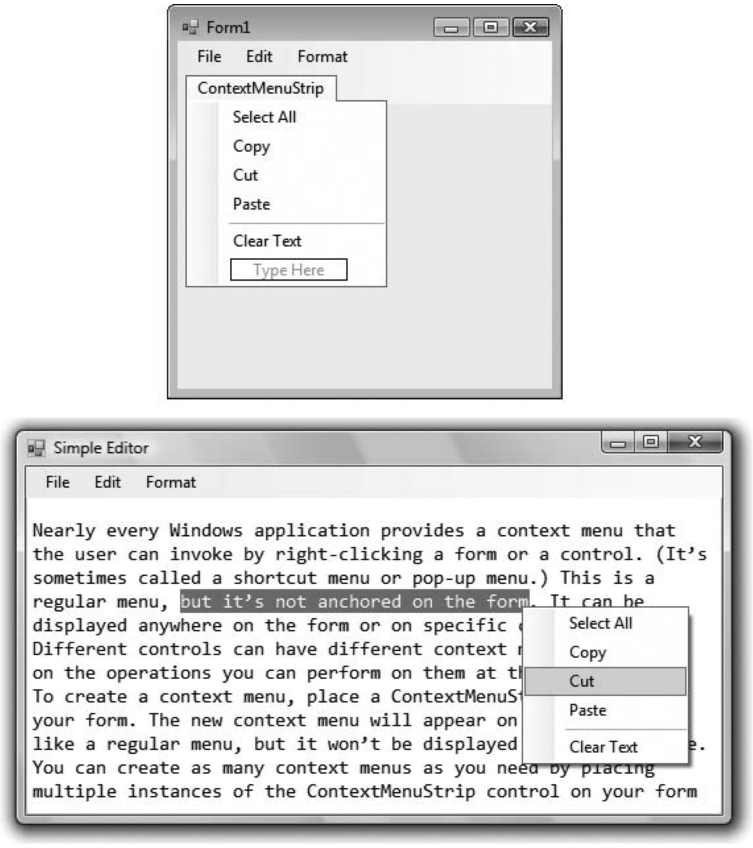
Designing a context menu is identical to designing a regular menu. The only difference is that the first command in the menu is always ContextMenuStrip and it's not displayed along with the menu. Figure 7.20 shows a context menu at design time and how the same menu is displayed at runtime.

You can create as many context menus as you want on a form. Each control has a `ContextMenu` property, which you can set to any of the existing ContextMenuStrip controls. Select the control for which you want to specify a context menu and locate the `ContextMenu` property in the Properties window. Expand the drop-down list and select the name of the desired context menu.

To edit one of the context menus on a form, select the appropriate ContextMenuStrip control at the bottom of the Designer. The corresponding context menu will appear on the form's menu bar, as if it were a regular form menu. This is temporary, however, and the only menu that appears on the form's menu bar at runtime is the one that corresponds to the MenuStrip control (and there can be only one of them on each form).

Iterating a Menu's Items

The last menu-related topic in this chapter demonstrates how to iterate through all the items of a menu structure, including their submenus, at any depth. The main menu of an application can be accessed by the expression `Me.MenuStrip1` (assuming that you're using the default names). This is a reference to the top-level commands of the menu, which appear in the form's menu bar. Each command, in turn, is represented by a ToolStripMenuItem object. All the items under a menu command form a `ToolStripMenuItems` collection, which you can scan to retrieve the individual commands.



```
Nearly every Windows application provides a context menu that
the user can invoke by right-clicking a form or a control. (It's
sometimes called a shortcut menu or pop-up menu.) This is a
regular menu, but it's not anchored on the form. It can be
displayed anywhere on the form or on specific c
Different controls can have different context m
on the operations you can perform on them at th
To create a context menu, place a ContextMenuSt
your form. The new context menu will appear on
like a regular menu, but it won't be displayed                e.
You can create as many context menus as you need by placing
multiple instances of the ContextMenuStrip control on your form
```

Figure 7.20
A context menu at design time (top) and at runtime (bottom)

The first command in a menu is accessed with the expression `Me.MenuStrip1.Items(0)`; this is the File command in a typical application. The expression `Me.MenuStrip1.Items(1)` is the second command on the same level as the File command (typically, the Edit menu).

To access the items *under* the first menu, use the `DropDownItems` collection of the top command. The first command in the File menu can be accessed by this expression:

```
Me.MenuStrip1.Items(0).DropDownItems(0)
```

The same items can be accessed by name as well, and this is how you should manipulate the menu items from within your code. In unusual situations, or if you're using dynamic menus to which you add and subtract commands at runtime, you'll have to access the menu items through the `DropDownItems` collection.

VB 2008 at Work: The MapMenu Project

The MapMenu project demonstrates how to access the items of a menu from within your application's code. The project's main form contains a menu, a TextBox control, and a Button control that prints the menu's structure in the TextBox. You can edit the menu before running the program, and the code behind the button will print the current structure of the menu items.

The code behind the Map Menu button iterates through the items of the form's MenuStrip items and prints their names, as well as the names of their drop-down items, in the Output window. It scans all the members of the control's `Items` collection and prints their captions. After printing each command's caption, it calls the `PrintSubMenu()` subroutine, passing the current ToolStripMenuItem as an argument. The `PrintSubMenu()` subroutine iterates through the DropDownItems of the collection passed as an argument and prints their captions. If one of the items leads to a nested submenu, it calls itself, passing the current ToolStripMenuItem as an argument. You can open the MapMenu project with Visual Studio and examine its code.

The Bottom Line

Use forms' properties. Forms expose a lot of trivial properties for setting their appearance. In addition, they expose a few properties that simplify the task of designing forms that can be resized at runtime. The `Anchor` property causes a control to be anchored to one or more edges of the form to which it belongs. The `Dock` property allows you to place on the form controls that are docked to one of its edges. To create forms with multiple panes that the user can resize at runtime, use the SplitContainer control. If you just can't fit all the controls in a reasonably sized form, use the `AutoScroll` properties to create a scrollable form.

Master It You've been asked to design a form with three distinct sections. You should also allow users to resize each section. How will you design this form?

Design applications with multiple forms. Typical applications are made up of multiple forms: the main form and one or more auxiliary forms. To show an auxiliary form from within the main form's code, call the auxiliary form's `Show` method, or the `ShowDialog` method if you want to display the auxiliary form modally (as a dialog box).

Master It How will you set the values of selected controls in a dialog box, display them, and then read the values selected by the user from the dialog box?

Design dynamic forms. You can create dynamic forms by populating them with controls at runtime through the form's `Controls` collection. First, create instances of the appropriate controls by declaring variables of the corresponding type. Then set the properties of the variable that represents the control. Finally, place the control on the form by adding it to the form's `Controls` collection.

Master It How will you add a TextBox control to your form at runtime and assign a handler to the control's `TextChanged` event?

Design menus. Both form menus and context menus are implemented through the MenuStrip control. The items that make up the menu are ToolStripMenuItem objects. The ToolStripMenuItem objects give you absolute control over the structure and appearance of the menus of your application.

Master It What are the two basic events fired by the ToolStripMenuItem object?

Chapter 8

More Windows Controls

In this chapter, we'll continue our discussion of the basic Windows controls with the controls that implement the common dialog boxes and the RichTextBox control.

The .NET Framework provides a set of controls for displaying common dialog boxes, such as the Open or Color dialog boxes. Each of these controls encapsulates a large amount of functionality that would take a lot of code to duplicate. The common dialog controls are fundamental components because they enable you to design user interfaces with the look and feel of a Windows application.

Besides the common dialog boxes, we'll also explore the RichTextBox control, which is an advanced version of the TextBox control. The RichTextBox control provides all the functionality you'll need to build a word processor — WordPad is actually built around the RichTextBox control. The RichTextBox control allows you to format text by mixing fonts and attributes, aligning paragraphs differently, and so on. You can also embed other objects in the document displayed in a RichTextBox, such as images. Sure, the RichTextBox control is nothing like a full-fledged word processor, but it's a great tool for editing formatted text at runtime.

In this chapter you'll learn how to do the following:

- Use the OpenFileDialog and SaveFileDialog controls to prompt users for filenames
- Use the ColorDialog and FontDialog controls to prompt users for colors and typefaces
- Use the RichTextBox control as an advanced text editor to present richly formatted text

The Common Dialog Controls

A rather tedious, but quite common, task in nearly every application is to prompt the user for filenames, font names and sizes, or colors to be used by the application. Designing your own dialog boxes for these purposes would be a hassle, not to mention that your applications wouldn't conform to the basic Windows interface design principles. In fact, all Windows applications use standard dialog boxes for common operations; two of them are shown in Figure 8.1. These dialog boxes are implemented as standard controls in the Toolbox. To use any of the common dialog controls in your interface, just place the appropriate control from the Dialog section of the Toolbox on your form and activate it from within your code by calling the `ShowDialog` method.

The common dialog controls are invisible at runtime, and they're not placed on your forms, because they're implemented as modal dialog boxes and they're displayed as needed. You simply add them to the project by double-clicking their icons in the Toolbox; a new icon appears in the components tray of the form, just below the Form Designer. The common dialog controls in the Toolbox are the following:

OpenFileDialog Lets users select a file to open. It also allows the selection of multiple files for applications that must process many files at once.

SaveFileDialog Lets users select or specify the path of a file in which the current document will be saved.

FolderBrowserDialog Lets users select a folder (an operation that can't be performed with the OpenFileDialog control).

ColorDialog Lets users select a color from a list of predefined colors or specify custom colors.

FontDialog Lets users select a typeface and style to be applied to the current text selection. The Font dialog box has an Apply button, which you can intercept from within your code and use to apply the currently selected font to the text without closing the dialog box.

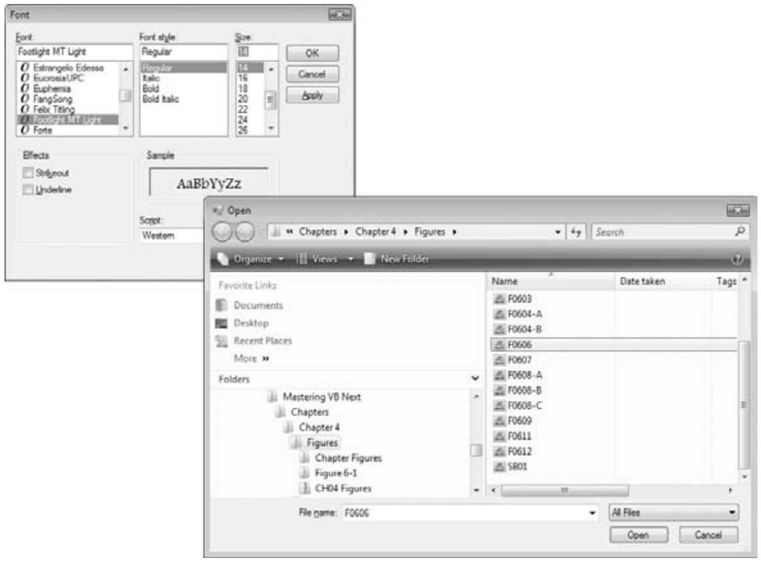


FIGURE 8.1
The Open and Font common dialog boxes

There are three more common dialog controls: the PrintDialog, PrintPreviewDialog, and PageSetupDialog controls. These controls are discussed in detail in Chapter 20, ''Printing with Visual Basic 2008,'' in the context of VB's printing capabilities.

Using the Common Dialog Controls

To display any of the common dialog boxes from within your application, you must first add an instance of the appropriate control to your project. Then you must set some basic properties of the control through the Properties window. Most applications set the control's properties from within the code because common dialogs interact closely with the application. When you call the Color common dialog, for example, you should preselect a color from within your application and make it the default selection on the control. When prompting the user for the color of the text, the default selection should be the current setting of the control's `ForeColor` property. Likewise, the Save dialog box must suggest a filename when it first pops up (or the file's extension, at least).

To display a common dialog box from within your code, you simply call the control's `ShowDialog` method, which is common for all controls. Note that all common dialog controls can be displayed only modally and they don't expose a `Show` method. As soon as you call the `ShowDialog` method, the corresponding dialog box appears onscreen, and the execution of the program is suspended until the box is closed. Using the Open, Save, and FolderBrowser dialog boxes, users can traverse the entire structure of their drives and locate the desired filename or folder. When the user clicks the Open or Save button, the dialog box closes and the program's execution resumes. The code should read the name of the file selected by the user through the `FileName` property and use it to open the file or store the current document there. The folder selected in the FolderBrowserDialog control is returned to the application through the `SelectedPath` property.

Here is the sequence of statements used to invoke the Open common dialog and retrieve the selected filename:

```
If OpenFileDialog1.ShowDialog = Windows.Forms.DialogResult.OK Then
   fileName = OpenFileDialog1.FileName
   ' Statements to open the selected file
End If
```

The `ShowDialog` method returns a value indicating how the dialog box was closed. You should read this value from within your code and ignore the settings of the dialog box if the operation was cancelled.

The variable *`fileName`* in the preceding code segment is the full pathname of the file selected by the user. You can also set the `FileName` property to a filename, which will be displayed when the Open dialog box is first opened:

```
OpenFileDialog1.FileName = _
           "C:\WorkFiles\Documents\Document1.doc"
If OpenFileDialog1.ShowDialog = _
              Windows.Forms.DialogResult.OK Then
   fileName = OpenFileDialog1.FileName
   ' Statements to open the selected file
End If
```

Similarly, you can invoke the Color dialog box and read the value of the selected color by using the following statements:

```
ColorDialog1.Color = TextBox1.BackColor
If ColorDialog1.ShowDialog = DialogResult.OK Then
   TextBox1.BackColor = ColorDialog1.Color
End If
```

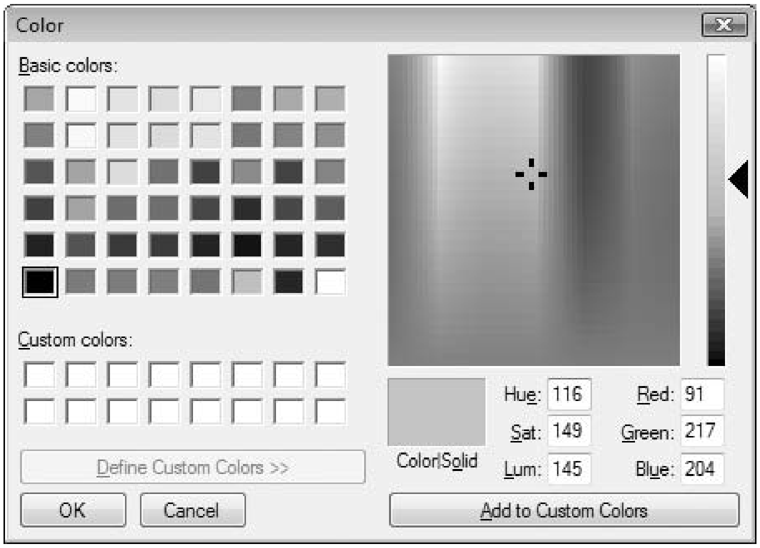
The `ShowDialog` method is common to all controls. The `Title` property is also common to all controls and it's the string displayed in the title bar of the dialog box. The default title is the name of the dialog box (for example, *Open*, *Color*, and so on), but you can adjust it from within your code with a statement such as the following:

```
ColorDialog1.Title = "Select Drawing Color"
```

The ColorDialog Control

The Color dialog box, shown in Figure 8.2, is one of the simplest dialog boxes. Its `Color` property returns the color selected by the user or sets the initially selected color when the user opens the dialog box.

FIGURE 8.2
The Color dialog box



Hue: 116 Sat: 149 Lum: 145 Red: 91 Green: 217 Blue: 204

The following statements set the initial color of the ColorDialog control, display the dialog box, and then use the color selected in the control to fill the form. First, place a ColorDialog control in the form and then insert the following statements in a button's `Click` event handler:

```
Private Sub Button1_Click(...) _
            Handles Button1.Click
   ColorDialog1.Color = Me.BackColor
   If ColorDialog1.ShowDialog =
Windows.Forms.DialogResult.OK Then
      Me.BackColor = ColorDialog1.Color
   End If
End Sub
```

The following sections discuss the basic properties of the ColorDialog control.

AllowFullOpen

Set this property to True if you want users to be able to open the dialog box and define their own custom colors, like the one shown in Figure 8.2. The `AllowFullOpen` property doesn't open the custom section of the dialog box; it simply enables the Define Custom Colors button in the dialog box. Otherwise, this button is disabled.

AnyColor

This property is a Boolean value that determines whether the dialog box displays all available colors in the set of basic colors.

Color

This is the color specified on the control. You can set it to a color value before showing the dialog box to suggest a reasonable selection. On return, read the value of the same property to find out which color was picked by the user in the control:

```
ColorDialog1.Color = Me.BackColor
If ColorDialog1.ShowDialog = DialogResult.OK Then
   Me.BackColor = ColorDialog1.Color
End If
```

CustomColors

This property indicates the set of custom colors that will be shown in the dialog box. The Color dialog box has a section called Custom Colors, in which you can display 16 additional custom colors. The `CustomColors` property is an array of integers that represent colors. To display three custom colors in the lower section of the Color dialog box, use a statement such as the following:

```
Dim colors() As Integer = {222663, 35453, 7888}
ColorDialog1.CustomColors = colors
```

You'd expect that the `CustomColors` property would be an array of Color values, but it's not. You can't create the array `CustomColors` with a statement such as this one:

```
Dim colors() As Color = _
           {Color.Azure, Color.Navy, Color.Teal}
```

Because it's awkward to work with numeric values, you should convert color values to integer values by using a statement such as the following:

```
Color.Navy.ToArgb
```

The preceding statement returns an integer value that represents the color navy. This value, however, is negative because the first byte in the color value represents the transparency of the color. To get the value of the color, you must take the absolute value of the integer value returned by the previous expression. To create an array of integers that represent color values, use a statement such as the following:

```
Dim colors() As Integer = _
           {Math.Abs(Color.Gray.ToArgb), _
            Math.Abs(Color.Navy.ToArgb), _
            Math.Abs(Color.Teal.ToArgb)}
```

Now you can assign the `colors` array to the `CustomColors` property of the control, and the colors will appear in the Custom Colors section of the Color dialog box.

SolidColorOnly

This indicates whether the dialog box will restrict users to selecting solid colors only. This setting should be used with systems that can display only 256 colors. Although today few systems can't display more than 256 colors, some interfaces are limited to this number. When you run an application through Remote Desktop, for example, only the solid colors are displayed correctly on the remote screen, regardless of the remote computer's graphics card (and that's for efficiency reasons).

The FontDialog Control

The Font dialog box, shown in Figure 8.3, lets the user review and select a font and then set its size and style. Optionally, users can also select the font's color and even apply the current settings to the selected text on a control of the form without closing the dialog box, by clicking the Apply button.

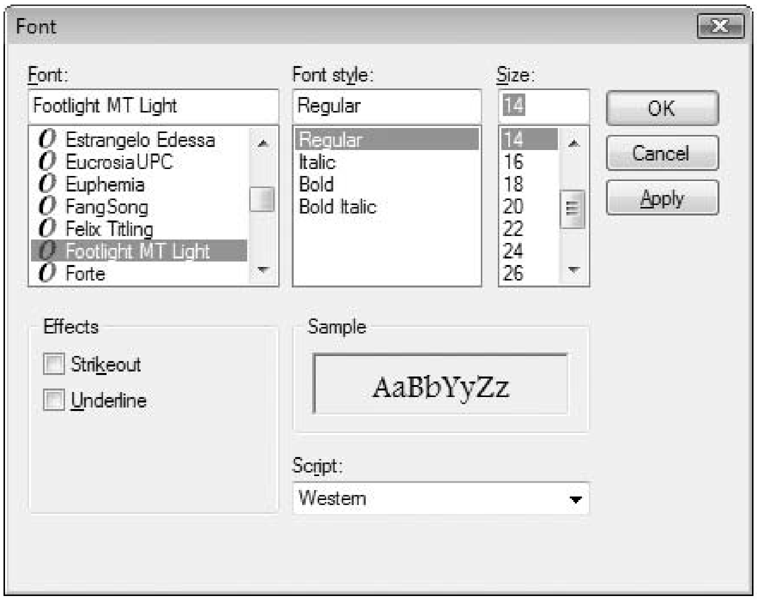


FIGURE 8.3
The Font dialog box

When the dialog is closed by clicking the OK button, you can retrieve the selected font by using the control's `Font` property. In addition to the OK button, the Font dialog box may contain the Apply button, which reports the current setting to your application. You can intercept the `Click` event of the Apply button and adjust the appearance of the text on your form while the common dialog is still visible.

The main property of this control is the `Font` property, which sets the initially selected font in the dialog box and retrieves the font selected by the user. The following statements display the Font dialog box after setting the initial font to the current font of the *`TextBox1`* control. When the user closes the dialog box, the code retrieves the selected font and assigns it to the same TextBox control:

```
FontDialog1.Font = TextBox1.Font
If FontDialog1.ShowDialog = DialogResult.OK Then
   TextBox1.Font = FontDialog1.Font
End If
```

Use the following properties to customize the Font dialog box before displaying it.

AllowScriptChange

This property is a Boolean value that indicates whether the Script combo box will be displayed in the Font dialog box. This combo box allows the user to change the current character set and select a non-Western language (such as Greek, Hebrew, Cyrillic, and so on).

AllowVerticalFonts

This property is a Boolean value that indicates whether the dialog box allows the display and selection of both vertical and horizontal fonts. Its default value is False, which displays only horizontal fonts.

Color, ShowColor

The `Color` property sets or returns the selected font color. To enable users to select a color for the font, you must also set the `ShowColor` property to True.

FixedPitchOnly

This property is a Boolean value that indicates whether the dialog box allows only the selection of fixed-pitch fonts. Its default value is False, which means that all fonts (fixed- and variable-pitch fonts) are displayed in the Font dialog box. Fixed-pitch fonts, or monospaced fonts, consist of characters of equal widths that are sometimes used to display columns of numeric values so that the digits are aligned vertically.

Font

This property is a Font object. You can set it to the preselected font before displaying the dialog box and assign it to a `Font` property upon return. You've already seen how to preselect a font and how to apply the selected font to a control from within your application.

You can also create a new Font object and assign it to the control's `Font` property. Upon return, the TextBox control's `Font` property is set to the selected font:

```
Dim newFont As Font("Verdana", 12, FontStyle.Underline)
FontDialog1.Font = newFont
If FontDialog1.ShowDialog() = DialogResult.OK Then
    TextBox1.ForeColor = FontDialog1.Color
End If
```

FontMustExist

This property is a Boolean value that indicates whether the dialog box forces the selection of an existing font. If the user enters a font name that doesn't correspond to a name in the list of available fonts, a warning is displayed. Its default value is True, and there's no reason to change it.

MaxSize, MinSize

These two properties are integers that determine the minimum and maximum point size the user can specify in the Font dialog box. Use these two properties to prevent the selection of extremely large or extremely small font sizes, because these fonts might throw off a well-balanced interface (text will overflow in labels, for example).

ShowApply

This property is a Boolean value that indicates whether the dialog box provides an Apply button. Its default value is False, so the Apply button isn't normally displayed. If you set this property to True, you must also program the control's `Apply` event — the changes aren't applied automatically to any of the controls in the current form.

The following statements display the Font dialog box with the Apply button:

```
Private Sub Button2_Click(...) Handles Button2.Click
   FontDialog1.Font = TextBox1.Font
   FontDialog1.ShowApply = True
   If FontDialog1.ShowDialog = DialogResult.OK Then
       TextBox1.Font = FontDialog1.Font
   End If
End Sub
```

The FontDialog control raises the `Apply` event every time the user clicks the Apply button. In this event's handler, you must read the currently selected font and use it in the form, so that users can preview the effect of their selection:

```
Private Sub FontDialog1_Apply(...) Handles FontDialog1.Apply
   TextBox1.Font = FontDialog1.Font
End Sub
```

ShowEffects

This property is a Boolean value that indicates whether the dialog box allows the selection of special text effects, such as strikethrough and underline. The effects are returned to the application as attributes of the selected Font object, and you don't have to do anything special in your application.

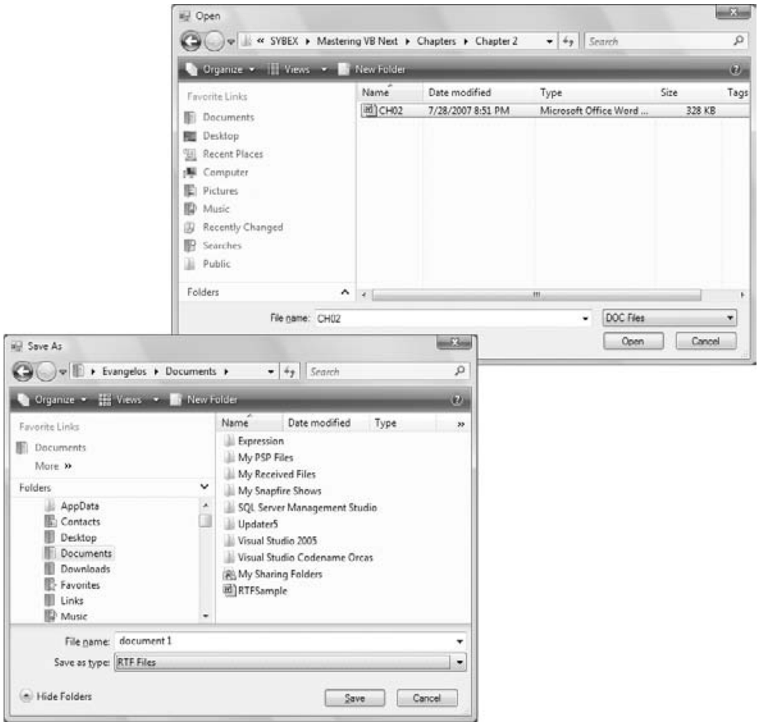
The OpenDialog and SaveDialog Controls

Open and Save As, the two most widely used common dialog boxes (see Figure 8.4), are implemented by the OpenFileDialog and SaveFileDialog controls. Nearly every application prompts users for filenames, and the .NET Framework provides two controls for this purpose. The two dialog boxes are nearly identical, and most of their properties are common, so we'll start with the properties that are common to both controls.

When either of the two controls is displayed, it rarely displays all the files in any given folder. Usually the files displayed are limited to the ones that the application recognizes so that users can easily spot the file they want. The `Filter` property limits the types of files that will appear in the Open or Save As dialog box.

It's also standard for the Windows interface not to display the extensions of files (although Windows distinguishes files by their extensions). The file type ComboBox, which appears at the bottom of the form next to the File Name box, contains the various file types recognized by the application. The various file types can be described in plain English with long descriptive names and without their extensions.

FIGURE 8.4
The Open and Save As common dialog boxes



The extension of the default file type for the application is described by the `DefaultExtension` property, and the list of the file types displayed in the Save As Type box is determined by the `Filter` property.

To prompt the user for a file to be opened, use the following statements. The Open dialog box displays the files with the extension `.bin` only.

```
OpenFileDialog1.DefaultExt = ".bin"
OpenFileDialog1.AddExtension = True
OpenFileDialog1.Filter = "Binary Files|*.bin"
If OpenFileDialog1.ShowDialog() = _
               Windows.Forms.DialogResult.OK Then
   Debug.WriteLine(OpenFileDialog1.FileName)
End If
```

The following sections describe the properties of the OpenFileDialog and SaveFileDialog controls.

ADDEXTENSION

This property is a Boolean value that determines whether the dialog box automatically adds an extension to a filename if the user omits it. The extension added automatically is the one specified by the DefaultExtension property, which you must set before calling the ShowDialog method. This is the default extension of the files recognized by your application.

CheckFileExists

This property is a Boolean value that indicates whether the dialog box displays a warning if the user enters the name of a file that does not exist in the Open dialog box, or if the user enters the name of a file that exists in the Save dialog box.

CheckPathExists

This property is a Boolean value that indicates whether the dialog box displays a warning if the user specifies a path that does not exist, as part of the user-supplied filename.

DefaultExt

This property sets the default extension for the filenames specified on the control. Use this property to specify a default filename extension, such as `.txt` or `.doc`, so that when a file with no extension is specified by the user, the default extension is automatically appended to the filename. You must also set the `AddExtension` property to True. The default extension property starts with the period, and it's a string — for example, `.bin`.

DereferenceLinks

This property indicates whether the dialog box returns the location of the file referenced by the shortcut or the location of the shortcut itself. If you attempt to select a shortcut on your desktop when the `DereferenceLinks` property is set to False, the dialog box will return to your application a value such as `C:\WINDOWS\SYSTEM32\lnkstub.exe`, which is the name of the shortcut, not the name of the file represented by the shortcut. If you set the `DereferenceLinks` property to True, the dialog box will return the actual filename represented by the shortcut, which you can use in your code.

FileName

Use this property to retrieve the full path of the file selected by the user in the control. If you set this property to a filename before opening the dialog box, this value will be the proposed filename. The user can click OK to select this file or select another one in the control. The two controls provide another related property, the `FileNames` property, which returns an array of filenames. To find out how to allow the user to select multiple files, see the discussion of the `MultipleFiles` and `FileNames` properties in ''VB 2008 at Work: Multiple File Selection'' at the end of this section.

Filter

This property is used to specify the type(s) of files displayed in the dialog box. To display text files only, set the `Filter` property to `Text files|*.txt`. The pipe symbol separates the description of the files (what the user sees) from the actual extension (how the operating system distinguishes the various file types).

If you want to display multiple extensions, such as `.BMP`, `.GIF`, and `.JPG`, use a semicolon to separate extensions with the `Filter` property. Set the Filter property to the string **Images|*.BMP;**

***.GIF;*.JPG** to display all the files of these three types when the user selects **Images** in the Save As Type combo box, under the box with the filename.

Don't include spaces before or after the pipe symbol because these spaces will be displayed on the dialog box. In the Open dialog box of an image-processing application, you'll probably provide options for each image file type, as well as an option for all images:

```
OpenFileDialog1.Filter = _
      "Bitmaps|*.BMP|GIF Images|*.GIF|" & _
      "JPEG Images|*.JPG|All Images|*.BMP;*.GIF;*.JPG"
```

FilterIndex

When you specify more than one file type when using the `Filter` property of the Open dialog box, the first file type becomes the default. If you want to use a file type other than the first one, use the `FilterIndex` property to determine which file type will be displayed as the default when the Open dialog box is opened. The index of the first type is 1, and there's no reason to ever set this property to 1. If you use the `Filter` property value of the example in the preceding section and set the `FilterIndex` property to 2, the Open dialog box will display GIF files by default.

InitialDirectory

This property sets the initial folder whose files are displayed the first time that the Open and Save dialog boxes are opened. Use this property to display the files of the application's folder or to specify a folder in which the application stores its files by default. If you don't specify an initial folder, the dialog box will default to the last folder where the most recent file was opened or saved. It's also customary to set the initial folder to the application's path by using the following statement:

```
OpenFileDialog1.InitialDirectory = Application.ExecutablePath
```

The expression `Application.ExecutablePath` returns the path in which the application's executable file resides.

RestoreDirectory

Every time the Open and Save As dialog boxes are displayed, the current folder is the one that was selected by the user the last time the control was displayed. The `RestoreDirectory` property is a Boolean value that indicates whether the dialog box restores the current directory before closing. Its default value is False, which means that the initial directory is not restored automatically. The `InitialDirectory` property overrides the `RestoreDirectory` property.

The following four properties are properties of the OpenFileDialog control only: `FileNames`, `MultiSelect`, `ReadOnlyChecked`, and `ShowReadOnly`.

FileNames

If the Open dialog box allows the selection of multiple files (see the later section ''VB 2008 at Work: Multiple File Selection''), the `FileNames` property contains the pathnames of all selected files. `FileNames` is a collection, and you can iterate through the filenames with an enumerator. This property should be used only with the OpenFileDialog control, even though the SaveFileDialog control exposes a `FileNames` property.

MultiSelect

This property is a Boolean value that indicates whether the user can select multiple files in the dialog box. Its default value is False, and users can select a single file. When the `MultiSelect` property is True, the user can select multiple files, but they must all come from the same folder (you can't allow the selection of multiple files from different folders). This property is unique to the OpenFileDialog control.

ReadOnlyChecked, ShowReadOnly

The `ReadOnlyChecked` property is a Boolean value that indicates whether the Read-Only check box is selected when the dialog box first pops up (the user can clear this box to open a file in read/write mode). You can set this property to True only if the `ShowReadOnly` property is also set to True. The `ShowReadOnly` property is also a Boolean value that indicates whether the Read-Only check box is available. If this check box appears on the form, the user can select it so the file will be opened as read-only. Files opened as read-only shouldn't be saved onto the same file — always prompt the user for a new filename.

The *OpenFile* and *SaveFile* Methods

The OpenFileDialog control exposes the `OpenFile` method, which allows you to quickly open the selected file. Likewise, the SaveFileDialog control exposes the `SaveFile` method, which allows you to quickly save a document to the selected file. Normally, after retrieving the name of the file selected by the user, you must open this file for reading (in the case of the Open dialog box) or writing (in the case of the Save dialog box). The topic of reading from or writing to files is discussed in detail in Chapter 15, ''Accessing Folders and Files.''

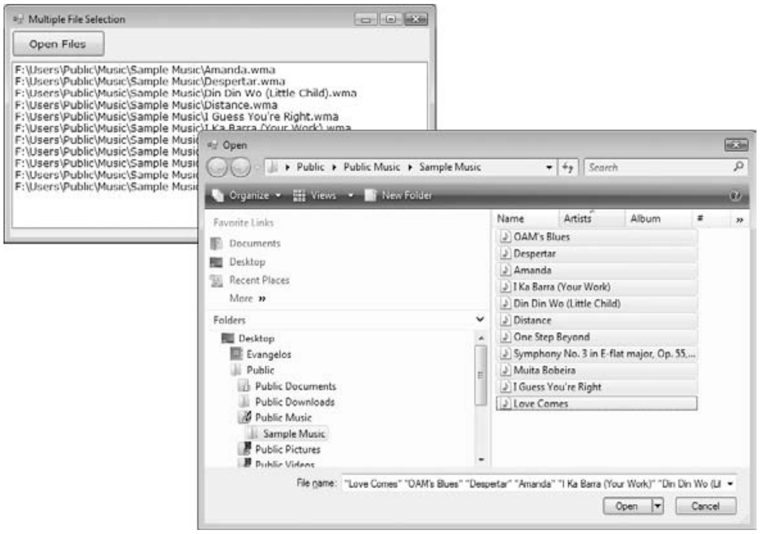
When this method is applied to the Open dialog box, the file is opened with read-only permission. The same method can be applied to the SaveFile dialog box, in which case the file is opened with read-write permission. Both methods return a Stream object, and you can call this object's `Read` and `Write` methods to read from or write to the file.

VB 2008 at Work: Multiple File Selection

The Open dialog box allows the selection of multiple files. This feature can come in handy when you want to process files en masse. You can let the user select many files, usually of the same type, and then process them one at a time. Or, you might want to prompt the user to select multiple files to be moved or copied.

To allow the user to select multiple files in the Open dialog box, set the `MultiSelect` property to True. The user can then select multiple files with the mouse by holding down the Shift or Ctrl key. The names of the selected files are reported by the property `FileNames`, which is an array of strings. The `FileNames` array contains the pathnames of all selected files, and you can iterate through them and process each file individually.

One of this chapter's sample projects is the MultipleFiles project, which demonstrates how to use the `FileNames` property. The application's form is shown in Figure 8.5. The button at the top of the form displays the Open dialog box, where you can select multiple files. After closing the dialog box by clicking the Open button, the application displays the pathnames of the selected files on a ListBox control.



```
F:\Users\Public\Music\Sample Music\Amanda.wma
F:\Users\Public\Music\Sample Music\Despertar.wma
F:\Users\Public\Music\Sample Music\Din Din Wo (Little Child).wma
F:\Users\Public\Music\Sample Music\Distance.wma
F:\Users\Public\Music\Sample Music\I Guess You're Right.wma
F:\Users\Public\Music\Sample Music\I Ka Barra (Your Work).wma
F:\Users\Public\Music\Sample Musi
F:\Users\Public\Music\Sample Musi
F:\Users\Public\Music\Sample Musi
F:\Users\Public\Music\Sample Musi
F:\Users\Public\Music\Sample Musi
```

File name: "Love Comes" "OAM's Blues" "Despertar" "Amanda" "I Ka Barra (Your Work)" "Din Din Wo (Li

FIGURE 8.5 The MultipleFiles project lets the user select multiple files in the Open dialog box.

The code behind the Open Files button is shown in Listing 8.1. In this example, I used the array's enumerator to iterate through the elements of the `FileNames` array. You can use any of the methods discussed in Chapter 2, ''The Visual Basic 2008 Language,'' to iterate through the array.

LISTING 8.1: Processing Multiple Selected Files

```
Private Sub bttnFile_Click(...) _
            Handles bttnFile.Click
    OpenFileDialog1.Multiselect = True
    OpenFileDialog1.ShowDialog()
    Dim filesEnum As IEnumerator
    ListBox1.Items.Clear()
    filesEnum = OpenFileDialog1.FileNames.GetEnumerator()
    While filesEnum.MoveNext
       ListBox1.Items.Add(filesEnum.Current)
    End While
End Sub
```

The FolderBrowserDialog Control

Sometimes we need to prompt users for a folder, rather than a filename. An application that processes files in batch mode shouldn't force users to select the files to be processed. Instead, it should allow users to select a folder and process all files of a specific type in the folder (it could encrypt all text documents or resize all image files, for example). As elaborate as the File Open dialog box might be, it doesn't allow the selection of a folder. To prompt users for a folder's path, use the FolderBrowser dialog box, which is a very simple one; it's shown in Figure 8.6 in the section ''VB 2008 at Work: Folder Browsing Demo Project.'' The FolderBrowserDialog control exposes a small number of properties, which are discussed next.

RootFolder

This property indicates the initial folder to be displayed when the dialog box is shown. It is not necessarily a string; it can also be a member of the `SpecialFolder` enumeration. To see the members of the enumeration, enter the following expression:

```
FolderBrowserDialog1.RootFolder =
```

As soon as you enter the equals sign, you will see the members of the enumeration. The most common setting for this property is My Computer, which represents the target computer's file system. You can set the `RootFolder` property to a number of special folders (for example, Personal, Desktop, ApplicationData, LocalApplicationData, and so on). You can also set this property to a string with the desired folder's pathname.

SelectedFolder

After the user closes the FolderBrowser dialog box by clicking the OK button, you can retrieve the name of the selected folder with the `SelectedFolder` property, which is a string, and you can use it with the methods of the `System.IO` namespace to access and manipulate the selected folder's files and subfolders.

ShowNewFolderButton

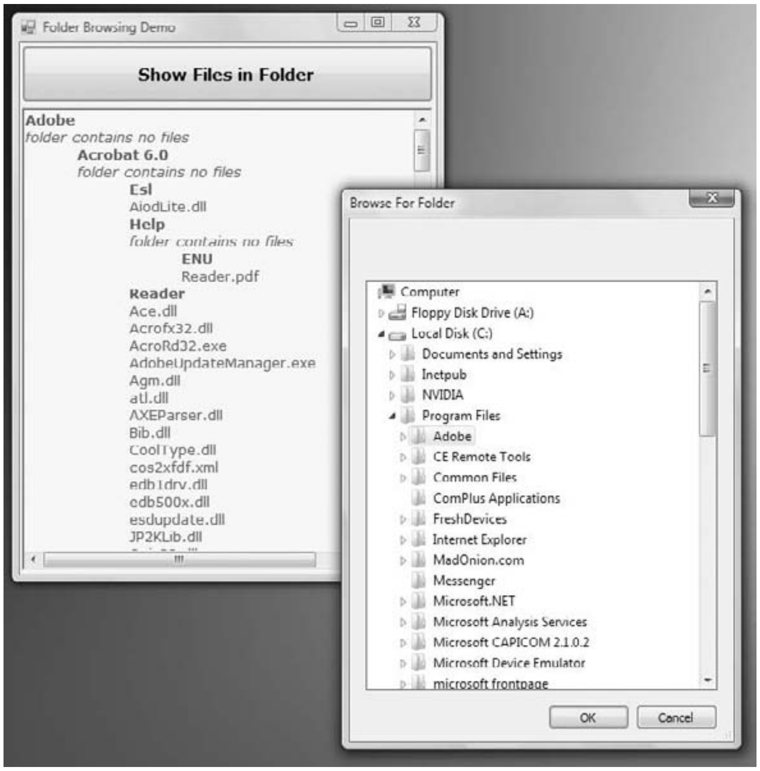
This property determines whether the dialog box will contain a New button; its default value is True. When users click the New button to create a new folder, the dialog box prompts them for the new folder's name, and creates a new folder with the specified name under the selected folder.

VB 2008 at Work: Folder Browsing Demo Project

The FolderBrowser control is a trivial control, but I'm including a sample application to demonstrate its use. The same application demonstrates how to retrieve the files and subfolders of the selected folder and how to create a directory listing in a RichTextBox control, like the one shown in Figure 8.6. The members of the `System.IO` namespace, which allow you to access and manipulate files and folders from within your code, are discussed in detail in Chapter 15.

The FolderBrowser dialog box is set to display the entire file system of the target computer and is invoked with the following statements:

```
FolderBrowserDialog1.RootFolder = Environment.SpecialFolder.MyComputer
FolderBrowserDialog1.ShowNewFolderButton = False
If FolderBrowserDialog1.ShowDialog = DialogResult.OK Then
' process files in selected folder
End If
```



```
Adobe
folder contains no files
        Acrobat 6.0
        folder contains no files
                Esl
                AiodLite.dll
                Help
                folder contains no files
                        ENU
                        Reader.pdf
                Reader
                Ace.dll
                Acrofx32.dll
                AcroRd32.exe
                AdobeUpdateManager.exe
                Agm.dll
                atl.dll
                AXEParser.dll
                Bib.dll
                CoolType.dll
                cos2xfdf.xml
                edb1drv.dll
                edb500x.dll
                esdupdate.dll
                JP2KLib.dll
```

FIGURE 8.6 Selecting a folder via the FolderBrowser dialog box

As usual, we examine the value returned by the ShowDialog method of the control and we proceed if the user has closed the dialog box by clicking the OK button. The code that iterates through the selected folder's files and subfolders, shown in Listing 8.2, is basically a demonstration of some members of the System.IO namespace, but I'll review it briefly here.

LISTING 8.2: Scanning a Folder

```
Private Sub bttnSelectFiles_Click(...) _
            Handles bttnSelectFiles.Click
    FolderBrowserDialog1.RootFolder = _
                 Environment.SpecialFolder.MyComputer
    FolderBrowserDialog1.ShowNewFolderButton = False
    If FolderBrowserDialog1.ShowDialog = Windows.Forms.DialogResult.OK Then
        RichTextBox1.Clear()
        ' Retrieve initial folder
        Dim initialFolder As String = _
                   FolderBrowserDialog1.SelectedPath
        Dim InitialDir As New IO.DirectoryInfo( _
                     FolderBrowserDialog1.SelectedPath)
        ' and print its name w/o any indentation
        PrintFolderName(InitialDir, "")
        ' and then print the files in the top folder
        If InitialDir.GetFiles("*.*").Length = 0 Then
            SwitchToItalics()
            RichTextBox1.AppendText( _
                 "folder contains no files" & vbCrLf)
            SwitchToRegular()
        Else
            PrintFileNames(InitialDir, "")
        End If
        Dim DI As IO.DirectoryInfo
        ' Iterate through every subfolder and print it
        For Each DI In InitialDir.GetDirectories
            PrintDirectory(DI)
        Next
    End If
End Sub
```

The selected folder's name is stored in the *`initialFolder`* variable and is passed as an argument to the constructor of the DirectoryInfo class. The *`InitialDir`* variable represents the specified folder. This object is passed to the `PrintFolderName()` subroutine, which prints the folder's name in bold. Then the code iterates through the same folder's files and prints them with the `PrintFileNames()` subroutine, which accepts as an argument the DirectoryInfo object that represents the current folder and the indentation level. After printing the initial folder's name and the names of the files in the folder, the code iterates through the subfolders of the initial folder. The `GetDirectories` method of the DirectoryInfo class returns a collection of objects, one for each subfolder under the folder represented by the *`InitialDir`* variable. For each subfolder, it calls the `PrintDirectory()` subroutine, which prints the folder's name and the files in this folder, and then iterates through the folder's subfolders. The code that iterates through the selected folder's files and subfolders is shown in Listing 8.3.

Listing 8.3: The PrintDirectory() Subroutine

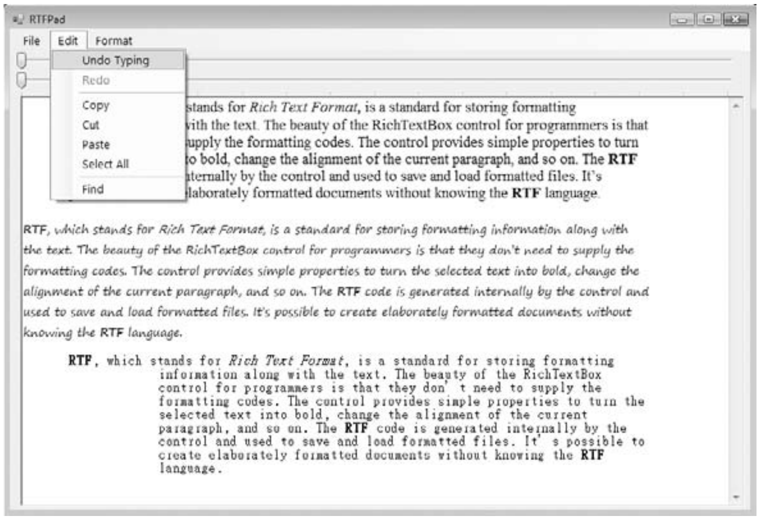
```
    Private Sub PrintDirectory(ByVal CurrentDir As IO.DirectoryInfo)
        Static IndentationLevel As Integer = 0
        IndentationLevel += 1
        Dim indentationString As String = ""
        indentationString = _
        New String(Convert.ToChar(vbTab), IndentationLevel)
        PrintFolderName(CurrentDir, indentationString)
        If CurrentDir.GetFiles("*.*").Length = 0 Then
            SwitchToItalics()
            RichTextBox1.AppendText(indentationString & _
                     "folder contains no files" & vbCrLf)
            SwitchToRegular()
        Else
            PrintFileNames(CurrentDir, indentationString)
        End If
        Dim folder As IO.DirectoryInfo
        For Each folder In CurrentDir.GetDirectories
            PrintDirectory(folder)
        Next
        IndentationLevel -= 1
    End Sub
```

The code that iterates through the subfolders of a given folder is discussed in detail in Chapter 15, so you need not worry if you can't figure out how it works yet. In the following section, you'll learn how to display formatted text in the RichTextBox control.

The RichTextBox Control

The RichTextBox control is the core of a full-blown word processor. It provides all the functionality of a TextBox control; it can handle multiple typefaces, sizes, and attributes, and offers precise control over the margins of the text (see Figure 8.7). You can even place images in your text on a RichTextBox control (although you won't have the kind of control over the embedded images that you have with Microsoft Word).

FIGURE 8.7
A word processor based on the functionality of the RichTextBox control



stands for *Rich Text Format*, is a standard for storing formatting
vith the text. The beauty of the RichTextBox control for programmers is that
upply the formatting codes. The control provides simple properties to turn
o bold, change the alignment of the current paragraph, and so on. The **RTF**
ternally by the control and used to save and load formatted files. It's
laborately formatted documents without knowing the **RTF** language.

RTF, *which stands for Rich Text Format, is a standard for storing formatting information along with the text. The beauty of the RichTextBox control for programmers is that they don't need to supply the formatting codes. The control provides simple properties to turn the selected text into bold, change the alignment of the current paragraph, and so on. The* **RTF** *code is generated internally by the control and used to save and load formatted files. It's possible to create elaborately formatted documents without knowing the* **RTF** *language.*

RTF, which stands for *Rich Text Format*, is a standard for storing formatting information along with the text. The beauty of the RichTextBox control for programmers is that they don' t need to supply the formatting codes. The control provides simple properties to turn the selected text into bold, change the alignment of the current paragraph, and so on. The **RTF** code is generated internally by the control and used to save and load formatted files. It' s possible to create elaborately formatted documents without knowing the **RTF** language.

The fundamental property of the RichTextBox control is its `Rtf` property. Similar to the `Text` property of the TextBox control, this property is the text displayed on the control. Unlike the `Text` property, however, which returns (or sets) the text of the control but doesn't contain formatting information, the `Rtf` property returns the text *along with* any formatting information. Therefore, you can use the RichTextBox control to specify the text's formatting, including paragraph indentation, font, and font size or style.

RTF, which stands for *Rich Text Format*, is a standard for storing formatting information along with the text. The beauty of the RichTextBox control for programmers is that they don't need to supply the formatting codes. The control provides simple properties to change the font of the selected text, change the alignment of the current paragraph, and so on. The RTF code is generated internally by the control and used to save and load formatted files. It's possible to create elaborately formatted documents without knowing the RTF specification.

The WordPad application that comes with Windows is based on the RichTextBox control. You can easily duplicate every bit of WordPad's functionality with the RichTextBox control, as you will see later, in the section ''VB 2008 at Work: The RTFPad Project.''

The RTF Language

A basic knowledge of the RTF format, its commands, and how it works will certainly help you understand the RichTextBox control's inner workings. RTF is a language that uses simple commands to specify the formatting of a document. These commands, or *tags*, are ASCII strings, such as `\par` (the tag that marks the beginning of a new paragraph) and `\b` (the tag that turns on the bold style). And this is where the value of the RTF format lies. RTF documents don't contain special characters and can be easily exchanged among different operating systems and computers, as long as there is an RTF-capable application to read the document. Let's look at an RTF document in action.

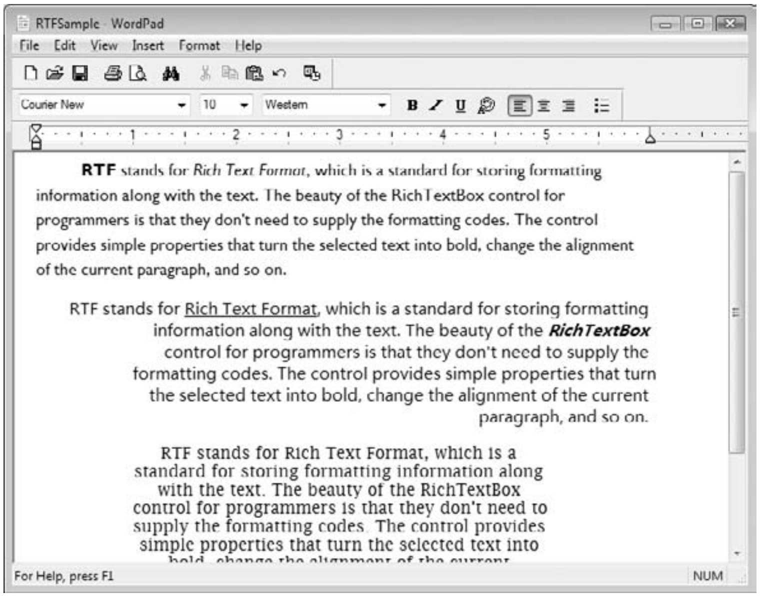
Open the WordPad application (choose Start ➤ Programs ➤ Accessories ➤ WordPad) and enter a few lines of text (see Figure 8.8). Select a few words or sentences, and format them in different ways with any of WordPad's formatting commands. Then save the document in RTF format: Choose File ➤ Save As, select Rich Text Format, and then save the file as `Document.rtf`. If you open this file with a text editor such as Notepad, you'll see the actual RTF code that produced the document. A section of the RTF file for the document shown in Figure 8.8 is shown in Listing 8.4.

Listing 8.4: The RTF Code for the First Paragraph of the Document in Figure 8.8

```
{\rtf1\ansi\ansicpg1252\deff0\deflang1033
{\fonttbl{\f0\fnil\fcharset0 Verdana;}{\f1\fswiss\fcharset0 Arial;}}
\viewkind4\uc1\pard\nowidctlpar\fi720 \b\f0\fs18 RTF
\b0 stands for \i Rich Text Format\i0 ,
which is a standard for storing formatting
information along with the text. The beauty
of the RichTextBox control for programmers
is that they don\rquote t need to supply the
formatting codes. The control provides simple
properties that turn the selected text into bold,
change the alignment of the current paragraph, and so on.\par
```

As you can see, all formatting tags are prefixed with the backslash (\) symbol. The tags are shown in bold to stand out in the listing. To display the \ symbol itself, insert an additional slash. Paragraphs are marked with the `\par` tag, and the entire document is enclosed in a pair of curly brackets. The `\li` and `\ri` tags that are followed by a numeric value specify the amount of the left and right indentation. If you assign this string to the RTF property of a RichTextBox control, the result will be the document shown in Figure 8.7, formatted exactly as it appears in WordPad.

FIGURE 8.8
The formatting applied to the text by using WordPad's commands is stored along with the text in RTF format.



RTF stands for *Rich Text Format*, which is a standard for storing formatting information along with the text. The beauty of the RichTextBox control for programmers is that they don't need to supply the formatting codes. The control provides simple properties that turn the selected text into bold, change the alignment of the current paragraph, and so on.

RTF stands for Rich Text Format, which is a standard for storing formatting information along with the text. The beauty of the ***RichTextBox*** control for programmers is that they don't need to supply the formatting codes. The control provides simple properties that turn the selected text into bold, change the alignment of the current paragraph, and so on.

RTF stands for Rich Text Format, which is a standard for storing formatting information along with the text. The beauty of the RichTextBox control for programmers is that they don't need to supply the formatting codes. The control provides simple properties that turn the selected text into

RTF is similar to Hypertext Markup Language (HTML), and if you're familiar with HTML, a few comparisons between the two standards will provide helpful hints and insight into the RTF language. Like HTML, RTF was designed to create formatted documents that could be displayed on different systems. The following RTF segment displays a sentence with a few words in italics:

```
\bRTF\b0 (which stands for Rich Text Format) is a \i
 document formatting language\i0 that uses simple
 commands to specify the formatting of the document.
```

The following is the equivalent HTML code:

```
<b>RTF</b> (which stands for Rich Text Format) is a
 <i>document formatting language</i> that uses simple
 commands to specify the formatting of the document.
```

The `<b>` and `<i>` tags of HTML, for example, are equivalent to the `\b` and `\i` tags of RTF. The closing tags in RTF are `\b0` and `\i0`, respectively.

RTF, however, is much more complicated than HTML. It's not nearly as easy to understand an RTF document as it is to understand an HTML document, because RTF was meant to be used internally by applications. As you can see in Listing 8.3, RTF contains information about the font being used, its size, and so on. Just as you need a browser to view HTML documents, you need an RTF-capable application to view RTF documents. WordPad, for instance, supports RTF and can both save a document in RTF format and read RTF files.

Although you don't need to understand the RTF specifications to produce formatted text with the RichTextBox control, if you want to generate RTF documents from within your code, visit the RTF Cookbook site at `http://search.cpan.org/~sburke/RTF-Writer/lib/RTF/Cookbook.pod`.

There's also a Microsoft resource on RTF at `http://msdn2.microsoft.com/en-us/library/aa140277(office.10).aspx`.

Text Manipulation and Formatting Properties

The RichTextBox control provides properties for manipulating the selected text on the control. The names of these properties start with the `Selection` or `Selected` prefix, and the most commonly used ones are shown in Table 8.1. Some of these properties are discussed in further detail in following sections.

TABLE 8.1: RichTextBox Properties for Manipulating Selected Text

PROPERTY	WHAT IT MANIPULATES
`SelectedText`	The selected text
`SelectedRtf`	The RTF code of the selected text
`SelectionStart`	The position of the selected text's first character
`SelectionLength`	The length of the selected text
`SelectionFont`	The font of the selected text
`SelectionColor`	The color of the selected text
`SelectionBackColor`	The background color of the selected text
`SelectionAlignment`	The alignment of the selected text
`SelectionIndent`, `SelectionRightIndent`, `SelectionHangingIndent`	The indentation of the selected text
`RightMargin`	The distance of the text's right margin from the left edge of the control
`SelectionTabs`	An array of integers that sets the tab stop positions in the control
`SelectionBullet`	Whether the selected text is bulleted
`BulletIndent`	The amount of bullet indent for the selected text

SELECTEDTEXT

The `SelectedText` property represents the selected text, whether it was selected by the user via the mouse or from within your code. To assign the selected text to a variable, use the following statement:

```
selText=RichTextbox1.SelectedText
```

You can also modify the selected text by assigning a new value to the SelectedText property. The following statement converts the selected text to uppercase:

```
RichTextbox1.SelectedText = _
            RichTextbox1.SelectedText.ToUpper
```

You can assign any string to the SelectedText property. If no text is selected at the time, the statement will insert the string at the location of the pointer.

SelectionStart, SelectionLength

To simplify the manipulation and formatting of the text on the control, two additional properties, SelectionStart and SelectionLength, report (or set) the position of the first selected character in the text and the length of the selection, respectively, regardless of the formatting of the selected text. One obvious use of these properties is to select (and highlight) some text on the control:

```
RichTextBox1.SelectionStart = 0
RichTextBox1.SelectionLength = 100
```

You can also use the Select method, which accepts as arguments the starting location and the length of the text to be selected.

SelectionAlignment

Use this property to read or change the alignment of one or more paragraphs. This property's value is one of the members of the HorizontalAlignment enumeration: Left, Right, and Center. Users don't have to select an entire paragraph to align it; just placing the pointer anywhere in the paragraph will do the trick, because you can't align part of the paragraph.

SelectionIndent, SelectionRightIndent, SelectionHangingIndent

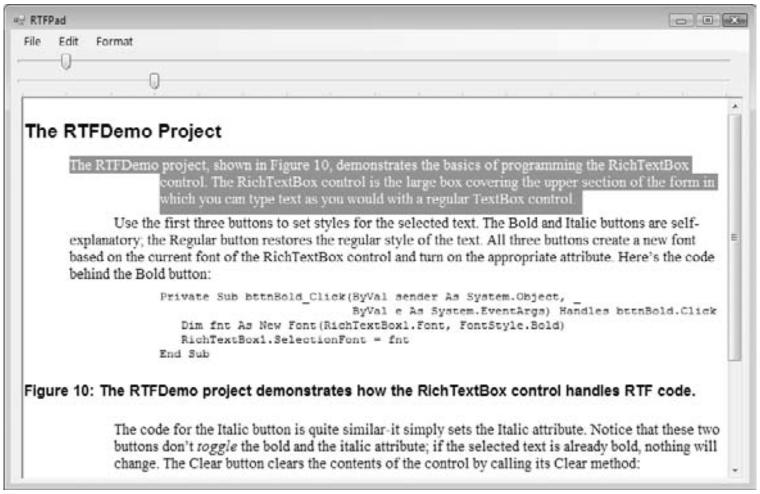
These properties allow you to change the margins of individual paragraphs. The Selection-Indent property sets (or returns) the amount of the text's indentation from the left edge of the control. The SelectionRightIndent property sets (or returns) the amount of the text's indentation from the right edge of the control. The SelectionHangingIndent property indicates the indentation of each paragraph's first line with respect to the following lines of the same paragraph. All three properties are expressed in pixels.

The SelectionHangingIndent property includes the current setting of the SelectionIndent property. If all the lines of a paragraph are aligned to the left, the SelectionIndent property can have any value (this is the distance of all lines from the left edge of the control), but the SelectionHangingIndent property must be zero. If the first line of the paragraph is shorter than the following lines, the SelectionHangingIndent has a negative value. Figure 8.9 shows several differently formatted paragraphs. The settings of the SelectionIndent and Selection-HangingIndent properties are determined by the two sliders at the top of the form.

SelectionBullet, BulletIndent

You use these properties to create a list of bulleted items. If you set the SelectionBullet property to True, the selected paragraphs are formatted with a bullet style, similar to the <ul> tag in HTML. To create a list of bulleted items, select them from within your code and assign the value True to the SelectionBullet property. To change a list of bulleted items back to normal text, make the same property False.

FIGURE 8.9
Various combinations of the `Selection-Indent` and `SelectionHanging-Indent` properties produce all possible paragraph styles.



The RTFDemo Project

The RTFDemo project, shown in Figure 10, demonstrates the basics of programming the RichTextBox control. The RichTextBox control is the large box covering the upper section of the form in which you can type text as you would with a regular TextBox control.

Use the first three buttons to set styles for the selected text. The Bold and Italic buttons are self-explanatory; the Regular button restores the regular style of the text. All three buttons create a new font based on the current font of the RichTextBox control and turn on the appropriate attribute. Here's the code behind the Bold button:

```
Private Sub bttnBold_Click(ByVal sender As System.Object, _
                       ByVal e As System.EventArgs) Handles bttnBold.Click
   Dim fnt As New Font(RichTextBox1.Font, FontStyle.Bold)
   RichTextBox1.SelectionFont = fnt
End Sub
```

Figure 10: The RTFDemo project demonstrates how the RichTextBox control handles RTF code.

The code for the Italic button is quite similar-it simply sets the Italic attribute. Notice that these two buttons don't *toggle* the bold and the italic attribute; if the selected text is already bold, nothing will change. The Clear button clears the contents of the control by calling its Clear method:

The paragraphs formatted as bullets are also indented from the left by a small amount. To set the amount of the indentation, use the `BulletIndent` property, which is also expressed in pixels.

SelectionTabs

Use this property to set the tab stops in the RichTextBox control. The Selection tab should be set to an array of integer values, which are the absolute tab positions in pixels. Use this property to set up a RichTextBox control for displaying tab-delimited data.

Using the RichTextBox Control to Display Delimited Data

As a developer I tend to favor the RichTextBox control over the TextBox control, even though I don't mix font styles or use the more-advanced features of the RichTextBox control. I suggest that you treat the RichTextBox control as an enhanced TextBox control and use it as a substitute for the TextBox control. One of the features of the RichTextBox control that I find very handy is its ability to set the tab positions and display tabular data. You can also display tabular data on a ListView control, as you will see in the following chapter, but it's simpler to use a RichTextBox control with its `ReadOnly` property set to True and its `SelectionTabs` property to an array of values that will accommodate your data. Here's how to set up a RichTextBox control to display a few rows of tab-delimited data:

```
RichTextBox1.ReadOnly = True
RichTextBox1.SelectionTabs = New Integer() {100, 160, 340}
RichTextBox1.AppendText("R1C1" & vbTab & _
             "R1C2" & vbTab & _
             "R1C3" & vbCrLf)
RichTextBox1.AppendText("R2C1" & vbTab & _
             "R2C2" & vbTab & _
             "R2C3" & vbCrLf)
```

This technique is a life-saver when I have to read the delimited data from a file. I just set up the tab positions and then load the data with the `LoadFile` method, which is discussed in the following section.

Methods

The first two methods of the RichTextBox control you need to know are `SaveFile` and `LoadFile`. The `SaveFile` method saves the contents of the control to a disk file, and the `LoadFile` method loads the control from a disk file.

SaveFile

The syntax of the `SaveFile` method is as follows:

```
RichTextBox1.SaveFile(path, filetype)
```

where *`path`* is the path of the file in which the current document will be saved. By default, the `SaveFile` method saves the document in RTF format and uses the `.RTF` extension. You can specify a different format by using the second optional argument, which can take on the value of one of the members of the `RichTextBoxStreamType` enumeration, described in Table 8.2.

Table 8.2: The *RichTextBoxStreamType* Enumeration

Format	Effect
`PlainText`	Stores the text on the control without any formatting
`RichNoOLEObjs`	Stores the text without any formatting and ignores any embedded OLE objects
`RichText`	Stores the text in RTF format (text with embedded RTF commands)
`TextTextOLEObjs`	Stores the text along with the embedded OLE objects
`UnicodePlainText`	Stores the text in Unicode format

LoadFile

Similarly, the `LoadFile` method loads a text or RTF file to the control. Its syntax is identical to the syntax of the `SaveFile` method:

```
RichTextBox1.LoadFile(path, filetype)
```

The *`filetype`* argument is optional and can have one of the values of the `RichTextBoxStream-Type` enumeration. Saving and loading files to and from disk files is as simple as presenting a Save or Open common dialog to the user and then calling one of the `SaveFile` or `LoadFile` methods with the filename returned by the common dialog box.

Select, SelectAll

The `Select` method selects a section of the text on the control, similar to setting the `Selection-Start` and `SelectionLength` properties. The `Select` method accepts two arguments: the location of the first character to be selected and the length of the selection:

```
RichTextBox1.Select(start, length)
```

The `SelectAll` method accepts no arguments and it selects all the text on the control.

Advanced Editing Features

The RichTextBox control provides all the text-editing features you'd expect to find in a text-editing application, similar to the TextBox control. Among its more-advanced features, the RichTextBox control provides the `AutoWordSelection` property, which controls how the control selects text. If it's True, the control selects a word at a time.

In addition to formatted text, the RichTextBox control can handle object linking and embedding (OLE) objects. You can insert images in the text by pasting them with the `Paste` method. The `Paste` method doesn't require any arguments; it simply inserts the contents of the Clipboard at the current location in the document.

The RichTextBox control encapsulates undo and redo operations at multiple levels. Each operation has a name (Typing, Deletion, and so on), and you can retrieve the name of the next operation to be undone or redone and display it on the menu. Instead of a simple Undo or Redo caption, you can change the captions of the Edit menu to something like Undo Delete or Redo Typing. To program undo and redo operations from within your code, you must use the properties and methods discussed in the following sections.

CanUndo, CanRedo

These two properties are Boolean values you can read to find out whether there's an operation that can be undone or redone. If they're False, you must disable the corresponding menu command from within your code. The following statements disable the Undo command if there's no action to be undone at the time (`EditUndo` is the name of the Undo command on the Edit menu):

```
If RichTextBox1.CanUndo Then
   EditUndo.Enabled = True
Else
   EditUndo.Enabled = False
End If
```

These statements should appear in the menu item's `Select` event handler (not in the `Click` event handler) because they must be executed before the menu is displayed. The `Select` event is triggered when a menu is opened. As a reminder, the `Click` event is fired when you click an item, and not when you open a menu. For more information on programming the events of a menu, see Chapter 7, ''Working with Forms.''

UndoActionName, RedoActionName

These two properties return the name of the action that can be undone or redone. The most common value of both properties is *`Typing`*, which indicates that the Undo command will delete a number of characters. Another common value is *`Delete`*, whereas some operations are named *`Unknown`*. If you change the indentation of a paragraph on the control, this action's name is *`Unknown`*. Even when an action's name is `Unknown`, the action can be undone with the `Undo` method.

The following statement sets the caption of the Undo command to a string that indicates the action to be undone (`Editor` is the name of a RichTextBox control):

```
If Editor.CanUndo Then
    EditUndo.Text = "Undo " & Editor.UndoActionName
End If
```

Undo, Redo

These two methods undo or redo an action. The `Undo` method cancels the effects of the last action of the user on the control. The `Redo` method redoes the most recent undo action. The `Redo` method does not repeat the last action; it applies to undo operations only.

Cutting and Pasting

To cut, copy, and paste text in the RichTextBox control, you can use the same techniques you use with the regular TextBox control. For example, you can replace the current selection by assigning a string to the `SelectedText` property. The RichTextBox, however, provides a few useful methods for performing these operations. The `Copy`, `Cut`, and `Paste` methods perform the corresponding operations. The `Cut` and `Copy` methods are straightforward and require no arguments. The `Paste` method accepts a single argument, which is the format of the data to be pasted. Because the data will come from the Clipboard, you can extract the format of the data in the Clipboard at the time and then call the `CanPaste` method to find out whether the control can handle this type of data. If so, you can then paste them in the control by using the `Paste` method.

This technique requires a bit of code because the Clipboard class doesn't return the format of the data in the Clipboard. You must call the following method of the Clipboard class to find out whether the data is of a specific type and then paste it on the control:

```
If Clipboard.GetDataObject. _
                GetDataPresent(DataFormats.Text) Then
   RichTextBox.Paste(DataFormats.Text)
End If
```

This is a very simple case because we know that the RichTextBox control can accept text. For a robust application, you must call the `GetDataPresent` method for each type of data your application should be able to handle. (You may not want to allow users to paste all types of data that the control can handle.) By the way, you can simplify the code with the help of the `ContainsText/ContainsImage` and `GetText/GetImage` methods of the `My.Application.Clipboard` object.

In the RTFPad project later in this chapter, we'll use a structured exception handler to allow users to paste anything in the control. If the control can't handle it, the data won't be pasted in the control.

Searching in a RichTextBox Control

To locate a string in the text of the RichTextBox control, use the `Find` method. The `Find` method is quite flexible, as it allows you to specify the type of the search, whether it will locate entire words, and so on. The simplest form of this method accepts the search string as an argument and returns the location of the first instance of the word in the text. If the search argument isn't found, the method returns the value −1.

```
RichTextBox1.Find(string)
```

Another equally simple syntax of the `Find` method allows you to specify how the control will search for the string:

```
RichTextBox1.Find(string, searchMode)
```

The *searchMode* argument is a member of the RichTextBoxFinds enumeration, which is shown in Table 8.3.

TABLE 8.3: The *RichTextBoxFinds* Enumeration

VALUE	EFFECT
MatchCase	Performs a case-sensitive search.
NoHighlight	The text found will not be highlighted.
None	Locates instances of the specified string even if they're not whole words.
Reverse	The search starts at the end of the document.
WholeWord	Locates only instances of the specified string that are whole words.

Two more forms of the Find method allow you specify the range of the text in which the search will take place:

```
RichTextBox1.Find(string, start, searchMode)
RichTextBox1.Find(string, start, end, searchMode)
```

The arguments *start* and *end* are the starting and ending locations of the search (use them to search for a string within a specified range only). If you omit the *end* argument, the search will start at the location specified by the *start* argument and will extend to the end of the text.

You can combine multiple values of the *searchMode* argument with the OR operator. The default search is case-insensitive, covers the entire document, and highlights the matching text on the control. The RTFPad application's Find command demonstrates how to use the Find method and its arguments to build a Search & Replace dialog box that performs all the types of text-searching operations you might need in a text-editing application.

Handling URLs in the Document

An interesting feature of the RichTextBox control is the automatic formatting of URLs embedded in the text. To enable this feature, set the DetectURLs property to True. Then, as soon as the control determines that you're entering a URL (usually after you enter the three *w*'s and the following period), it will format the text as a hyperlink. When the pointer rests over a hyperlink, its shape turns into a hand, just as it would in Internet Explorer. Run the RTFDemo project, enter a URL such as http://www.sybex.com, and see how the RichTextBox control handles it.

In addition to formatting the URL, the RichTextBox control triggers the LinkClicked event when a hyperlink is clicked. To display the corresponding page from within your code, enter the following statement in the LinkClicked event handler:

```
Private Sub RichTextBox1_LinkClicked( _
     ByVal sender As Object, _
     ByVal e As System.Windows.Forms.LinkClickedEventArgs) _
     Handles RichTextBox1.LinkClicked
   System.Diagnostics.Process.Start(e.LinkText)
End Sub
```

The System.Diagnostics.Process class provides the `Start` method, which starts an application. You can specify either the name of the executable or the path of a file. If you specify, the `Start` method will look up the associated application and start it. As you can see, handling embedded URLs with the RichTextBox control is almost trivial.

Displaying a Formatted Directory Listing

This is a good point to review the subroutines that produced the formatted directory listings shown in Figure 8.6. Folder names are printed in bold by the `PrintFolderName()` subroutine, and filenames are printed in regular style by the `PrintFileNames()` subroutine. Both subroutines accept as arguments a DirectoryInfo object that represents the folder whose name (or files) we want to print, as well as an indentation string. This string is increased every time the code drills down to a subfolder and is decreased every time it moves up to a parent folder. Listing 8.5 shows the implementation of the two subroutines.

LISTING 8.5: The *PrintFolderName()* and *PrintFileNames()* Subroutines

```
Private Sub PrintFolderName( _
             ByVal folder As IO.DirectoryInfo, _
             ByVal Indentation As String)
    SwitchToBold()
    RichTextBox1.AppendText(Indentation)
    RichTextBox1.AppendText(folder.Name & vbCrLf)
    SwitchToRegular()
End Sub

Private Sub PrintFileNames( _
              ByVal folder As IO.DirectoryInfo, _
              ByVal indentation As String)
    Dim file As IO.FileInfo
    For Each file In folder.GetFiles("*.*")
        RichTextBox1.AppendText( _
                 indentation & file.Name & vbCrLf)
    Next
End Sub
```

The code for printing folder names and filenames is trivial. Before calling the `AppendText` method to add a new folder name to the control, the code calls the `SwitchToBold()` subroutine. After printing the folder name, it calls the `SwitchToRegular` subroutine to reset the font. The two subroutines manipulate the `SelectionFont` property. Because no text is selected at the time, the subroutines simply change the attributes of the text that will be appended to the control with the next call to the `AppendText` method. The implementation of the two subroutines is shown next:

```
Private Sub SwitchToItalics()
    RichTextBox1.SelectionFont = _
        New Font(RichTextBox1.SelectionFont.Name, _
        RichTextBox1.SelectionFont.Size, FontStyle.Italic)
End Sub
```

```
Private Sub SwitchToRegular()
    RichTextBox1.SelectionFont = _
        New Font(RichTextBox1.SelectionFont.Name, _
            RichTextBox1.SelectionFont.Size, FontStyle.Regular)
End Sub
```

VB 2008 at Work: The RTFPad Project

Creating a functional — even fancy — word processor based on the RichTextBox control is unexpectedly simple. The challenge is to provide a convenient interface that lets the user select text, apply attributes and styles to it, and then set the control's properties accordingly. The RTFPad sample application of this section does just that.

The RTFPad application (refer to Figure 8.7) is based on the TextPad application developed in Chapter 6, ''Basic Windows Controls.'' It contains the same text-editing commands and some additional text-formatting commands that can be implemented only with the RichTextBox control; for example, it allows you to apply multiple fonts and styles to the text, and, of course, multiple Undo/Redo operations.

The two TrackBar controls above the RichTextBox control manipulate the indentation of the text. We already explored this arrangement in the discussion of the TrackBar control in Chapter 6, but let's review the operation of the two controls again. Each TrackBar control has a width of 816 pixels, which is equivalent to 8.5 inches on a monitor that has a resolution of 96 dots per inch (dpi). The height of the TrackBar controls is 42 pixels, but unfortunately they can't be made smaller. The `Minimum` property of both controls is 0, and the `Maximum` property is 16. The `TickFrequency` is 1. With these values, you can adjust the indentation in steps of 1/2 inch. Set the `Maximum` property to 32 and you'll be able to adjust the indentation in steps of 1/4 inch. It's not the perfect interface, as it's built for A4 pages in portrait orientation only. You can experiment with this interface to build an even more functional word processor.

Each time the user slides the top TrackBar control, the code sets the `SelectionIndent` property to the proper percentage of the control's width. Because the `SelectionHangingIndent` includes the value of the `SelectionIndent` property, it also adjusts the setting of the `SelectionHangingIndent` property. Listing 8.6 is the code that's executed when the upper TrackBar control is scrolled.

Listing 8.6: Setting the *SelectionIndent* Property

```
Private Sub TrackBar1_Scroll(...) _
            Handles TrackBar1.Scroll
    Editor.SelectionIndent = Convert.ToInt32( _
               Editor.Width * _
              (TrackBar1.Value / TrackBar1.Maximum))
    Editor.SelectionHangingIndent = Convert.ToInt32( _
               Editor.Width * _
              (TrackBar2.Value / TrackBar2.Maximum) - Editor.SelectionIndent)
End Sub
```

`Editor` is the name of the RichTextBox control on the form. The code sets the control's indentation to the same percentage of the control's width, as indicated by the value of the top TrackBar control. It also does the same for the `SelectionHangingIndent` property, which is controlled by the lower TrackBar control. If the user has scrolled the lower TrackBar control, the code sets the RichTextBox control's `SelectionHangingIndent` property in the event handler, as presented in Listing 8.7.

Listing 8.7: Setting the *SelectionHangingIndent* Property

```
Private Sub TrackBar2_Scroll(...) _
            Handles TrackBar2.Scroll
    Editor.SelectionHangingIndent = _
             Convert.ToInt32(Editor.Width * _
             (TrackBar2.Value / TrackBar2.Maximum) - _
             Editor.SelectionIndent)
End Sub
```

Enter a few lines of text in the control, select one or more paragraphs, and check out the operation of the two sliders.

The `Scroll` events of the two TrackBar controls adjust the text's indentation. The opposite action must take place when the user rests the pointer on another paragraph: The sliders' positions must be adjusted to reflect the indentation of the selected paragraph. The selection of a new paragraph is signaled to the application by the `SelectionChanged` event. The statements of Listing 8.8, which are executed from within the `SelectionChanged` event, adjust the two slider controls to reflect the indentation of the text.

Listing 8.8: Setting the Slider Controls

```
Private Sub Editor_SelectionChanged(...) _
Handles Editor.SelectionChanged
    If Editor.SelectionIndent = Nothing Then
        TrackBar1.Value = TrackBar1.Minimum
        TrackBar2.Value = TrackBar2.Minimum
    Else
        TrackBar1.Value = Convert.ToInt32( _
                  Editor.SelectionIndent * _
                  TrackBar1.Maximum / Editor.Width)
        TrackBar2.Value = Convert.ToInt32( _
                  (Editor.SelectionHangingIndent / _
                   Editor.Width) * _
                   TrackBar2.Maximum + TrackBar1.Value)
    End If
End Sub
```

If the user selects multiple paragraphs with different indentations, the `SelectionIndent` property returns Nothing. The code examines the value of this property and, if it's Nothing, it moves both controls to the left edge. This way, the user can slide the controls and set the indentations for multiple paragraphs. Some applications make the handles gray to indicate that the selected text doesn't have uniform indentation, but unfortunately you can't gray the sliders and keep them enabled. Of course, you can always design a custom control. This wouldn't be a bad idea, especially if you consider that the TrackBar controls are too tall for this type of interface and can't be made very narrow (as a result, the interface of the RTFPad application isn't very elegant).

The File Menu

The RTFPad application's File menu contains the usual Open, Save, and Save As commands, which are implemented with the control's `LoadFile` and `SaveFile` methods. Listing 8.9 shows the implementation of the Open command in the File menu.

Listing 8.9: The Open Command

```
Private Sub OpenToolStripMenuItem_Click(...) _
            Handles OpenToolStripMenuItem.Click
    If DiscardChanges() Then
        OpenFileDialog1.Filter = _
               "RTF Files|*.RTF|DOC Files|*.DOC|" & _
               "Text Files|*.TXT|All Files|*.*"
        If OpenFileDialog1.ShowDialog() = _
                       DialogResult.OK Then
            fName = OpenFileDialog1.FileName
            Editor.LoadFile(fName)
            Editor.Modified = False
        End If
    End If
End Sub
```

The `fName` variable is declared on the form's level and holds the name of the currently open file. This variable is set every time a new file is successfully opened and it's used by the Save command to automatically save the open file, without prompting the user for a filename.

`DiscardChanges()` is a function that returns a Boolean value, depending on whether the control's contents can be discarded. The function examines the Editor control's `Modified` property. If True, it prompts users as to whether they want to discard the edits. Depending on the value of the `Modified` property and the user response, the function returns a Boolean value. If the `DiscardChanges()` function returns True, the program goes on and opens a new document. If the function returns False, the program aborts the operation to give the user a chance to save the document. Listing 8.10 shows the `DiscardChanges()` function.

Listing 8.10: The *DiscardChanges()* Function

```
Function DiscardChanges() As Boolean
   If Editor.Modified Then
      Dim reply As MsgBoxResult
      reply = MsgBox( _
          "Text hasn't been saved. Discard changes?", _
           MsgBoxStyle.YesNo)
```

```
            If reply = MsgBoxResult.No Then
                Return False
            Else
                Return True
            End If
        Else
            Return True
        End If
    End Function
```

The `Modified` property becomes True after typing the first character and isn't reset back to False. The RichTextBox control doesn't handle this property very intelligently and doesn't reset it to False even after saving the control's contents to a file. The application's code sets the `Editor.Modified` property to False after creating a new document, as well as after saving the current document.

The Save As command (see Listing 8.11) prompts the user for a filename and then stores the `Editor` control's contents to the specified file. It also sets the `fName` variable to the file's path, so that the Save command can use it.

Listing 8.11: The Save As Command

```
    Private Sub SaveAsToolStripMenuItem_Click(...) _
                Handles SaveAsToolStripMenuItem.Click
        SaveFileDialog1.Filter = _
                    "RTF Files|*.RTF|DOC Files" & _
                    "|*.DOC|Text Files|*.TXT|All Files|*.*"
        SaveFileDialog1.DefaultExt = "RTF"
        If SaveFileDialog1.ShowDialog() = DialogResult.OK Then
            fName = SaveFileDialog1.FileName
            Editor.SaveFile(fName)
            Editor.Modified = False
        End If
    End Sub
```

The Save command's code is similar, only it doesn't prompt the user for a filename. It calls the `SaveFile` method, passing the `fName` variable as an argument. If the `fName` variable has no value (in other words, if a user attempts to save a new document by using the Save command), the code activates the event handler of the Save As command automatically and resets the control's `Modified` property to False. Listing 8.12 shows the code behind the Save command.

Listing 8.12: The Save Command

```
    Private Sub SaveToolStripMenuItem_Click(...) _
                Handles SaveToolStripMenuItem.Click
        If fName <> "" Then
            Editor.SaveFile(fName)
            Editor.Modified = False
        Else
            SaveAsToolStripMenuItem_Click(sender, e)
        End If
    End Sub
```

The Edit Menu

The Edit menu contains the usual commands for exchanging data through the Clipboard (Copy, Cut, Paste), Undo/Redo commands, and a Find command to invoke the Search & Replace dialog box. All the commands are almost trivial, thanks to the functionality built into the control. The basic Cut, Copy, and Paste commands call the RichTextBox control's `Copy`, `Cut`, and `Paste` methods to exchange data through the Clipboard. Listing 8.13 shows the implementation of the Paste command.

Listing 8.13: The Paste Command

```
    Private Sub PasteToolStripMenuItem_Click(...) _
                Handles PasteToolStripMenuItem.Click
        Try
            Editor.Paste()
        Catch exc As Exception
            MsgBox( _
                "Can't paste current clipboard's contents")
        End Try
    End Sub
```

As you may recall from the discussion of the Paste command, we can't use the `CanPaste` method because it's not trivial; you have to handle each data type differently. By using an exception handler, we allow the user to paste all types of data that the RichTextBox control can accept, and display a message when an error occurs.

The Undo and Redo commands of the Edit menu are coded as follows. First, we display the name of the action to be undone or redone in the Edit menu. When the Edit menu is selected, the `DropDownOpened` event is fired. This event takes place before the `Click` event, so I inserted a few lines of code that read the name of the most recent action that can be undone or redone and print it next to the Undo or Redo command's caption. If there's no such action, the program will disable the corresponding command. Listing 8.14 is the code that's executed when the Edit menu is dropped.

Listing 8.14: Setting the Captions of the Undo and Redo Commands

```
    Private Sub EditToolStripMenuItem_DropDownOpened(...)_
            Handles EditToolStripMenuItem.DropDownOpened
        If Editor.UndoActionName <> "" Then
            UndoToolStripMenuItem.Text = _
                         "Undo " & Editor.UndoActionName
            UndoToolStripMenuItem.Enabled = True
        Else
            UndoToolStripMenuItem.Text = "Undo"
            UndoToolStripMenuItem.Enabled = False
        End If
        If Editor.RedoActionName <> "" Then
            RedoToolStripMenuItem.Text = _
                          "Redo" & Editor.RedoActionName
            RedoToolStripMenuItem.Enabled = True
        Else
            RedoToolStripMenuItem.Text = "Redo"
            RedoToolStripMenuItem.Enabled = False
        End If
    End Sub
```

When the user selects one of the Undo or Redo commands, the code simply calls the appropriate method from within the menu item's `Click` event handler, as shown in Listing 8.15.

LISTING 8.15: Undoing and Redoing Actions

```
    Private Sub RedoToolStripMenuItem_Click(...) _
                  Handles RedoToolStripMenuItem.Click
        If Editor.CanRedo Then Editor().Redo()
    End Sub

    Private Sub UndoToolStripMenuItem_Click(...) _
                  Handles UndoToolStripMenuItem.Click
        If Editor.CanUndo Then Editor.Undo()
    End Sub
```

Calling the `CanUndo` and `CanRedo` method is unnecessary; if the corresponding action can't be performed, the two menu items will be disabled, but an additional check does no harm.

THE FORMAT MENU

The commands of the Format menu control the alignment and the font attributes of the current selection. The Font command displays the Font dialog box and then assigns the font selected by the user to the current selection. Listing 8.16 shows the code behind the Font command.

LISTING 8.16: The Font Command

```
    Private Sub FontToolStripMenuItem_Click(...) _
                    Handles FontToolStripMenuItem.Click
        If Not Editor.SelectionFont Is Nothing Then
            FontDialog1.Font = Editor.SelectionFont
        Else
```

```
            FontDialog1.Font = Nothing
        End If
        FontDialog1.ShowApply = True
        If FontDialog1.ShowDialog() = DialogResult.OK Then
            Editor.SelectionFont = FontDialog1.Font
        End If
    End Sub
```

Notice that the code preselects a font in the dialog box, which is the font of the current selection. If the current selection isn't formatted with a single font, no font is preselected.

To enable the Apply button of the Font dialog box, set the control's `ShowApply` property to True and insert the following statement in its `Apply` event handler:

```
Private Sub FontDialog1_Apply( _
                ByVal sender As Object, _
                ByVal e As System.EventArgs) _
                Handles FontDialog1.Apply
    Editor.SelectionFont = FontDialog1.Font
End Sub
```

The options of the Align menu set the RichTextBox control's `SelectionAlignment` property to different members of the `HorizontalAlignment` enumeration. The Align ➢ Left command, for example, is implemented with the following statement:

```
Editor.SelectionAlignment = HorizontalAlignment.Left
```

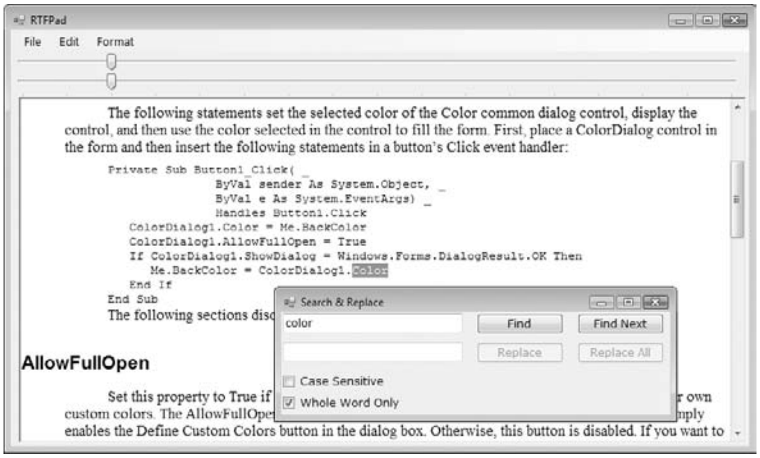
The Search & Replace Dialog Box

The Find command in the Edit menu opens the dialog box shown in Figure 8.10, which performs search-and-replace operations (whole-word or case-sensitive match, or both). The Search & Replace form (it's the `frmFind` form in the project) has its `TopMost` property set to True, so that it remains visible while it's open, even if it doesn't have the focus. The code behind the buttons on this form is quite similar to the code for the Search & Replace dialog box of the TextPad application, with one basic difference: the RTFPad project's code uses the RichTextBox control's `Find` method; the simple TextBox control doesn't provide an equivalent method and we had to use the methods of the String class to perform the same operations. The `Find` method of the RichTextBox control performs all types of searches, and some of its options are not available with the `IndexOf` method of the String class.

To invoke the Search & Replace dialog box, the code calls the `Show` method of the `frmFind` form, as discussed in Chapter 6, via the following statement:

```
frmFind.Show()
```

The `Find` method of the RichTextBox control allows you to perform case-sensitive or -insensitive searches, as well as search for whole words only. These options are specified through an argument of the `RichTextBoxFinds` type. The `SetSearchMode()` function (see Listing 8.17) examines the settings of the two check boxes at the bottom of the form and sets the `Find` method's search mode.



The following statements set the selected color of the Color common dialog control, display the control, and then use the color selected in the control to fill the form. First, place a ColorDialog control in the form and then insert the following statements in a button's Click event handler:

```
Private Sub Button1_Click( _
                 ByVal sender As System.Object, _
                 ByVal e As System.EventArgs) _
                 Handles Button1.Click
   ColorDialog1.Color = Me.BackColor
   ColorDialog1.AllowFullOpen = True
   If ColorDialog1.ShowDialog = Windows.Forms.DialogResult.OK Then
      Me.BackColor = ColorDialog1.Color
   End If
End Sub
```

The following sections disc

AllowFullOpen

Set this property to True if
custom colors. The AllowFullOpe
enables the Define Custom Colors button in the dialog box. Otherwise, this button is disabled. If you want to

r own
mply

color

☐ Case Sensitive
☑ Whole Word Only

FIGURE 8.10
The Search & Replace dialog box of the RTFPad application

LISTING 8.17: Setting the Search Options

```
Function SetSearchMode() As RichTextBoxFinds
    Dim mode As RichTextBoxFinds = _
                 RichTextBoxFinds.None
    If chkCase.Checked = True Then
        mode = mode Or RichTextBoxFinds.MatchCase
    End If
    If chkWord.Checked = True Then
        mode = mode Or RichTextBoxFinds.WholeWord
    End If
    Return mode
End Function
```

The `Click` event handlers of the Find and Find Next buttons call this function to retrieve the constant that determines the type of search specified by the user on the form. This value is then passed to the `Find` method. Listing 8.18 shows the code behind the Find and Find Next buttons.

LISTING 8.18: The Find and Find Next Commands

```
Private Sub bttnFind_Click(...) _
            Handles bttnFind.Click
   Dim wordAt As Integer
   Dim srchMode As RichTextBoxFinds
   srchMode = SetSearchMode()
   wordAt = frmEditor.Editor.Find( _
                    txtSearchWord.Text, 0, srchMode)
   If wordAt = -1 Then
      MsgBox("Can't find word")
      Exit Sub
   End If
   frmEditor.Editor.Select(wordAt, _
                  txtSearchWord.Text.Length)
   bttnFindNext.Enabled = True
   bttnReplace.Enabled = True
   bttnReplaceAll.Enabled = True
   frmEditor.Editor.ScrollToCaret()
End Sub

Private Sub bttnFindNext_Click(...) _
            Handles bttnFindNext.Click
   Dim selStart As Integer
   Dim srchMode As CompareMethod
   srchMode = SetSearchMode()
   selStart = frmEditor.Editor.Find( _
                txtSearchWord.Text, _
                frmEditor.Editor.SelectionStart + 2, _
                srchMode)
   If selStart = -1 Then
      MsgBox("No more matches")
      Exit Sub
   End If
   frmEditor.Editor.Select( _
                selStart, txtSearchWord.Text.Length)
   frmEditor.Editor.ScrollToCaret()
End Sub
```

Notice that both event handlers call the `ScrollToCaret` method to force the selected text to become visible — should the `Find` method locate the desired string outside the visible segment of the text.

The Bottom Line

Use the OpenFileDialog and SaveFileDialog controls to prompt users for filenames. Windows applications use certain controls to prompt users for common information, such as filenames, colors, and fonts. Visual Studio provides a set of controls, which are grouped in the Dialogs section of the Toolbox. All common dialog controls provide a `ShowDialog` method, which displays the corresponding dialog box in a modal way. The `ShowDialog` method returns a value of the `DialogResult` type, which indicates how the dialog box was closed, and you should examine this value before processing the data.

Master It Your application needs to open an existing file. How will you prompt users for the file's name?

Master It You're developing an application that encrypts multiple files (or resizes many images) in batch mode. How will you prompt the user for the files to be processed?

Use the ColorDialog and FontDialog controls to prompt users for colors and typefaces. The Color and Font dialog boxes allow you to prompt users for a color value and a font, respectively. Before showing the corresponding dialog box, set its `Color` or `Font` property according to the current selection, and then call the control's `ShowDialog` method.

> **Master It** How will you display color attributes in the Color dialog box when you open it? How will you display the attributes of the selected text's font in the Font dialog box when you open it?

Use the RichTextBox control as an advanced text editor to present richly formatted text. The RichTextBox control is an enhanced TextBox control that can display multiple fonts and styles, format paragraphs with different styles, and provide a few more advanced text-editing features. Even if you don't need the formatting features of this control, you can use it as an alternative to the TextBox control. At the very least, the RichTextBox control provides more editing features, a more-useful undo function, and more-flexible search features.

> **Master It** You want to display a document with a title in large, bold type, followed by a couple of items in regular style. How will you create a document like the following one on a RichTextBox control?

Document's Title
  Item 1
    Description for item 1
  Item 2
    Description for item 2

Chapter 9

The TreeView and ListView Controls

In Chapter 6, ''Basic Windows Controls,'' you learned how to use the ListBox control for displaying lists of strings and storing objects. The items of a ListBox control can be sorted, but they have no particular structure. I'm sure most of you wish that the ListBox control had more ''features,'' such as the means to store additional information along with each item or to present hierarchical lists. A hierarchical list is a tree that reflects the structure of the list: items that belong to other items appear under their parent with the proper indentation. For instance, a list of city and state names should be structured so that each city appears under the corresponding state.

The answer to the shortcomings of the ListBox control can be found in the TreeView and ListView controls. These two Windows controls are among the more-advanced ones, and they are certainly more difficult to program than the ones discussed in the preceding chapters. These two controls, however, are the basic makings of unique user interfaces, as you'll see in this chapter's examples. The TreeView and ListView controls implement two of the more-advanced data structures and were designed to hide much of the complexity of these structures — and they do this very well.

In this chapter, you'll learn how to do the following:

- Create and present hierarchical lists by using the TreeView control
- Create and present lists of structured items by using the ListView control

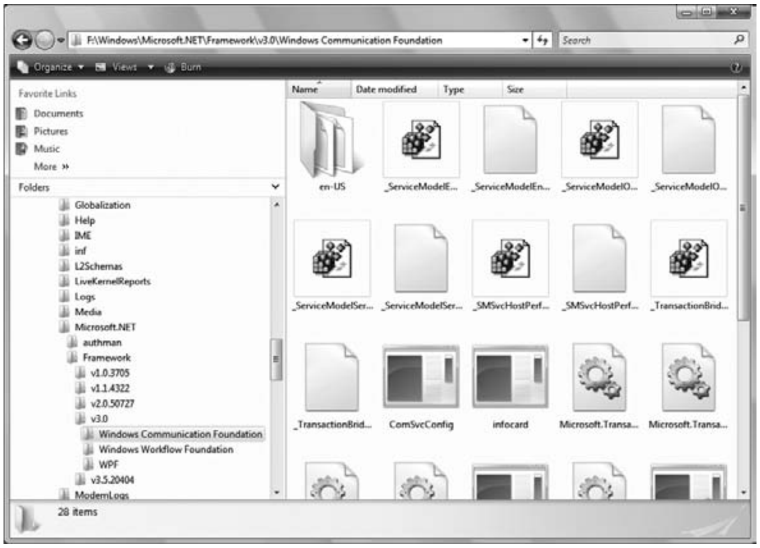
Understanding the ListView, TreeView, and ImageList Controls

I will start with a general discussion of the two controls to help you understand what they do and when to use them. A basic understanding of the data structures they implement is also required to use them efficiently in your applications. Then I'll discuss their members and demonstrate how to use the controls. If you find the examples too difficult to understand, you can always postpone the use of these controls in your applications.

Some of the code I present in this chapter can be used as is in many situations, so you should look at the examples and see whether you can incorporate some of their code in your applications. The ListView and TreeView controls are excellent tools for designing elaborate Windows interfaces, and I feel they deserve to be covered in detail. It's also common to use the ImageList control in conjunction with the ListView and TreeView controls. The purpose of the ImageList control is to store the images that we want to display, along with the items of the other two controls, so I'll discuss briefly the ImageList control in this chapter.

Figure 9.1 shows the TreeView and ListView controls used in tandem. What you see in Figure 9.1 is Windows Explorer, a utility for examining and navigating your hard disk's structure. The left pane, where the folders are displayed, is a TreeView control. The folder names are displayed in a manner that reflects their structure on the hard disk. You can expand and contract certain branches and view only the segment(s) of the tree structure you're interested in.

FIGURE 9.1
Windows Explorer is made up of a TreeView (left pane) and a ListView (right pane) control.



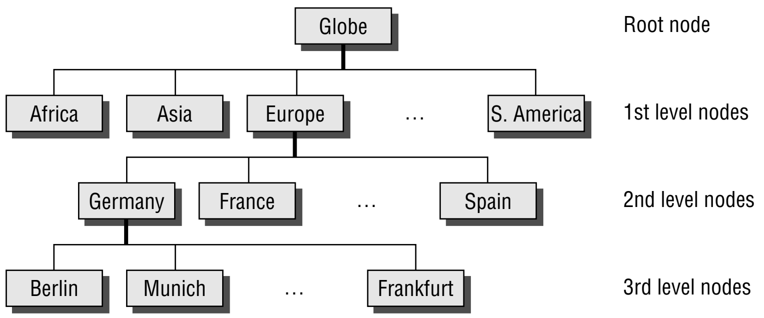
The right pane is a ListView control. The items on the ListView control can be displayed in five ways (as large or small icons, as a list, on a grid, or tiled). They are the various views you can set through the View menu of Windows Explorer. Although most people prefer to look at the contents of the folders as icons, the most common view is the Details view, which displays not only filenames, but also their attributes. In the Details view, the list can be sorted according to any of its columns, making it easy for the user to locate any item based on various criteria (file type, size, creation date, and so on). A Windows Explorer window with a detailed view of the files is shown later in this chapter, in Figure 9.4.

Tree and List Structures

The TreeView control implements a data structure known as a *tree*. A tree is the most appropriate structure for storing hierarchical information. The organizational chart of a company, for example, is a tree structure. Every person reports to another person above him or her, all the way to the president or CEO. Figure 9.2 depicts a possible organization of continents, countries, and cities as a tree. Every city belongs to a country, and every country to a continent. In the same way, every computer file belongs to a folder that may belong to an even bigger folder, and so on up to the drive level. You can't draw large tree structures on paper, but it's possible to create a similar structure in the computer's memory without size limitations.

Each item in the tree of Figure 9.2 is called a *node*, and nodes can be nested to any level. Oddly, the top node is the *root* of the tree, and the subordinate nodes are called *child nodes*. If you try to visualize this structure as a real tree, think of it as an upside-down tree with the branches emerging from the root. The end nodes, which don't lead to any other nodes, are called *leaf nodes* or *end nodes*.

FIGURE 9.2
The world viewed as a tree



To locate a city, you must start at the root node and select the continent to which the city belongs. Then you must find the country (in the selected continent) to which the city belongs. Finally, you can find the city you're looking for. If it's not under the appropriate country node, it doesn't exist.

TREEVIEW ITEMS ARE JUST STRINGS

The items displayed on a TreeView control are just strings. Moreover, the TreeView control doesn't require that the items be unique. You can have identically named nodes in the same branch — as unlikely as this might be for a real application. There's no property that makes a node unique in the tree structure or even in its own branch.

You can also start with a city and find its country. The country node is the city node's *parent node*. Notice that there is only one route from child nodes to their parent nodes, which means that you can instantly locate the country or continent of a city. The data of Figure 9.2 is shown in Figure 9.3 in a TreeView control. Only the nodes we're interested in are expanded. The plus sign indicates that the corresponding node contains child nodes. To view them, click the button with the plus sign to expand the node.

The tree structure is ideal for data with parent-child relations (relations that can be described as *belongs to* or *owns*). The continents-countries-cities data is a typical example. The folder structure on a hard disk is another typical example. Any given folder is the child of another folder or the root folder.

Many programs are based on tree structures. Computerized board games use a tree structure to store all possible positions. Every time the computer has to make a move, it locates the board's status on the tree and selects the ''best'' next move. For instance, in tic-tac-toe, the tree structure that represents the moves in the game has nine nodes on the first level, which correspond to all the possible positions for the first token on the board (the *X* or *O* mark). Under each possible initial position, there are eight nodes, which correspond to all the possible positions of the second token on the board (one of the nine positions is already taken). On the second level, there are 9×8, or 72, nodes. On the third level, there are 7 child nodes under each node that correspond to all the possible positions of the third token, a total of 72×7, or 504 nodes, and so on. In each node, you can store a value that indicates whether the corresponding move is good or bad. When the computer has to make a move, it traverses the tree to locate the current status of the board, and then it makes a good move.

Of course, tic-tac-toe is a simple game. In principle, you could design a chess game by using a tree. This tree, however, would grow so large so quickly that it couldn't be stored in any reasonable amount of memory. Moreover, scanning the nodes of this enormous tree would be an extremely slow process. If you also consider that chess moves aren't just good or bad (there are better and not-so-good moves), and you must look ahead many moves to decide which move is the best for the current status of the board, you'll realize that this ad hoc approach is totally infeasible. Practically speaking, such a program requires either infinite resources or infinite time. That's why the chess-playing algorithms use *heuristic approaches*, which store every recorded chess game in a database and consult this database to pick the best next move.

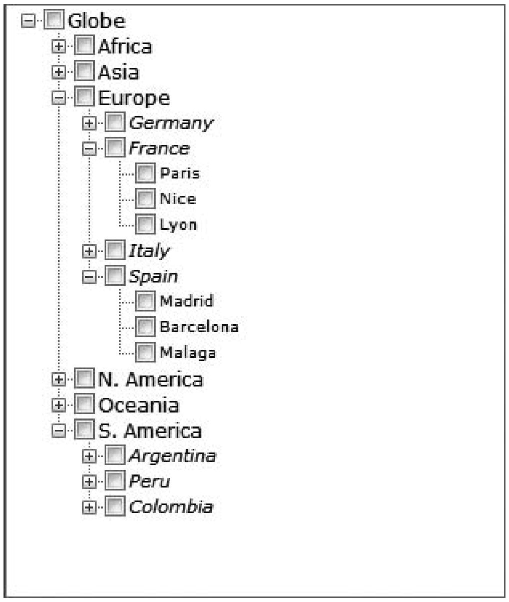
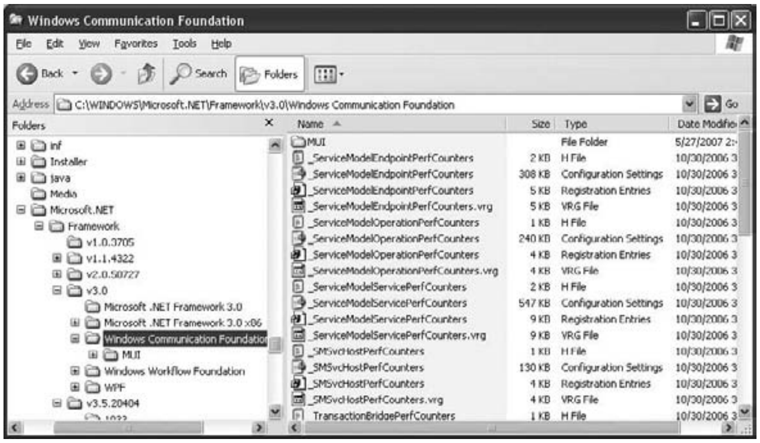


FIGURE 9.3
The tree of Figure 9.2 implemented with a TreeView control

Maintaining a tree structure is a fundamental operation in software design; computer science students spend a good deal of their time implementing tree structures. Fortunately, with Visual Basic you don't have to implement tree structures on your own. The TreeView control is a mechanism for storing hierarchically structured data in a control with a visible interface. The TreeView control hides (or *encapsulates*, in object-oriented terminology) the details of the implementation and allows you to set up tree structures with a few lines of code — in short, all the gain without the pain (almost).

The ListView control implements a simpler structure, known as a *list*. A list's items aren't structured in a hierarchy; they are all on the same level and can be traversed serially, one after the other. You can also think of the list as a multidimensional array, but the list offers more features. A list item can have subitems and can be sorted according to any column. For example, you can set up a list of customer names (the list's items) and assign a number of subitems to each customer: a contact, an address, a phone number, and so on. Or you can set up a list of files with their attributes as subitems. Figure 9.4 shows a Windows folder mapped on a ListView control. Each file is an item, and its attributes are the subitems. As you already know, you can sort this list by filename, size, file type, and so on. All you have to do is click the header of the corresponding column.

The ListView control is a glorified ListBox control. If all you need is a control to store sorted objects, use a ListBox control. If you want more features, such as storing multiple items per row, sorting them in different ways, or locating them based on any subitem's value, you must consider the ListView control. You can also look at the ListView control as a view-only grid.



C:\WINDOWS\Microsoft.NET\Framework\v3.0\Windows Communication Foundation

Name	Size	Type	Date Modifie
MUI		File Folder	5/27/2007 2:
_ServiceModelEndpointPerfCounters	2 KB	H File	10/30/2006 3
_ServiceModelEndpointPerfCounters	308 KB	Configuration Settings	10/30/2006 3
_ServiceModelEndpointPerfCounters	5 KB	Registration Entries	10/30/2006 3
_ServiceModelEndpointPerfCounters.vrg	5 KB	VRG File	10/30/2006 3
_ServiceModelOperationPerfCounters	1 KB	H File	10/30/2006 3
_ServiceModelOperationPerfCounters	240 KB	Configuration Settings	10/30/2006 3
_ServiceModelOperationPerfCounters	4 KB	Registration Entries	10/30/2006 3
_ServiceModelOperationPerfCounters.vrg	4 KB	VRG File	10/30/2006 3
_ServiceModelServicePerfCounters	2 KB	H File	10/30/2006 3
_ServiceModelServicePerfCounters	547 KB	Configuration Settings	10/30/2006 3
_ServiceModelServicePerfCounters	9 KB	Registration Entries	10/30/2006 3
_ServiceModelServicePerfCounters.vrg	9 KB	VRG File	10/30/2006 3
_SMSvcHostPerfCounters	1 KB	H File	10/30/2006 3
_SMSvcHostPerfCounters	130 KB	Configuration Settings	10/30/2006 3
_SMSvcHostPerfCounters	4 KB	Registration Entries	10/30/2006 3
_SMSvcHostPerfCounters.vrg	4 KB	VRG File	10/30/2006 3
TransactionBridgePerfCounters	1 KB	H File	10/30/2006 3

FIGURE 9.4
A folder's files displayed in a ListView control (Details view)

The TreeView and ListView controls are commonly used along with the ImageList control. The ImageList control is a simple control for storing images so they can be retrieved quickly and used at runtime. You populate the ImageList control with the images you want to use on your interface, usually at design time, and then you recall them by an index value at runtime. Before we get into the details of the TreeView and ListView controls, a quick overview of the ImageList control is in order.

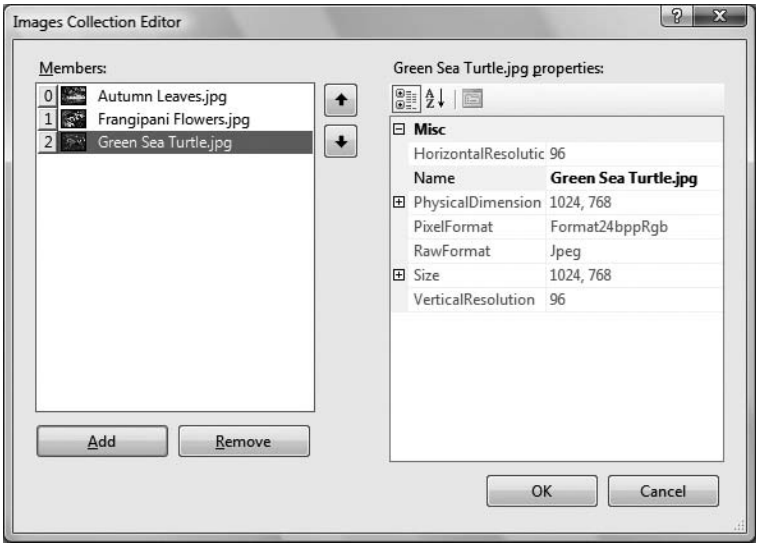
The ImageList Control

The ImageList is a simple control that stores images used by other controls at runtime. For example, a TreeView control can use icons to identify its nodes. The simplest and quickest method of preparing these images is to create an ImageList control and add to it all the icons you need for decorating the TreeView control's nodes. The ImageList control maintains a series of bitmaps in memory that the TreeView control can access quickly at runtime. Keep in mind that the ImageList control can't be used on its own and remains invisible at runtime.

To use the ImageList control in a project, double-click its icon in the Toolbox (you'll find it in the Components tab) to place an instance of the control on your form. To load images to an ImageList control, locate the `Images` property in the Properties window and click the ellipsis button next to the property name. Alternatively, you can select the Choose Images command of the control's context menu. The Images Collection Editor dialog box (see Figure 9.5) will pop up, and you can load all the images you want by selecting the appropriate files. All the images should have the same dimensions — but this is not a requirement. Notice that the ImageList control doesn't resize the images; you must make sure that they have the proper sizes before loading them into the control.

To add an image to the collection, click the Add button. You'll be prompted to select an image file through the Open File dialog box. Each image you select is added to the list. When you select an image in this list, the properties of the image are displayed in the same dialog box — but you can't change these properties, except for the image's name, which is the file's name by default. Add a few images and then close the Images Collection Editor. In the control's Properties window, you can set the size of all images and the `TransparentColor` property, which is a color that will be treated as transparent for all images (this color is also known as the *key color*). The images will be resized accordingly by the control as they're displayed.

FIGURE 9.5
The Images Collection Editor dialog box



Members:

0 Autumn Leaves.jpg
1 Frangipani Flowers.jpg
2 Green Sea Turtle.jpg

Green Sea Turtle.jpg properties:

Misc	
HorizontalResolutic	96
Name	**Green Sea Turtle.jpg**
⊞ PhysicalDimension	1024, 768
PixelFormat	Format24bppRgb
RawFormat	Jpeg
⊞ Size	1024, 768
VerticalResolution	96

The other method of adding images to an ImageList control is to call the `Add` method of the `Images` collection, which contains all the images stored in the control. To add an image at runtime, you must first create an Image object with the image (or icon) you want to add to the control and then call the `Add` method as follows:

```
ImageList1.Images.Add(image)
```

where *`image`* is an Image object with the desired image. You will usually call this method as follows:

```
ImageList1.Images.Add(Image.FromFile(path))
```

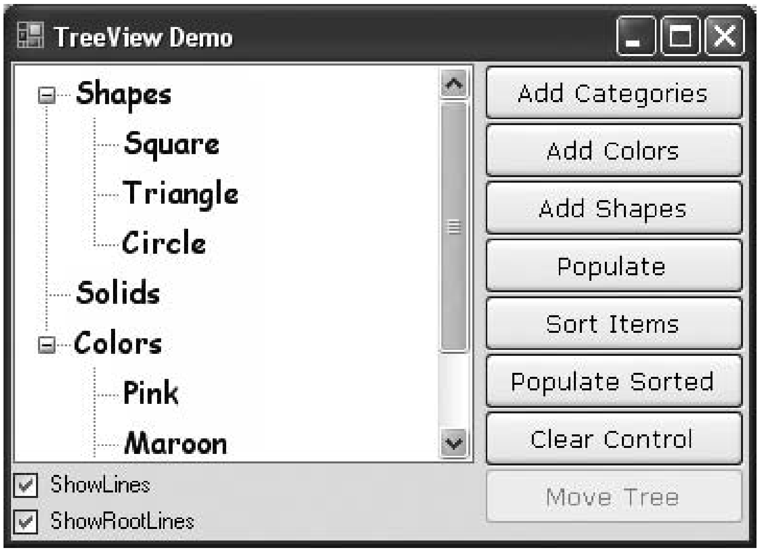
where *`path`* is the full path of the file with the image.

The `Images` collection of the ImageList control is a collection of Image objects, not the files in which the pictures are stored. This means that the image files need not reside on the computer on which the application will be executed, as long as they have been added to the collection at design time.

The TreeView Control

Let's start our discussion with a few simple properties that you can set at design time. To experiment with the properties discussed in this section, open the TreeViewDemo project. The project's main form is shown in Figure 9.6. After setting some properties (they are discussed next), run the project and click the Populate button to populate the control. After that, you can click the other buttons to see the effect of the various property settings on the control.

FIGURE 9.6
The TreeViewDemo project demonstrates the basic properties and methods of the TreeView control.



Here are the basic properties that determine the appearance of the control:

ShowCheckBoxes If this property is True, a check box appears in front of each node. If the control displays check boxes, you can select multiple nodes; otherwise, you're limited to a single selection.

FullRowSelect This True/False value determines whether a node will be selected even if the user clicks outside the node's caption.

HideSelection This property determines whether the selected node will remain highlighted when the focus is moved to another control. By default, the selected node doesn't remain highlighted when the control loses the focus.

HotTracking This property is another True/False value that determines whether nodes are highlighted as the pointer hovers over them. When it's True, the TreeView control behaves like a web document with the nodes acting as hyperlinks — they turn blue while the pointer hovers over them. Use the `NodeMouseHover` event to detect when the pointer hovers over a node.

Indent This property specifies the indentation level in pixels. The same indentation applies to all levels of the tree — each level is indented by the same number of pixels with respect to its parent level.

PathSeparator A node's full name is made up of the names of its parent nodes, separated by a backslash. To use a different separator, set this property to the desired symbol.

ShowLines The `ShowLines` property is a True/False value that determines whether the control's nodes will be connected to its parent items with lines. These lines help users visualize the hierarchy of nodes, and it's customary to display them.

ShowPlusMinus The `ShowPlusMinus` property is a True/False value that determines whether the plus/minus button is shown next to the nodes that have children. The plus button is displayed when the node is collapsed, and it causes the node to expand when clicked. Likewise, the minus sign is displayed when the node is expanded, and it causes the node to collapse when clicked. Users can also expand the current node by pressing the left-arrow button and collapse it with the right-arrow button.

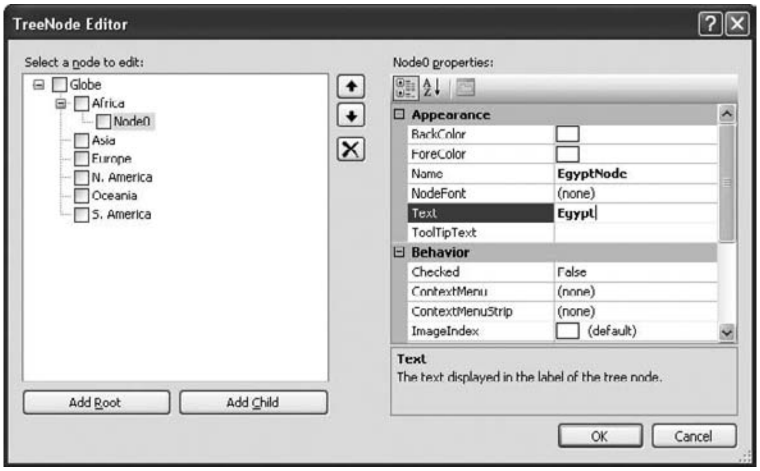
ShowRootLines This is another True/False property that determines whether there will be lines between each node and root of the tree view. Experiment with the ShowLines and ShowRootLines properties to find out how they affect the appearance of the control.

Sorted This property determines whether the items in the control will be automatically sorted. The control sorts each level of nodes separately. In our Globe example, it will sort the continents, then the countries within each continent, and then the cities within each country.

Adding Nodes at Design Time

Let's look now at the process of populating the TreeView control. Adding an initial collection of nodes to a TreeView control at design time is trivial. Locate the Nodes property in the Properties window, and you'll see that its value is Collection. To add items, click the ellipsis button, and the TreeNode Editor dialog box will appear, as shown in Figure 9.7. To add a root item, just click the Add Root button. The new item will be named Node0 by default. You can change its caption by selecting the item in the list and setting its Text property accordingly. You can also change the node's Name property, as well as the node's appearance by using the NodeFont, FontColor, and ForeColor properties.

FIGURE 9.7
The TreeNode Editor dialog box



BackColor	
ForeColor	
Name	EgyptNode
NodeFont	(none)
Text	Egypt
ToolTipText	
Checked	False
ContextMenu	(none)
ContextMenuStrip	(none)
ImageIndex	(default)

Text
The text displayed in the label of the tree node.

To specify an image for the node, set the control's ImageList property to the name of an ImageList control that contains the appropriate images, and then set either the node's ImageKey property to the name of the image, or the node's ImageIndex property to the index of the desired image in the ImageList control. If you want to display a different image when the control is selected, set the SelectedImageKey or the SelectedImageIndex property accordingly.

You can add root items by clicking the Add Root button, or you can add items under the selected node by clicking the Add Child button. Follow these steps to enter the root node with the string **Globe**, a child node for Europe, and two more nodes under Europe: Germany and Italy. I'm assuming that you're starting with a clean control. If your TreeView control contains any items, clear them all by selecting one item at a time in the list and pressing the Delete key, or clicking the delete button (the one with the X icon) on the dialog box.

Click the Add Root button first. A new node is added automatically to the list of nodes, and it is named `Node0`. Select it with the mouse, and its properties appear in the right pane of the TreeNode Editor window. Here you can change the node's `Text` property to **GLOBE**. You can specify the appearance of each node by setting its font and fore/background colors.

Then click the Add Child button, which adds a new node under the GLOBAL root node. Select it with the mouse as before, and change its `Text` property to **Europe**. Then select the newly added node in the list and click the Add Child button again. Name the new node **Germany**. You've successfully added a small hierarchy of nodes. To add another node under Europe, select the Europe node in the list and click the Add Child button again. Name the new item **Italy**.

Continue adding a few cities under each country. You might add child nodes under the wrong parent, which can happen if you forget to select the proper parent node before clicking the Add Child button. To delete a node, select it with the mouse and click the Delete button. Note that when a node is deleted, all the nodes under it are deleted, too. Moreover, this action can't be undone. So be careful when deleting nodes.

Click the OK button to close the TreeNode Editor's window and return to your form. The nodes you added to the TreeView control are there, but they're collapsed. Only the root nodes are displayed with the plus sign in front of their names. Click the plus sign to expand the tree and see its child nodes. The TreeView control behaves the same at design time as it does at runtime — as far as navigating the tree goes, at least.

The nodes added to a TreeView control at design time will appear each time the form is loaded. You can add new nodes through your code, and you will see how this is done in the following section.

Adding Nodes at Runtime

Adding items to the control at runtime is a bit more involved. All the nodes belong to the control's `Nodes` collection, which is made up of TreeNode objects. To access the `Nodes` collection, use the following expression, where *`TreeView1`* is the control's name and `Nodes` is a collection of TreeNode objects:

```
TreeView1.Nodes
```

This expression returns a collection of TreeNode objects and exposes the proper members for accessing and manipulating the individual nodes. The control's `Nodes` property is the collection of all root nodes.

To access the first node, use the expression `TreeView.Nodes(0)` (this is the Globe node in our example). The `Text` property returns the node's value, which is a string. `TreeView1.Nodes(0).Text` is the caption of the root node on the control. The caption of the second node on the same level is `TreeView1.Nodes(1).Text`, and so on.

The following statements print the strings shown highlighted below them (these strings are not part of the statements; they're the output that the statements produce):

```
Debug.WriteLine(TreeView1.Nodes(0).Text)
GLOBE
Debug.WriteLine(TreeView1.Nodes(0).Nodes(0).Text)
Europe
Debug.WriteLine(TreeView1.Nodes(0).Nodes(0).Nodes(1).Text)
Italy
```

Let's take a closer look at these expressions. `TreeView1.Nodes(0)` is the first root node, the Globe node. Under this node, there is a collection of nodes, the `TreeView1.Nodes(0).Nodes` collection. Each node in this collection is a continent name. The first node in this collection is Europe, and you can access it with the expression `TreeView1.Nodes(0).Nodes(0)`. If you want to change the appearance of the node Europe, type a period after the preceding expression to access its properties (the `NodeFont` property to set its font, the `ForeColor` property to set it color, the `ImageIndex` property, and so on). Likewise, this node has its own `Nodes` collection, which contains the countries under the specific continent.

Adding New Nodes

The `Add` method adds a new node to the `Nodes` collection. The `Add` method accepts as an argument a string or a TreeNode object. The simplest form of the `Add` method is

```
newNode = Nodes.Add(nodeCaption)
```

where *`nodeCaption`* is a string that will be displayed on the control. Another form of the `Add` method allows you to add a TreeNode object directly (*`nodeObj`* is a properly initialized TreeNode variable):

```
newNode = Nodes.Add(nodeObj)
```

To use this form of the method, you must first declare and initialize a TreeNode object:

```
Dim nodeObj As New TreeNode
nodeObj.Text = "Tree Node"
nodeObj.ForeColor = Color.BlueViolet
TreeView1.Nodes.Add(nodeObj)
```

The last overloaded form of the `Add` method allows you to specify the index in the current `Nodes` collection, where the node will be added:

```
newNode = Nodes.Add(index, nodeObj)
```

The *`nodeObj`* TreeNode object must be initialized as usual.
To add a child node to the root node, use a statement such as the following:

```
TreeView1.Nodes(0).Nodes.Add("Asia")
```

To add a country under Asia, use a statement such as the following:

```
TreeView1.Nodes(0).Nodes(1).Nodes.Add("Japan")
```

The expressions can get quite lengthy. The proper way to add child items to a node is to create a TreeNode variable that represents the parent node, under which the child nodes will be added. Let's say that the *`ContinentNode`* variable in the following example represents the node Europe:

```
Dim ContinentNode As TreeNode
ContinentNode = TreeView1.Nodes(0).Nodes(2)
```

Then you can add child nodes to the `ContinentNode` node:

```
ContinentNode.Nodes.Add("France")
ContinentNode.Nodes.Add("Germany")
```

To add yet another level of nodes, the city nodes, create a new variable that represents a specific country. The `Add` method actually returns a TreeNode object that represents the newly added node, so you can add a country and a few cities by using statements such as the following:

```
Dim CountryNode As TreeNode
CountryNode = ContinentNode.Nodes.Add("Germany")
CountryNode.Nodes.Add("Berlin")
CountryNode.Nodes.Add("Frankfurt")
```

Then you can continue adding countries under another continent as follows:

```
CountryNode = ContinentNode.Nodes.Add("Italy")
CountryNode.Nodes.Add("Rome")
```

THE *NODES* COLLECTION MEMBERS

The `Nodes` collection exposes the usual members of a collection. The `Count` property returns the number of nodes in the `Nodes` collection. Again, this is not the total number of nodes in the control, just the number of nodes in the current `Nodes` collection. The expression

```
TreeView1.Nodes.Count
```

returns the number of all nodes in the first level of the control. In the case of the Globe example, it returns the value 1. The expression

```
TreeView1.Nodes(0).Nodes.Count
```

returns the number of continents in the Globe example. Again, you can simplify this expression by using an intermediate TreeNode object:

```
Dim Continents As TreeNode
Continents = TreeView1.Nodes(0)
Debug.WriteLine( _
    "There are " & Continents.Nodes.Count.ToString & _
    " continents on the control")
```

The `Clear` method removes all the child nodes from the current node. If you apply this method to the control's root node, it will clear the control. To remove all the cities under the Germany node, use a statement such as the following:

```
TreeView1.Nodes(0).Nodes(2).Nodes(1).Nodes.Clear
```

This example assumes that the third node under Globe corresponds to Europe, and the second node under Europe corresponds to Germany.

The Item property retrieves a node specified by an index value. The expression Nodes.Item(1) is equivalent to the expression Nodes(1). Finally, the Remove method removes a node from the Nodes collection. Its syntax is

```
Nodes.Remove(index)
```

where *index* is the order of the node in the current Nodes collection. To remove the selected node, call the Remove method on the SelectedNode property without arguments:

```
TreeView1.SelectedNode.Remove
```

Or you can apply the Remove method to a TreeNode object that represents the node you want to remove:

```
Dim Node As TreeNode
Node = TreeView1.Nodes(0).Nodes(7)
Node.Remove
```

There are four properties that allow you to retrieve any node at the current segment of the tree: FirstNode, NextNode, PrevNode, and LastNode. Let's say the current node is the Germany node. The FirstNode property will return the first city under Germany (the first node in the current segment of the tree), and LastNode will return the last city under Germany. PrevNode and NextNode allow you to iterate through the nodes of the current segment: They return the next and previous nodes on the current segment of the tree (the *sibling nodes*, as they're called). See the section called ''Enumerating the Nodes Collection'' later in this chapter for an example.

Basic Nodes Properties

There are a few properties you will find extremely handy as you program the TreeView control. The IsVisible property is a True/False value indicating whether the node to which it's applied is visible. To bring an invisible node into view, call its EnsureVisible method:

```
If Not TreeView1.SelectedNode.IsVisible Then
    TreeView1.EnsureVisible
End If
```

How can the selected node be invisible? It can, if you select it from within your code in a search operation. The IsSelected property returns True if the specified node is selected, while the IsExpanded property returns True if the specified node is expanded. You can toggle a node's state by calling its Toggle method. You can also expand or collapse a node by calling its Expand or Collapse method, respectively. Finally, you can collapse or expand all nodes by calling the CollapseAll or ExpandAll method of the TreeView control.

VB 2008 at Work: The TreeViewDemo Project

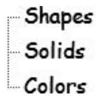
It's time to demonstrate the members discussed so far with an example. The project you'll build in this section is the TreeViewDemo project. The project's main form is shown in Figure 9.6.

The Add Categories button adds the three top-level nodes to the TreeView control via the statements shown in Listing 9.1. These are the control's root nodes. The other two Add buttons add nodes under the root nodes.

LISTING 9.1: The Add Categories Button

```
Protected Sub AddCategories_Click(...) _
              Handles AddCategories.Click
   TreeView1.Nodes.Add("Shapes")
   TreeView1.Nodes.Add("Solids")
   TreeView1.Nodes.Add("Colors")
End Sub
```

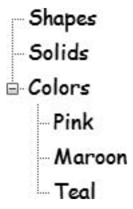
When these statements are executed, three root nodes are added to the list. After clicking the Add Categories button, your TreeView control looks like the one shown here.



To add a few nodes under the node Colors, you must retrieve the `Colors Nodes` collection and add child nodes to this collection, as shown in Listing 9.2.

LISTING 9.2: The Add Colors Button

```
Protected Sub AddColors_Click(...) _
              Handles AddColors.Click
   Dim cnode As TreeNode
   cnode = TreeView1.Nodes(2)
   cnode.Nodes.Add("Pink")
   cnode.Nodes.Add("Maroon")
   cnode.Nodes.Add("Teal")
End Sub
```



When these statements are executed, three more nodes are added under the Colors node, but the Colors node won't be expanded. Therefore, its child nodes won't be visible. To see its child nodes, you must double-click the Colors node to expand it (or click the plus sign in front of it, if there is one). The same TreeView control with its Colors node expanded is shown to the left. Alternatively, you can add a statement that calls the `Expand` method of the `cnode` object, after adding the color nodes to the control:

```
cnode.Expand()
```

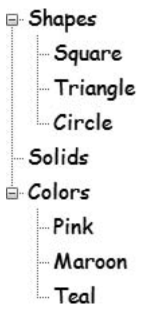
Run the project, click the first button (Add Categories), and then click the second button (Add Colors). If you click the Add Colors button first, you'll get a `NullReferenceException`, indicating that the node can't be inserted unless its parent node already exists. I added a few statements in the TreeViewDemo project's code to disable the buttons that generate similar runtime errors.

To add child nodes under the Shapes node, use the statements shown in Listing 9.3. This is the Add Shapes button's `Click` event handler.

LISTING 9.3: The Add Shapes Button

```
Protected Sub AddShapes_Click(...) _
              Handles AddShapes.Click
   Dim snode As TreeNode
   snode = treeview1.Nodes(0)
   snode.Nodes.Add("Square")
   snode.Nodes.Add("Triangle")
   snode.Nodes.Add("Circle")
End Sub
```

If you run the project and click the three buttons in the order in which they appear on the form, the TreeView control will be populated with colors and shapes. If you double-click the items Colors and Shapes, the TreeView control's nodes will be expanded.



Notice that the code knows the order of the root node to which it's adding child nodes. This approach doesn't work with a sorted tree. If your TreeView control is sorted, you must create a hierarchy of nodes explicitly by using the following statements:

```
snode = TreeView1.Nodes.Add("Shapes")
snode.Add("Square")
snode.Add("Circle")
snode.Add("Triangle")
```

These statements will work regardless of the control's `Sorted` property setting. The three shapes will be added under the Shapes node, and their order will be determined automatically. Of course, you can always populate the control in any way you like and then turn on the `Sorted` property.

Inserting a Root Node

Let's revise the code we've written so far to display all the nodes under a new header. In other words, we'll add a new node called *Items* that will act as the root node for existing nodes. It's not a common operation, but it's an interesting example of how to manipulate the nodes of a TreeView control at runtime.

First, we must add the new root node. Before we do so, however, we must copy all the first-level nodes into local variables. We'll use these variables to add the current root nodes under the new (and single) root node. There are three root nodes currently in our control, so we need three local variables. The three variables are of the TreeNode type, and they're set to the root nodes of the original tree. Then we must clear the entire tree, add the new root node (the Items node), and finally add all the copied nodes under the new root. The code behind the Move Tree button is shown in Listing 9.4.

Listing 9.4: Moving an Entire Tree

```
Protected Sub MoveTree_Click(...) _
              Handles bttnMoveTree.Click
   Dim colorNode, shapeNode, solidNode As TreeNode
   colorNode = TreeView1.Nodes(0)
   shapeNode = TreeView1.Nodes(1)
   solidNode = TreeView1.Nodes(2)
   TreeView1.Nodes.Clear()
   TreeView1.Nodes.Add("Items")
   TreeView1.Nodes(0).Nodes.Add(colorNode)
   TreeView1.Nodes(0).Nodes.Add(shapeNode)
   TreeView1.Nodes(0).Nodes.Add(solidNode)
End Sub
```

You can revise this code so that it uses an array of Node objects instead of individual variables to store all the root nodes. For a routine that will work with any tree, you must assume that the number of nodes is unknown, so the ArrayList would be a better choice. The following loop stores all the root nodes of the *TreeView1* control to the `TVList` ArrayList:

```
Dim TVList As New ArrayList
Dim node As TreeNode
For Each node in TreeView1.Nodes
   TVList.Add(node)
Next
```

Likewise, the following loop extracts the root nodes from the `TVList` ArrayList:

```
Dim node As TreeNode
Dim itm As Object
TreeView1.Nodes.Clear
For Each itm In TVList
   node = CType(itm, TreeNode)
```

```
      TreeView1.Nodes.Add(node)
   Next
```

Enumerating the *Nodes* Collection

As you saw in the previous example, a Node object can include an entire tree under it. When we move a node, it takes with it the entire `Nodes` collection located under it. You can scan all the nodes in a `Nodes` collection by using a loop, which starts with the first node and then moves to the next node with the help of the `FirstNode` and `NextNode` properties. The following loop prints the names of all continents in the GlobeTree control:

```
Dim CurrentNode As TreeNode
CurrentNode = GlobeTree.Nodes(0).Nodes(0).FirstNode
While CurrentNode IsNot Nothing
   Debug.WriteLine(CurrentNode.text)
   CurrentNode = CurrentNode.NextNode
End While
```

The last property demonstrated by the TreeViewDemo project is the `Sorted` property, which sorts the child nodes of the node to which it's applied. When you set the `Sorted` property of a node to True, every child node you attach to it will be inserted automatically in alphabetical order. If you reset the `Sorted` property to False, any child nodes you attach will be appended to the end of the existing sorted nodes.

VB 2008 at Work: The Globe Project

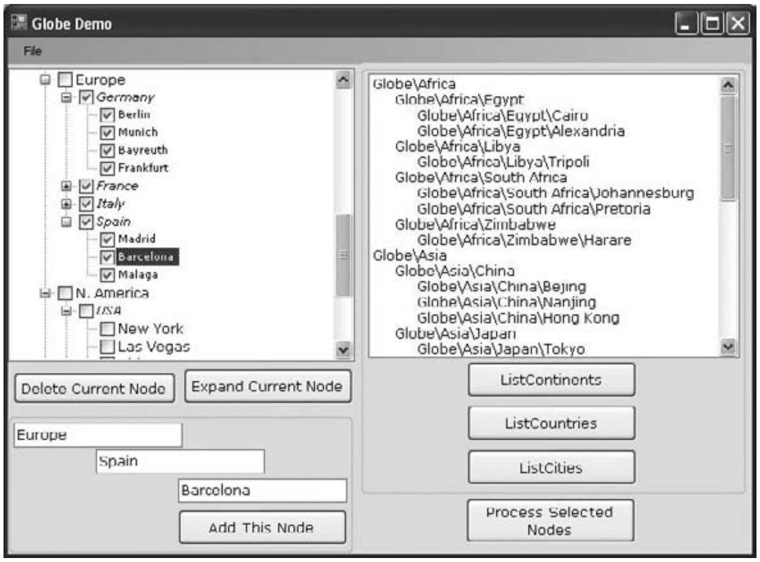
The Globe project demonstrates many of the techniques we've discussed so far. It's not the simplest example of a TreeView control, and its code is lengthy, but it will help you understand how to manipulate nodes at runtime. Because TreeView is not a simple control, before ending this section I want to show you a nontrivial example that you can use as a starting point for your own custom applications.

The Globe project consists of a single form, which is shown in Figure 9.8. The TreeView control at the left contains a rather obvious tree structure that shows continents, countries, and cities. The control is initially populated with the continents, which were added at design time. The countries and cities are added from within the form's `Load` event handler. Although the continents were added at design time, there's no particular reason not to add them to the control at runtime. It would have been simpler to add all the nodes at runtime by using the TreeNode Editor, but I decided to add a few nodes at design time just for demonstration purposes.

When a node is selected from the TreeView control, its text is displayed in the TextBox controls at the bottom of the form. When a continent name is selected, the continent's name appears in the first TextBox, and the other two TextBoxes are empty. When a country is selected, its name appears in the second TextBox, and its continent appears in the first TextBox. Finally, when a city is selected, it appears in the third TextBox, along with its country and continent in the other two TextBoxes.

You can also use the same TextBox controls to add new nodes. To add a new continent, just supply the name of the continent in the first TextBox and leave the other two empty. To add a new country, supply its name in the second TextBox and the name of the continent it belongs to in the first one. Finally, to add a city, supply a continent, country, and city name in the three TextBoxes. The program will add new nodes as needed.

FIGURE 9.8
The Globe project



```
Globe\Africa
    Globe\Africa\Egypt
        Globe\Africa\Egypt\Cairo
        Globe\Africa\Egypt\Alexandria
    Globe\Africa\Libya
        Globe\Africa\Libya\Tripoli
    Globe\Africa\South Africa
        Globe\Africa\South Africa\Johannesburg
        Globe\Africa\South Africa\Pretoria
    Globe\Africa\Zimbabwe
        Globe\Africa\Zimbabwe\Harare
Globe\Asia
    Globe\Asia\China
        Globe\Asia\China\Beijing
        Globe\Asia\China\Nanjing
        Globe\Asia\China\Hong Kong
    Globe\Asia\Japan
        Globe\Asia\Japan\Tokyo
```

Run the Globe application and expand the continents and countries to see the tree structure of the data stored in the control. Add new nodes to the control, and enumerate these nodes by clicking the buttons on the right-hand side of the form. These buttons list the nodes at a given level (continents, countries, and cities). When you add new nodes, the code places them in their proper place in the list. If you specify a new city and a new country under an existing continent, a new country node will be created under the specified continent, and a new city node will be inserted under the specified country.

ADDING NEW NODES

Let's take a look at the code of the Globe project. We'll start by looking at the code that populates the TreeView control. The root node (GLOBE) and the continent names were added at design time through the TreeNode Editor.

When the application starts, the code adds the countries to each continent and adds the cities to each country. The code in the form's `Load` event goes through all the continents already in the control and examines their `Text` properties. Depending on the continent represented by the current node, the code adds the corresponding countries and some city nodes under each country node.

If the current node is Africa, the first country to be added is Egypt. The Egypt node is added to the *`ContinentNode`* variable. The new node is returned as a TreeNode object and is stored in the *`CountryNode`* variable. Then the code uses this object to add nodes that correspond to cities under the Egypt node. The form's `Load` event handler is quite lengthy, so I'm showing only the code that adds the first country under each continent and the first city under each country (see Listing 9.5). The variable *`GlobeNode`* is the root node of the TreeView control, and it was declared and initialized with the following statement:

```
Dim GlobeNode As TreeNode = GlobeTree.Nodes(0)
```

LISTING 9.5: Adding the Nodes of Africa

```
For Each ContinentNode In GlobeNode.Nodes
   Select Case ContinentNode.Text
      Case "Europe"
         CountryNode = ContinentNode.Nodes.Add("Germany")
         CountryNode.Nodes.Add("Berlin")
      Case "Asia"
         CountryNode = ContinentNode.Nodes.Add("China")
         CountryNode.Nodes.Add("Beijing")
      Case "Africa"
         CountryNode = ContinentNode.Nodes.Add("Egypt")
         CountryNode.Nodes.Add("Cairo")
         CountryNode.Nodes.Add("Alexandria")
      Case "Oceania"
         CountryNode = ContinentNode.Nodes.Add("Australia")
         CountryNode.Nodes.Add("Sydney")
      Case "N. America"
         CountryNode = ContinentNode.Nodes.Add("USA")
         CountryNode.Nodes.Add("New York")
      Case "S. America"
         CountryNode = ContinentNode.Nodes.Add("Argentina")
   End Select
Next
```

The remaining countries and their cities are added via similar statements, which you can examine if you open the Globe project. Notice that the GlobeTree control could have been populated entirely at design time, but this wouldn't be much of a demonstration. Let's move on to a few more interesting aspects of programming the TreeView control.

Retrieving the Selected Node

The selected node is given by the property `SelectedNode`. After retrieving the selected node, you can also retrieve its parent node and the entire path to the root node. The parent node of the selected node is `TreeView1.SelectedNode.Parent`. If this node has a parent, you can retrieve it by calling the `Parent` property of the previous expression. The `FullPath` property of a node retrieves the selected node's full path. The `FullPath` property of the Rome node is as follows:

```
GLOBE\Europe\Italy\Rome
```

The slashes separate the segments of the node's path. As mentioned earlier, you can specify any other character for this purpose by setting the control's `PathSeparator` property.

To remove the selected node from the tree, call the `Remove` method:

```
TreeView1.SelectedNode.Remove
```

If the selected node is a parent control for other nodes, the `Remove` method will take with it all the nodes under the selected one. To select a node from within your code, set the control's SelectedNode property to the TreeNode object that represents the node you want to select.

One of the operations you'll want to perform with the TreeView control is to capture the selection of a node. The TreeView control fires the BeforeSelect and AfterSelect events, which notify your application about the selection of another node. If you need to know which node was previously selected, you must use the BeforeSelect event. The second argument of both events has two properties, TreeNode and Action, which let you find out the node that fired the event and the action that caused it. The e.Node property is a TreeViewNode object that represents the selected node. Use it in your code as you would use any other node of the control. The e.Action property is a member of the TreeViewAction enumeration (ByKeyboard, ByMouse, Collapse, Expand, Unknown). Use this property to find out the action that caused the event. The actions of expanding and collapsing a tree branch fire their own events, which are the BeforeExpand/AfterExpand and the BeforeCollapse/AfterCollapse events, respectively.

The Globe project retrieves the selected node and extracts the parts of the node's path. The individual components of the path are displayed in the three TextBox controls at the bottom of the form. Listing 9.6 shows the event handler for the TreeView control's AfterSelect event.

LISTING 9.6: Processing the Selected Node

```
Private Sub GlobeTree_AfterSelect(...) _
            Handles GlobeTree.AfterSelect
   If GlobeTree.SelectedNode Is Nothing Then Exit Sub
   Dim components() As String
   txtContinent.Text = ""
   txtCountry.Text = ""
   txtCity.Text = ""

    Dim separators() As Char
    separators = GlobeTree.PathSeparator.ToCharArray
    components = _
             GlobeTree.SelectedNode.FullPath. _
             ToString.Split(separators)
   If components.Length > 1 Then _
                txtContinent.Text = components(1)
   If components.Length > 2 Then _
                txtCountry.Text = components(2)
   If components.Length > 3 Then _
                txtCity.Text = components(3)
End Sub
```

The Split method of the String data type extracts the parts of a string that are delimited by the PathSeparator character (the backslash character). If any of the captions contain this character, you should change the default to a different character by setting the PathSeparator property to some other character.

The code behind the Delete Current Node and Expand Current Node buttons is simple. To delete a node, call the selected node's Remove method. To expand a node, call the selected node's Expand method.

Processing Multiple Selected Nodes

The GlobeTree control has its ShowCheckBoxes property set to True so that users can select multiple nodes. I added this feature to demonstrate how you can allow users to select any number of nodes and then process them.

As you will notice by experimenting with the TreeView control, you can select a node that has subordinate nodes, but these nodes will not be affected; they will remain deselected (or selected, if you have already selected them). In most cases, however, when we select a parent node, we actually intend to select all the nodes under it. When you select a country, for example, you're in effect selecting not only the country, but also all the cities under it. The code of the Process Selected Nodes button assumes that when a parent node is selected, the code must also select all the nodes under it.

Let's look at the code that iterates through the control's nodes and isolates the selected ones. It doesn't really process them; it simply prints their captions in the ListBox control. However, you can call a function to process the selected nodes in any way you like. The code behind the Process Selected Nodes button starts with the continents. It creates a TreeNodeCollection with all the continents and then goes through the collection with a For Each...Next loop. At each step, it creates another TreeNodeCollection, which contains all the subordinate nodes (the countries under the selected continent) and goes through the new collection. This loop is also interrupted at each step to retrieve the cities in the current country and process them with another loop. The code behind the Process Selected Nodes button is straightforward, as you can see in Listing 9.7.

Listing 9.7: Processing All Selected Nodes

```
Protected Sub bttnProcessSelected_Click(...) _
              Handles bttnProcessSelected.Click
   Dim continent, country, city As TreeNode
   Dim Continents, Countries, Cities As TreeNodeCollection
   ListBox1.Items.Clear()
   Continents = GlobeTree.Nodes(0).Nodes
   For Each continent In Continents
      If continent.Checked Then ListBox1.Items.Add(continent.FullPath)
      Countries = continent.Nodes
      For Each country In Countries
         If country.Checked Or country.Parent.Checked Then _
            ListBox1.Items.Add("     " & country.FullPath)
         Cities = country.Nodes
         For Each city In Cities
            If city.Checked Or city.Parent.Checked Or _
               city.Parent.Parent.Checked Then _
                  ListBox1.Items.Add("          " & city.FullPath)
         Next
      Next
   Next
End Sub
```

The code examines the Checked property of the current node, as well as the Checked property of the parent node, all the way to the root node. If any of them is True, the node is considered selected. You should try to add the appropriate code to select all subordinate nodes of a parent node when the parent node is selected (whether you deselect the subordinate nodes when the parent node is deselected is entirely up to you and depends on the type of application you're developing). The `Nodes` collection exposes the `GetEnumerator` method, and you can revise the last listing so that it uses an enumerator in place of each `For Each`. . .`Next` loop. If you want to retrieve the selected nodes only, and ignore the unselected child nodes of a selected parent node, use the `CheckedNodes` collection.

Adding New Nodes

The Add This Node button lets the user add new nodes to the tree at runtime. The number and type of the node(s) added depend on the contents of the TextBox controls:

- If only the first TextBox control contains text, a new continent will be added.
- If the first two TextBox controls contain text:
  - If the continent exists, a new country node is added under the specified continent.
  - If the continent doesn't exist, a new continent node is added, and then a new country node is added under the continent's node.
- If all three TextBox controls contain text, the program adds a continent node (if needed), then a country node under the continent node (if needed), and finally, a city node under the country node.

Obviously, you can omit a city, or a city and country, but you can't omit a continent name. Likewise, you can't specify a city without a country, or a country without a continent. The code will prompt you accordingly when it detects any condition that prevents it from adding the new node. If the node exists already, the program selects the existing node and doesn't issue any warnings. The Add This Node button's code is shown in Listing 9.8.

Listing 9.8: Adding Nodes at Runtime

```
Private Sub bttnAddNode_Click(...) _
            Handles bttnAddNode.Click
   Dim nd As TreeNode
   Dim Continents As TreeNode
   If txtContinent.Text.Trim <> "" Then
      Continents = GlobeTree.Nodes(0)
      Dim ContinentFound, CountryFound, CityFound As Boolean
      Dim ContinentNode, CountryNode, CityNode As TreeNode
      For Each nd In Continents.Nodes
         If nd.Text.ToUpper = txtContinent.Text.ToUpper Then
            ContinentFound = True
            Exit For
         End If
      Next
      If Not ContinentFound Then
         nd = Continents.Nodes.Add(txtContinent.Text)
      End If
```

```
            ContinentNode = nd
            If txtCountry.Text.Trim <> "" Then
               Dim Countries As TreeNode
               Countries = ContinentNode
               If Not Countries Is Nothing Then
                  For Each nd In Countries.Nodes
                     If nd.Text.ToUpper = txtCountry.Text.ToUpper Then
                        CountryFound = True
                        Exit For
                     End If
                  Next
               End If
               If Not CountryFound Then
                  nd = ContinentNode.Nodes.Add(txtCountry.Text)
               End If
               CountryNode = nd
               If txtCity.Text.Trim <> "" Then
                  Dim Cities As TreeNode
                  Cities = CountryNode
                  If Not Cities Is Nothing Then
                     For Each nd In Cities.Nodes
                        If nd.Text.ToUpper = txtCity.Text.ToUpper Then
                           CityFound = True
                           Exit For
                        End If
                     Next
                  End If
                  If Not CityFound Then
                     nd = CountryNode.Nodes.Add(txtCity.Text)
                  End If
                  CityNode = nd
               End If
            End If
         End If
      End Sub
```

The listing is quite lengthy, but it's not hard to follow. First, it attempts to find a continent that matches the name in the first TextBox. If it succeeds, it does not need to add a new continent node. If not, a new continent node must be added. To avoid simple data-entry errors, the code converts the continent names to uppercase before comparing them to the uppercase of each node's name. The same happens with the countries and the cities. As a result, each node's pathname is unique — you can't have the same city name under the same country more than once. It is possible, however, to add the same city name to two different countries.

Listing Continents/Countries/Cities

The three buttons ListContinents, ListCountries, and ListCities populate the ListBox control with the names of the continents, countries, and cities, respectively. The code is straightforward and is based on the techniques discussed in previous sections. To print the names of the continents, it iterates through the children of the GLOBE node. Listing 9.9 shows the complete code of the ListContinents button.

Listing 9.9: Retrieving the Continent Names

```
Private Sub bttnListContinents_Click(...) _
            Handles bttnListContinents.Click
   Dim Nd As TreeNode, continentNode As TreeNode
   Dim continent As Integer, continents As Integer
   ListBox1.Items.Clear()
   Nd = GlobeTree.Nodes(0)
   continents = Nd.Nodes.Count
   continentNode = Nd.Nodes(0)
   For continent = 1 To continents
      ListBox1.Items.Add(continentNode.Text)
      continentNode = continentNode.NextNode
   Next
End Sub
```

The code behind the ListCountries button is equally straightforward, although longer. It must scan each continent, and within each continent, it must scan in a similar fashion the continent's child nodes. To do this, you must set up two nested loops: the outer one to scan the continents, and the inner one to scan the countries. The complete code for the ListCountries button is shown in Listing 9.10. Notice that in this example, I used `For...Next` loops to iterate through the current level's nodes, and I also used the `NextNode` method to retrieve the next node in the sequence.

Listing 9.10: Retrieving the Country Names

```
Private Sub bttnListCountries_Click(...) _
            Handles bttnListCountries.Click
   Dim Nd, CountryNode, ContinentNode As TreeNode
   Dim continent, continents, country, countries As Integer
   ListBox1.Items.Clear()
   Nd = GlobeTree.Nodes.Item(0)
   continents = Nd.Nodes.Count
   ContinentNode = Nd.Nodes(0)
   For continent = 1 To continents
      countries = ContinentNode.Nodes.Count
      CountryNode = ContinentNode.Nodes(0)
      For country = 1 To countries
         ListBox1.Items.Add(CountryNode.Text)
         CountryNode = CountryNode.NextNode
      Next
      ContinentNode = ContinentNode.NextNode
   Next
End Sub
```

When the `ContinentNode.Next` method is called, it returns the next node in the Continents level. Then the property `ContinentNode.Nodes(0)` returns the first node in the Countries level. As you can guess, the code of the ListCities button uses the same two nested lists as the previous listing and an added inner loop, which scans the cities of each country.

The code behind these command buttons requires some knowledge of the information stored in the tree. The code will work with trees that have two or three levels of nodes such as the Globe tree, but what if the tree's depth is allowed to grow to a dozen levels? A tree that represents the structure of a folder on your hard disk, for example, might easily contain a dozen nested folders. Obviously, to scan the nodes of this tree, you can't put together unlimited nested loops. The next section describes a technique for scanning any tree, regardless of how many levels it contains.

Finally, the application's File menu contains commands for storing the nodes to a file and loading the same nodes in a later session. These commands use serialization, a topic that's discussed in detail in Chapter 16, ''XML and Object Serialization.'' For now, you can use these commands to persist the edited nodes to a disk file and read them back.

Scanning the TreeView Control

You have seen how to scan the entire tree of the TreeView control by using a `For Each...Next` loop that iterates through the `Nodes` collection. This technique, however, requires that you know the structure of the tree, and you must write as many nested loops as there are nested levels of nodes. It works with simple trees, but it's quite inefficient when it comes to mapping a file system to a TreeView control. The following section explains how to iterate through a TreeView control's node, regardless of the nesting depth.

VB 2008 at Work: The TreeViewScan Project

The TreeViewScan project, whose main form is shown in Figure 9.9, demonstrates the process of scanning the nodes of a TreeView control. The form contains a TreeView control on the left, which is populated with the same data as the Globe project, and a ListBox control on the right, in which the tree's nodes are listed. Child nodes in the ListBox control are indented according to the level to which they belong.

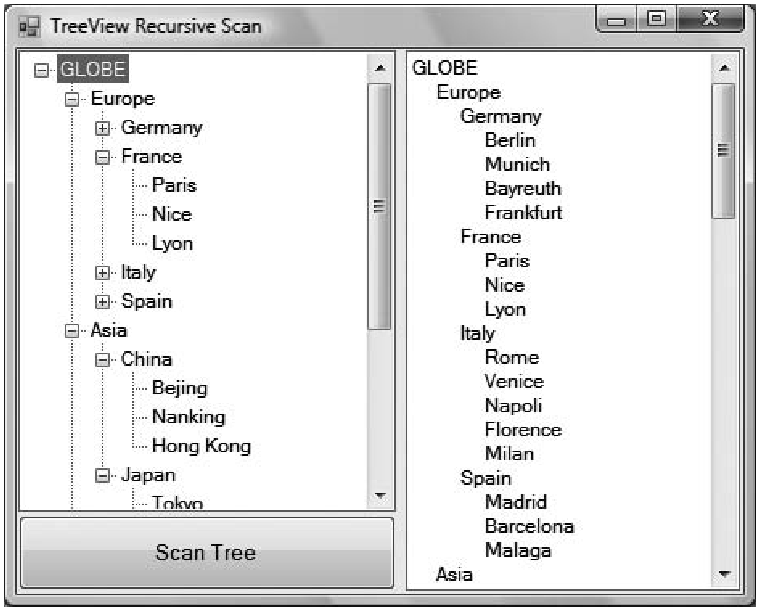
Scanning the child nodes in a tree calls for a *recursive procedure:* a procedure that calls itself. Think of a tree structure that contains all the files and folders on your `C:` drive. If this structure contained no subfolders, you'd need to set up a loop to scan each folder, one after the other. Because most folders contain subfolders, the process must be interrupted at each folder to scan the subfolders of the current folder. The process of scanning a drive recursively is described in detail in Chapter 15, ''Accessing Folders and Files.''

Recursive Scanning of the Nodes Collection

To scan the nodes of the *`TreeView1`* control, start at the top node of the control by using the following statement:

```
ScanNode(GlobeTree.Nodes(0))
```

This is the code behind the Scan Tree button, and it doesn't get any simpler. It calls the `ScanNode()` subroutine to scan the child nodes of a specific node, which is passed to the subroutine as an argument. `GlobeTree.Nodes(0)` is the root node. By passing the root node to the `ScanNode()` subroutine, we're in effect asking it to scan the entire tree.



GLOBE
Europe
Germany
France
Paris
Nice
Lyon
Italy
Spain
Asia
China
Bejing
Nanking
Hong Kong
Japan
Tokyo

```
GLOBE
     Europe
          Germany
               Berlin
               Munich
               Bayreuth
               Frankfurt
          France
               Paris
               Nice
               Lyon
          Italy
               Rome
               Venice
               Napoli
               Florence
               Milan
          Spain
               Madrid
               Barcelona
               Malaga
     Asia
```

FIGURE 9.9 The TreeViewScan application demonstrates how to scan the nodes of a TreeView control recursively.

This example assumes that the TreeView control contains a single root node and that all other nodes are under the root node. If your control contains multiple root nodes, then you must set up a small loop and call the `ScanNode()` subroutine once for each root node:

```
For Each node In GlobeTree.Nodes
   ScanNode(node)
Next
```

Let's look now at the `ScanNode()` subroutine shown in Listing 9.11.

LISTING 9.11: Scanning a Tree Recursively

```
Sub ScanNode(ByVal node As TreeNode)
   Dim thisNode As TreeNode
   Static indentationLevel As Integer
   Application.DoEvents()
   ListBox1.Items.Add(Space(indentationLevel) & node.Text)
   If node.Nodes.Count > 0 Then
      indentationLevel += 5
      For Each thisNode In node.Nodes
         ScanNode(thisNode)
      Next
      indentationLevel -= 5
   End If
End Sub
```

This subroutine is deceptively simple. First, it adds the caption of the current node to the *ListBox1* control. If this node (represented by the *Node* variable) contains child nodes, the code must scan them all. The `Node.Nodes.Count` method returns the number of nodes under the current node; if this value is positive, we must scan all the items of the `Node.Nodes` collection. To do this, the `ScanNode()` subroutine must call itself, passing a different argument each time. If you're familiar with recursive procedures, you'll find the code quite simple. You may find the notion of a function calling itself a bit odd, but it's no different from calling another function. The execution of the function that makes the call is suspended until the called function returns.

You can use the `ScanNode()` subroutine as is to scan any TreeView control. All you need is a reference to the root node (or the node you want to scan recursively), which you must pass to the `ScanNode()` subroutine as an argument. The subroutine will scan the entire subtree and display its nodes in a ListBox control. The nodes will be printed one after the other. To make the list easier to read, the code indents the names of the nodes by an amount that's proportional to the nesting level. Nodes of the first level aren't indented at all. Nodes on the second level are indented by 5 spaces, nodes on the third level are indented by 10 spaces, and so on. The variable *indentationLevel* keeps track of the nesting level and is used to specify the indentation of the corresponding node. It's increased by 5 when we start scanning a new subordinate node and decreased by the same amount when we return to the next level up. The *indentationLevel* variable is declared as Static so that it maintains its value between calls.

Run the TreeViewScan project and expand all nodes. Then click the Scan Tree button to populate the list on the right with the names of the continents/countries/cities. Obviously, the ListBox control is not a substitute for the TreeView control. The data have no particular structure; even when the names are indented, there are no tree lines connecting the nodes, and users can't expand and collapse the control's contents.

The ListView Control

The ListView control is similar to the ListBox control except that it can display its items in many forms, along with any number of subitems for each item. To use the ListView control in your project, place an instance of the control on a form and then set its basic properties, which are described in the following list.

View* and *Arrange Two properties determine how the various items will be displayed on the control: the `View` property, which determines the general appearance of the items, and the `Arrange` property, which determines the alignment of the items on the control's surface. The `View` property can have one of the values shown in Table 9.1.

The `Arrange` property can have one of the settings shown in Table 9.2.

HeaderStyle This property determines the style of the headers in *Details* view. It has no meaning when the `View` property is set to anything else, because only the Details view has columns. The possible settings of the `HeaderStyle` property are shown in Table 9.3.

AllowColumnReorder This property is a True/False value that determines whether the user can reorder the columns at runtime, and it's meaningful only in Details view. If this property is set to True, the user can move a column to a new location by dragging its header with the mouse and dropping it in the place of another column.

Activation This property, which specifies how items are activated with the mouse, can have one of the values shown in Table 9.4.

Table 9.1: Settings of the *View* Property

Setting	Description
LargeIcon	(Default) Each item is represented by an icon and a caption below the icon.
SmallIcon	Each item is represented by a small icon and a caption that appears to the right of the icon.
List	Each item is represented by a caption.
Details	Each item is displayed in a column with its subitems in adjacent columns.
Tile	Each item is displayed with an icon and its subitems to the right of the icon. This view is available only on Windows XP and Windows Server 2003.

Table 9.2: Settings of the *Arrange* Property

Setting	Description
Default	When an item is moved on the control, the item remains where it is dropped.
Left	Items are aligned to the left side of the control.
SnapToGrid	Items are aligned to an invisible grid on the control. When the user moves an item, the item moves to the closest grid point on the control.
Top	Items are aligned to the top of the control.

Table 9.3: Settings of the *HeaderStyle* Property

Setting	Description
Clickable	Visible column header that responds to clicking
Nonclickable	(Default) Visible column header that does not respond to clicking
None	No visible column header

Table 9.4: Settings of the *Activation* Property

Setting	Description
OneClick	Items are activated with a single click. When the cursor is over an item, it changes shape, and the color of the item's text changes.
Standard	(Default) Items are activated with a double-click. No change in the selected item's text color takes place.
TwoClick	Items are activated with a double-click, and their text changes color as well.

FullRowSelect This property is a True/False value, indicating whether the user can select an entire row or just the item's text, and it's meaningful only in Details view. When this property is False, only the first item in the selected row is highlighted.

GridLines Another True/False property. If True, grid lines between items and subitems are drawn. This property is meaningful only in Details view.

Group The items of the ListView control can be grouped into categories. To use this feature, you must first define the groups by using the control's `Group` property, which is a collection of strings. You can add as many members to this collection as you want. After that, as you add items to the ListView control, you can specify the group to which they belong. The control will group the items of the same category together and display the group's title above each group. You can easily move items between groups at runtime by setting the corresponding item's `Group` property to the name of the desired group.

LabelEdit The `LabelEdit` property lets you specify whether the user will be allowed to edit the text of the items. The default value of this property is False. Notice that the `LabelEdit` property applies to the item's `Text` property only; you can't edit the subitems (unfortunately, you can't use the ListView control as an editable grid).

MultiSelect A True/False value, indicating whether the user can select multiple items from the control. To select multiple items, click them with the mouse while holding down the Shift or Ctrl key. If the control's `ShowCheckboxes` property is set to True, users can select multiple items by marking the check box in front of the corresponding item(s).

Scrollable A True/False value that determines whether the scroll bars are visible. Even if the scroll bars are invisible, users can still bring any item into view. All they have to do is select an item and then press the arrow keys as many times as needed to scroll the desired item into view.

Sorting This property determines how the items will be sorted, and its setting can be None, Ascending, or Descending. To sort the items of the control, call the `Sort` method, which sorts the items according to their caption. It's also possible to sort the items according to any of their subitems, as explained in the section ''Sorting the ListView Control'' later in this chapter.

The *Columns* Collection

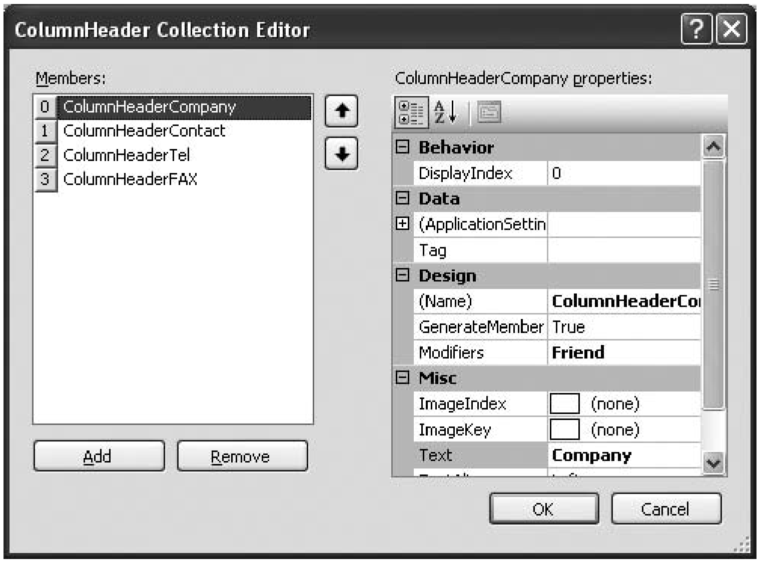
To display items in Details view, you must first set up the appropriate columns. The first column corresponds to the item's caption, and the following columns correspond to its subitems. If you don't set up at least one column, no items will be displayed in Details view. Conversely, the `Columns` collection is meaningful only when the ListView control is used in Details view.

The items of the `Columns` collection are of the ColumnHeader type. The simplest way to set up the appropriate columns is to do so at design time by using a visual tool. Locate and select the `Columns` property in the Properties window, and click the ellipsis button next to the property. The ColumnHeader Collection Editor dialog box will appear, as shown in Figure 9.10, in which you can add and edit the appropriate columns.

Adding columns to a ListView control and setting their properties through the dialog box shown in Figure 9.10 is quite simple. Don't forget to size the columns according to the data you anticipate storing in them and to set their headers.

It is also possible to manipulate the `Columns` collection from within your code as follows. Create a ColumnHeader object for each column in your code, set its properties, and then add it to the control's `Columns` collection:

```
Dim ListViewCol As New ColumnHeader
ListViewCol.Text = "New Column"
ListViewCol.TextAlign = HorizontalAlignment.Center
ListViewCol.Width = 125
ListView1.Columns.Add(ListViewCol)
```


ColumnHeader Collection Editor
Members:
ColumnHeaderCompany properties:
Add
Remove
OK
Cancel

Members:

0	ColumnHeaderCompany
1	ColumnHeaderContact
2	ColumnHeaderTel
3	ColumnHeaderFAX

ColumnHeaderCompany properties:

Behavior	
DisplayIndex	0
Data	
(ApplicationSettin	
Tag	
Design	
(Name)	**ColumnHeaderCo**
GenerateMember	True
Modifiers	**Friend**
Misc	
ImageIndex	(none)
ImageKey	(none)
Text	**Company**

FIGURE 9.10
The ColumnHeader Collection Editor dialog box

ADDING AND REMOVING COLUMNS AT RUNTIME

To add a new column to the control, use the Add method of the Columns collection. The syntax of the Add method is as follows:

```
ListView1.Columns.Add(header, width, textAlign)
```

The *header* argument is the column's header (the string that appears on top of the items). The *width* argument is the column's width in pixels, and the last argument determines how the text will be aligned. The *textAlign* argument can be *Center*, *Left*, or *Right*.

The Add method returns a ColumnHeader object, which you can use later in your code to manipulate the corresponding column. The ColumnHeader object exposes a Name property, which can't be set with the Add method:

```
Header1 = TreeView1.Add( _
           "Column 1", 60, ColAlignment.Left)
Header1.Name = "Column1"
```

After the execution of these statements, the first column can be accessed not only by index, but also by name.

To remove a column, call the Remove method:

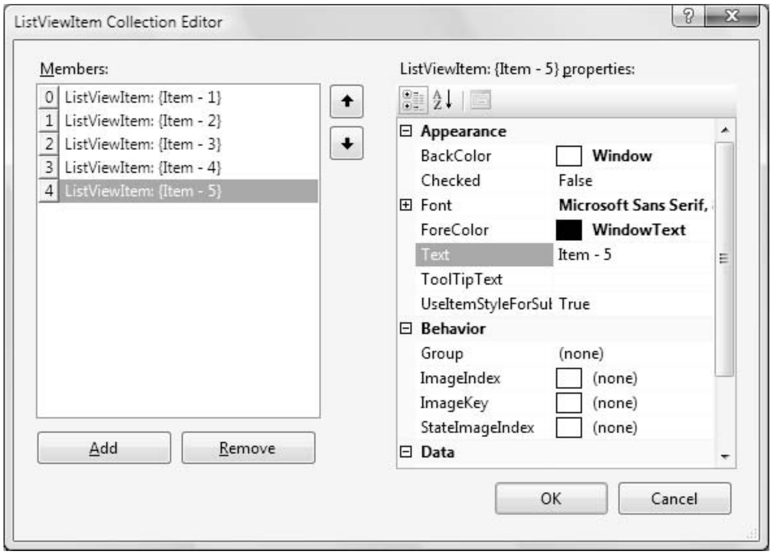
```
ListView1.Columns(3).Remove
```

The indices of the following columns are automatically decreased by one. The `Clear` method removes all columns from the `Columns` collection. Like all collections, the `Columns` collection exposes the `Count` property, which returns the number of columns in the control.

ListView Items and Subitems

As with the TreeView control, the ListView control can be populated either at design time or at runtime. To add items at design time, click the ellipsis button next to the `ListItems` property in the Properties window. When the ListViewItem Collection Editor dialog box pops up, you can enter the items, including their subitems, as shown in Figure 9.11.

FIGURE 9.11
The ListViewItem Collection Editor dialog box



ListViewItem Collection Editor

Members:

0 ListViewItem: {Item - 1}
1 ListViewItem: {Item - 2}
2 ListViewItem: {Item - 3}
3 ListViewItem: {Item - 4}
4 ListViewItem: {Item - 5}

ListViewItem: {Item - 5} properties:

Appearance	
BackColor	Window
Checked	False
Font	Microsoft Sans Serif,
ForeColor	WindowText
Text	Item - 5
ToolTipText	
UseItemStyleForSul	True
Behavior	
Group	(none)
ImageIndex	(none)
ImageKey	(none)
StateImageIndex	(none)
Data	

Click the Add button to add a new item. Each item has subitems, which you can specify as members of the `SubItems` collection. To add an item with three subitems, you must populate the item's `SubItems` collection with the appropriate elements. Click the ellipsis button next to the `SubItems` property in the ListViewItem Collection Editor; the ListViewSubItem Collection Editor will appear. This dialog box is similar to the ListViewItem Collection Editor dialog box, and you can add each item's subitems. Assuming that you have added the item called `Item 1` in the ListViewItem Collection Editor, you can add these subitems: `Item 1-a`, `Item 1-b`, and `Item 1-c`. The first subitem (the one with zero index) is actually the main item of the control.

Notice that you can set other properties such as the color and font for each item, the check box in front of the item that indicates whether the item is selected, and the image of the item. Use this window to experiment with the appearance of the control and the placement of the items, especially in Details view because subitems are visible only in this view. Even then, you won't see anything unless you specify headers for the columns. Note that you can add more subitems than there are columns in the control. Some of the subitems will remain invisible.

Unlike the TreeView control, the ListView control allows you to specify a different appearance for each item and each subitem. To set the appearance of the items, use the `Font`, `BackColor`, and `ForeColor` properties of the ListViewItem object.

Almost all ListView controls are populated at runtime. Not only that, but you should be able to add and remove items during the course of the application. The items of the ListView control are of the ListViewItem type, and they expose members that allow you to control the appearance of the items on the control. These members are as follows:

***BackColor/ForeColor* properties** These properties set or return the background/foreground colors of the current item or subitem.

***Checked* property** This property controls the status of an item. If it's True, the item has been selected. You can also select an item from within your code by setting its `Checked` property to True. The check boxes in front of each item won't be visible unless you set the control's `ShowCheckBoxes` property to True.

***Font* property** This property sets the font of the current item. Subitems can be displayed in a different font if you specify one by using the `Font` property of the corresponding subitem (see the section titled ''The SubItems Collection,'' later in this chapter). By default, subitems inherit the style of the basic item. To use a different style for the subitems, set the item's `UseItemStyleForSubItems` property to False.

***Text* property** This property indicates the caption of the current item or subitem.

***SubItems* collection** This property holds the subitems of a ListViewItem. To retrieve a specific subitem, use a statement such as the following:

```
sitem = ListView1.Items(idx1).SubItems(idx2)
```

where *`idx1`* is the index of the item, and *`idx2`* is the index of the desired subitem.*

To add a new subitem to the `SubItems` collection, use the `Add` method, passing the text of the subitem as an argument:

```
LItem.SubItems.Add("subitem's caption")
```

The argument of the `Add` method can also be a ListViewItem object. Create a ListViewItem, populate it, and then add it to the `Items` collection as shown here:

```
Dim LI As New ListViewItem
LI.Text = "A New Item"
Li.SubItems.Add("Its  first subitem")
Li.SubItems.Add("Its  second subitem")
' statements to add more subitems
ListView1.Items.Add(LI)
```

If you want to add a subitem at a specific location, use the `Insert` method. The `Insert` method of the `SubItems` collection accepts two arguments: the index of the subitem before which the new subitem will be inserted, and a string or ListViewItem to be inserted:

```
LItem.SubItems.Insert(idx, subitem)
```

Like the ListViewItem objects, each subitem can have its own font, which is set with the `Font` property.

The items of the ListView control can be accessed through the `Items` property, which is a collection. As such, it exposes the standard members of a collection, which are described in the following section. Its item has a `SubItems` collection that contains all the subitems of the corresponding item.

The *Items* Collection

All the items on the ListView control form a collection: the `Items` collection. This collection exposes the typical members of a collection that let you manipulate the control's items. These members are discussed next.

***Add* method** This method adds a new item to the `Items` collection. The syntax of the `Add` method is as follows:

```
ListView1.Items.Add(caption)
```

You can also specify the index of the image to be used, along with the item and a collection of subitems to be appended to the new item, by using the following form of the `Add` method:

```
ListView1.Items.Add(caption, imageIndex)
```

where *`imageIndex`* is the index of the desired image on the associated ImageList control.

Finally, you can create a ListViewItem object in your code and then add it to the ListView control by using the following form of the `Add` method:

```
ListView1.Items.Add(listItemObj)
```

The following statements create a new item, set its individual subitems, and then add the newly created ListViewItem object to the control:

```
LItem.Text = "new item"
LItem.SubItems.Add("sub item 1a")
LItem.SubItems.Add("sub item 1b")
LItem.SubItems.Add("sub item 1c")
ListView1.Items.Add(LItem)
```

***Count* property** Returns the number of items in the collection.

***Item* property** Retrieves an item specified by an index value.

***Clear* method** Removes all the items from the collection.

***Remove* method** Removes an item from the collection.

The *SubItems* Collection

Each item in the ListView control may have one or more subitems. You can think of the item as the key of a record, and the subitems as the other fields of the record. The subitems are displayed only in Details mode, but they are available to your code in any view. For example, you can display all items as icons, and when the user clicks an icon, show the values of the selected item's subitems on other controls.

To access the subitems of a given item, use its SubItems collection. The following statements add an item and three subitems to the *ListView1* control:

```
Dim LItem As ListViewItem
LItem = ListView1.Items.Add("Alfred's Futterkiste")
LItem.SubItems.Add("Maria Anders")
LItem.SubItems.Add("030-0074321")
LItem.SubItems.Add("030-0076545")
```

To access the SubItems collection, you need a reference to the item to which the subitems belong. The Add method returns a reference to the newly added item, the LItem variable, which is then used to access the item's subitems, as shown in the preceding code segment.

Displaying the subitems on the control requires some overhead. Subitems are displayed only in Details view mode. However, setting the View property to Details is not enough. You must first create the columns of the Details view, as explained earlier. The ListView control displays only as many subitems as there are columns in the control. The first column, with the header Company, displays the items of the list. The following columns display the subitems. Moreover, you can't specify which subitem will be displayed under each header. The first subitem (Maria Anders in the preceding example) will be displayed under the second header, the second subitem (030-0074321 in the same example) will be displayed under the third header, and so on. At runtime, the user can rearrange the columns by dragging them with the mouse. To disable the rearrangement of the columns at runtime, set the control's AllowColumnReorder property to False (its default value is True).

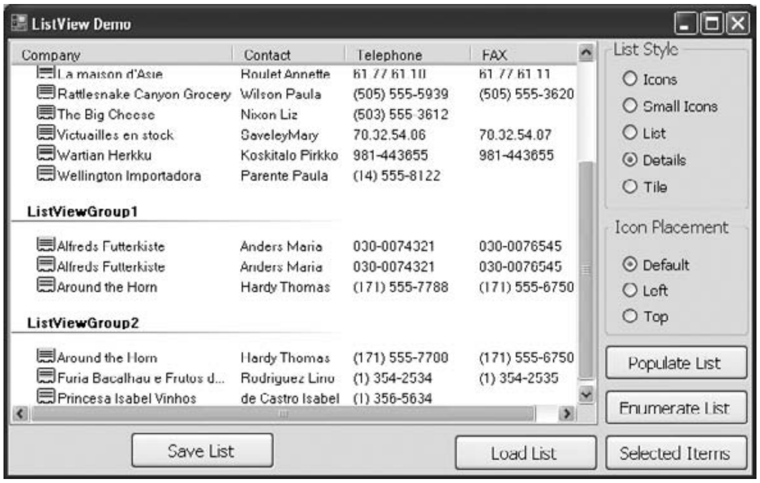
Unless you set up each column's width, they will all have the same width. The width of individual columns is specified in pixels, and you can set it to a percentage of the total width of the control, especially if the control is docked to the form. The following code sets up a ListView control with four headers, all having the same width:

```
Dim LWidth As Integer
LWidth = ListView1.Width - 5
ListView1.ColumnHeaders.Add("Company", LWidth / 4)
ListView1.ColumnHeaders.Add("Contact", LWidth / 4)
ListView1.ColumnHeaders.Add("Phone", LWidth / 4)
ListView1.ColumnHeaders.Add("FAX", LWidth / 4)
ListView1.View = DetailsView
```

This subroutine sets up four headers of equal width. The first header corresponds to the item (not a subitem). The number of headers you set up must be equal to the number of subitems you want to display on the control, plus one. The constant 5 is subtracted to compensate for the width of the column separators. If the control is anchored to the vertical edges of the form, you must execute these statements from within the form's Resize event handler, so that the columns are resized automatically as the control is resized.

VB 2008 at Work: The ListViewDemo Project

Let's put together the members of the ListView control to create a sample application that populates the control and enumerates its items. The sample application of this section is the ListViewDemo project. The application's form, shown in Figure 9.12, contains a ListView control whose items can be displayed in all possible views, depending on the status of the RadioButton controls in the List Style section on the right side of the form.



Company	Contact	Telephone	FAX
La maison d'Asie	Roulet Annette	61 77 61 10	61 77 61 11
Rattlesnake Canyon Grocery	Wilson Paula	(505) 555-5939	(505) 555-3620
The Big Cheese	Nixon Liz	(503) 555-3612	
Victuailles en stock	SaveleyMary	70.32.54.06	70.32.54.07
Wartian Herkku	Koskitalo Pirkko	981-443655	981-443655
Wellington Importadora	Parente Paula	(14) 555-8122	
ListViewGroup1			
Alfreds Futterkiste	Anders Maria	030-0074321	030-0076545
Alfreds Futterkiste	Anders Maria	030-0074321	030-0076545
Around the Horn	Hardy Thomas	(171) 555-7788	(171) 555-6750
ListViewGroup2			
Around the Horn	Hardy Thomas	(171) 555-7788	(171) 555-6750
Furia Bacalhau e Frutos d...	Rodriguez Lino	(1) 354-2534	(1) 354-2535
Princesa Isabel Vinhos	de Castro Isabel	(1) 356-5634	

FIGURE 9.12
The ListViewDemo project demonstrates the basic members of the ListView control.

The control's headers and their widths were set at design time through the ColumnHeader Collection Editor, as explained earlier. To populate the ListView control, click the Populate List button, whose code is shown next. The code creates a new ListViewItem object for each item to be added. Then it calls the `Add` method of the `SubItems` collection to add the item's subitems (contact, phone, and fax numbers). After the ListViewItem has been set up, it's added to the control via the `Add` method of its `Items` collection.

Listing 9.12 shows the statements that insert the first two items in the list. The remaining items are added by using similar statements, which need not be repeated here. The sample data I used in the ListViewDemo application came from the Northwind sample database.

LISTING 9.12: Populating a ListView Control

```
Dim LItem As New ListViewItem()
LItem.Text = "Alfred's Futterkiste"
LItem.SubItems.Add("Anders Maria")
LItem.SubItems.Add("030-0074321")
LItem.SubItems.Add("030-0076545")
LItem.ImageIndex = 0
ListView1.Items.Add(LItem)

LItem = New ListViewItem()
LItem.Text = "Around the Horn"
LItem.SubItems.Add("Hardy Thomas")
LItem.SubItems.Add("(171) 555-7788")
LItem.SubItems.Add("(171) 555-6750")
LItem.ImageIndex = 0
ListView1.Items.Add(LItem)
```

Enumerating the List

The Enumerate List button scans all the items in the list and displays them along with their subitems in the Immediate window. To scan the list, you must set up a loop that enumerates all the items in the `Items` collection. For each item in the list, set up a nested loop that scans all the subitems of the current item. The complete code for the Enumerate List button is shown in Listing 9.13.

Listing 9.13: Enumerating Items and SubItems

```
Private Sub bttnEnumerate_Click(...) _
            Handles bttnEnumerate.Click
   Dim i, j As Integer
   Dim LItem As ListViewItem
   For i = 0 To ListView1.Items.Count - 1
      LItem = ListView1.Items(i)
      Debug.WriteLine(LItem.Text)
      For j = 0 To LItem.SubItems.Count - 1
         Debug.WriteLine("   " & ListView1.Columns(j).Text & _
                           "  " & Litem.SubItems(j).Text)
      Next
   Next
End Sub
```

Notice that each item may have a different number of subitems. The output of this code in the Immediate window is shown next. The subitems appear under the corresponding item, and they are indented by three spaces:

```
Alfred's Futterkiste
   Company  Alfred's Futterkiste
   Contact  Anders Maria
   Telephone  030-0074321
   FAX  030-0076545
Around the Horn
   Company  Around the Horn
   Contact  Hardy Thomas
   Telephone  (171) 555-7788
   FAX  (171) 555-6750
```

The code in Listing 9.13 uses a `For...Next` loop to iterate through the items of the control. You can also set up a `For Each...Next` loop, as shown here:

```
Dim LI As ListViewItem
For Each LI In ListView1.Items
   { access the current item through the LI variable}
Next
```

Sorting the ListView Control

The ListView control provides the Sort method, which sorts the list's items, and the Sorting property, which determines how the items will be sorted. The Sort method sorts the items in the first column alphabetically. Each item may contain any number of subitems, and you should be able to sort the list according to any column. The values stored in the subitems can represent different data types (numeric values, strings, dates, and so on), but the control doesn't provide a default sorting mechanism for all data types. Instead, it uses a custom comparer object, which you supply, to sort the items. (The topic of building custom comparers is discussed in detail in Chapter 14, ''Storing Data in Collections.'') A *custom comparer* is a function that compares two items and returns an integer value (–1, 0, or 1) that indicates the order of the two items. After this function is in place, the control uses it to sort its items.

The ListView control's ListViewItemSorter property accepts the name of a custom comparer, and the items on the control are sorted according to the custom comparer as soon as you call the Sort method. You can provide several custom comparers and sort the items in many different ways. If you plan to display subitems along with your items in Details view, you should make the list sortable by any column. It's customary for a ListView control to sort its items according to the values in a specific column each time the header of this column is clicked. And this is exactly the type of functionality you'll add to the ListViewDemo project in this section.

The ListViewDemo control displays contact information. The items are company names, and the first subitem under each item is the name of a contact. We'll create two custom comparers to sort the list according to either company name or contact. The two methods are identical because they compare strings, but it's not any more complicated to compare dates, distances, and so on.

Let's start with the two custom comparers. Each comparer must be implemented in its own class, and you assign the name of the custom comparer to the ListViewItem property of the control. Listing 9.14 shows the ListCompanyComparer and ListContactComparer classes.

LISTING 9.14: The Two Custom Comparers for the ListViewDemo Project

```
Class ListCompanySorter
   Implements IComparer
   Public Function CompareTo(ByVal o1 As Object, _
                   ByVal o2 As Object) As Integer _
                   Implements System.Collections.IComparer.Compare
      Dim item1, item2 As ListViewItem
      item1 = CType(o1, ListViewItem)
      item2 = CType(o2, ListViewItem)
      If item1.ToString.ToUpper > item2.ToString.ToUpper Then
         Return 1
      Else
         If item1.ToString.ToUpper < item2.ToString.ToUpper Then
            Return -1
         Else
            Return 0
         End If
      End If
   End Function
End Class
```

```
Class ListContactSorter
   Implements IComparer
   Public Function CompareTo(ByVal o1 As Object, _
                   ByVal o2 As Object) As Integer _
                   Implements System.collections.IComparer.Compare
      Dim item1, item2 As ListVewItem
      item1 = CType(o1, ListViewItem)
      item2 = CType(o2, ListViewItem)
      If item1.SubItems(1).ToString.ToUpper > _
                   item2.SubItems(1).ToString.ToUpper Then
         Return 1
      Else
         If item1.SubItems(1).ToString.ToUpper < _
                   item2.SubItems(1).ToString.ToUpper Then
            Return -1
         Else
            Return 0
         End If
      End If
   End Function
End Class
```

The code is straightforward. If you need additional information, see the discussion of the IComparer interface in Chapter 14. The two functions are identical, except that the first one sorts according to the item, and the second one sorts according to the first subitem.

To test the custom comparers, you simply assign their names to the `ListViewItemSorter` property of the ListView control. To take advantage of our custom comparers, we must write some code that intercepts the clicks on the control's headers and calls the appropriate comparer. The ListView control fires the `ColumnClick` event each time a column header is clicked. This event handler reports the index of the column that was clicked through the `e.Column` property, and we can use this argument in our code to sort the items accordingly. Listing 9.15 shows the event handler for the `ColumnClick` event.

LISTING 9.15: The ListView Control's *ColumnClick* Event Handler

```
Public Sub ListView1_ColumnClick(...) _
           Handles ListView1.ColumnClick
   Select Case e.column
      Case 0
         ListView1.ListViewItemSorter = New ListCompanySorter()
         ListView1.Sorting = SortOrder.Ascending
      Case 1
         ListView1.LisViewtItemSorter = New ListContactSorter()
         ListView1.Sorting = SortOrder.Ascending
    End Select
End Sub
```

Processing Selected Items

The user can select multiple items from a ListView control by default. Even though you can display a check mark in front of each item, it's not customary. Multiple items in a ListView control are selected with the mouse while holding down the Ctrl or Shift key.

The selected items form the `SelectedListItemCollection`, which is a property of the control. You can iterate through this collection with a `For...Next` loop or through the enumerator object exposed by the collection. In the following example, I use a `For Each...Next` loop. Listing 9.16 is the code behind the Selected Items button of the ListViewDemo project. It goes through the selected items and displays each one of them, along with its subitems, in the Output window. Notice that you can select multiple items in any view, even when the subitems are not visible. They're still there, however, and they can be retrieved through the `SubItems` collection.

Listing 9.16: Iterating the Selected Items on a ListView Control

```
Private Sub bttnIterate_Click(...) _
            Handles bttnIterate.Click
   Dim LItem As ListViewItem
   Dim LItems As ListView.SelectedListViewItemCollection
   LItems = ListView1.SelectedItems
   For Each LItem In LItems
      Debug.Write(LItem.Text & vbTab)
      Debug.Write(LItem.SubItems(0).ToString & vbTab)
      Debug.Write(LItem.SubItems(1).ToString & vbTab)
      Debug.WriteLine(LItem.SubItems(2).ToString & vbTab)
   Next
End Sub
```

Real World Scenario

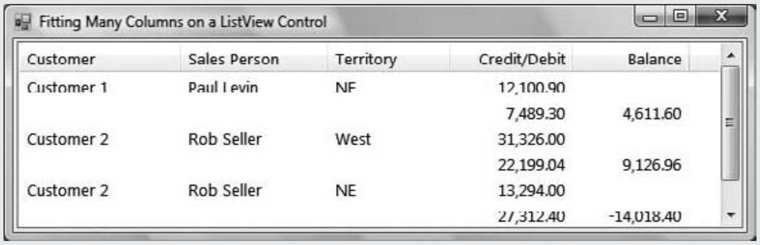
Fitting More Data into a ListView Control

A fairly common problem in designing practical user interfaces with the ListView control is how to display more columns than can be viewed in a reasonably sized window. This is especially true for accounting applications, which may have several debit/credit/balance columns. It's typical to display these values for the previous period, the current period, and then the totals, or to display the period values along with the corresponding values of the previous year, year-to-date values, and so on.

The first approach is to use a smaller font, but this won't take you far. A more-practical approach is to use two (or even more) rows on the control for displaying a single row of data. For example, you can display credit and debit data in two rows, as shown in the following figure. This arrangement saves you the space of one column on the screen. You could even display the balance on a third row and use different colors. The auxiliary rows, which are introduced to accommodate more data on the control, could have a different background color too.

Adding auxiliary columns is straightforward; just add an empty string for the cells that don't change values, because all rows must have the same structure. The first two rows of the ListView control in the preceding screen capture were added by using the following statements:

```
Dim LI As ListViewItem
LI = ListView1.Items.Add("Customer 1")
LI.SubItems.Add("Paul Levin")
LI.SubItems.Add("NE")
LI.SubItems.Add("12,100.90")
LI = ListView1.Items.Add("")
LI.SubItems.Add("")
LI.SubItems.Add("")
LI.SubItems.Add("7,489.30")
LI.SubItems.Add((12100.9 - 7489.3).ToString("#,###.00"))
```



Customer	Sales Person	Territory	Credit/Debit	Balance
Customer 1	Paul Levin	NE	12,100.90	
			7,489.30	4,611.60
Customer 2	Rob Seller	West	31,326.00	
			22,199.04	9,126.96
Customer 2	Rob Seller	NE	13,294.00	
			27,312.40	-14,018.40

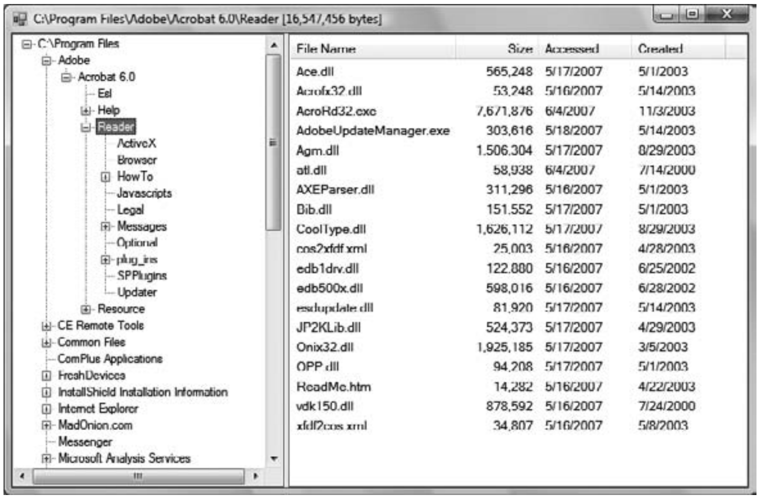
If your code reacts to the selection of an item with the mouse, or the double-click event, you must take into consideration that users may click an auxiliary row. The following `If` structure in the control's `SelectedIndexChanged` event handler prints the item's text, no matter which of the two rows of an item are selected on the control:

```
If ListView1.SelectedItems.Count = 0 Then Exit Sub
Dim idx As Integer
If ListView1.SelectedItems(0).Index Mod 2 <> 0 Then
    idx = ListView1.SelectedItems(0).Index - 1
Else
    idx = ListView1.SelectedItems(0).Index
End If
Debug.WriteLine(ListView1.Items(idx).Text)
```

VB 2008 at Work: The CustomExplorer Project

The last example in this chapter combines the TreeView and ListView controls. It's a fairly advanced example, but I included it here for the most ambitious readers. It can also be used as the starting point for many custom applications, so give it a try. You can always come back to this project after you've mastered other aspects of the Framework, such as the FileIO namespace.

The CustomExplorer project, shown in Figure 9.13, displays a structured list of folders in the left pane, and a list of files in the selected folder in the right pane. The left pane is populated when the application starts, and it might take a while. On my Pentium system, it takes nearly 30 seconds to populate the TreeView control with the structure of the Windows folder (which includes FallBack folders and three versions of the Framework; more than 50,000 files in 1,700 folders in all). You can expand any folder in this pane and view its subfolders. To view the files in a folder, click the folder name, and the right pane will be populated with the names of the selected folder's files, along with other data, such as the file size, date of creation, and date of last modification. You haven't seen the classes for accessing folders and files yet, but you shouldn't have a problem following the code. If you have to, you can review the members of the IO namespace in Chapter 15, in which I discuss in detail the same project's code.



C:\Program Files\Adobe\Acrobat 6.0\Reader [16,547,456 bytes]

```
C:\Program Files
  Adobe
    Acrobat 6.0
      Esl
      Help
      Reader
        ActiveX
        Browser
        HowTo
        Javascripts
        Legal
        Messages
        Optional
        plug_ins
        SPPlugins
        Updater
      Resource
  CE Remote Tools
  Common Files
  ComPlus Applications
  FreshDevices
  InstallShield Installation Information
  Internet Explorer
  MadOnion.com
  Messenger
  Microsoft Analysis Services
```

File Name	Size	Accessed	Created
Ace.dll	565,248	5/17/2007	5/1/2003
Acrofx32.dll	53,248	5/16/2007	5/14/2003
AcroRd32.exe	7,671,876	6/4/2007	11/3/2003
AdobeUpdateManager.exe	303,616	5/18/2007	5/14/2003
Agm.dll	1.506.304	5/17/2007	8/29/2003
atl.dll	58,938	6/4/2007	7/14/2000
AXEParser.dll	311,296	5/16/2007	5/1/2003
Bib.dll	151.552	5/17/2007	5/1/2003
CoolType.dll	1,626,112	5/17/2007	8/29/2003
cos2xfdf.xml	25,003	5/16/2007	4/28/2003
edb1drv.dll	122.880	5/16/2007	6/25/2002
edb500x.dll	598,016	5/16/2007	6/28/2002
esdupdate.dll	81,920	5/17/2007	5/14/2003
JP2KLib.dll	524,373	5/17/2007	4/29/2003
Onix32.dll	1,925,185	5/17/2007	3/5/2003
OPP.dll	94,208	5/17/2007	5/1/2003
ReadMe.htm	14,282	5/16/2007	4/22/2003
vdk150.dll	878,592	5/16/2007	7/24/2000
xfdf2cos.xml	34,807	5/16/2007	5/8/2003

FIGURE 9.13
The CustomExplorer project demonstrates how to combine a TreeView and a ListView control on the same form.

This section's project is not limited to displaying folders and files; you can populate the two controls with data from several sources. For example, you can display customers in the left pane (and organize them by city or state) and display their related data, such as invoices and payments, in the right pane. Or you can populate the left pane with product names, and the right pane with the respective sales. In general, you can use the project as an interface for many types of applications. You can even use it as a custom Explorer to add features that are specific to your applications.

The TreeView control on the left pane is populated from within the Form's `Load` event handler subroutine with the subfolders of the `C:\Program Files` folder:

```
Dim Nd As New TreeNode()
Nd = TreeView1.Nodes.Add("C:\Program Files")
ScanFolder("c:\Program Files", ND)
```

The first argument is the name of the folder to be scanned, and the second argument is the root node, under which the entire tree of the specified folder will appear. To populate the control with the files of another folder or drive, change the name of the path accordingly. The code is short, and all the work is done by the `ScanFolder()` subroutine. The `ScanFolder()` subroutine, which is a short recursive procedure that scans all the folders under a specific folder, is shown in Listing 9.17.

LISTING 9.17: The *ScanFolder()* Subroutine

```
Sub ScanFolder(ByVal folderSpec As String, _
               ByRef currentNode As TreeNode)
```

```
        Dim thisFolder As FileIO.Folder
        Dim allFolders As FileIO.FolderCollection
        allFolders = My.Computer.FileSystem. _
                           GetFolder(folderSpec).FindSubFolders("*.*")
        For Each thisFolder In allFolders
            Dim Nd As TreeNode
            Nd = New TreeNode(thisFolder.FolderName)
            currentNode.Nodes.Add(Nd)
            folderSpec = thisFolder.FolderPath
            ScanFolder(folderSpec, Nd)
            Me.Text = "Scanning " & folderSpec
            Me.Refresh()
        Next
    End Sub
```

The variable `FolderSpec` represents the current folder (the one passed to the `ScanFolder()` subroutine as an argument). The code creates the *`allFolders`* collection, which contains all the subfolders of the current folder. Then it scans every folder in this collection and adds its name to the TreeView control. After adding a folder's name to the TreeView control, the procedure must scan the subfolders of the current folder. It does so by calling itself and passing another folder's name as an argument.

Notice that the `ScanFolder()`subroutine doesn't simply scan a folder. It also adds a node to the TreeView control for each new folder it runs into. That's why it accepts two arguments: the name of the current folder and the node that represents this folder on the control. All folders are placed under their parent folder, and the structure of the tree represents the structure of your hard disk (or the section of the hard disk you're mapping on the TreeView control). All this is done with a small recursive subroutine: the `ScanFolder()` subroutine.

Viewing a Folder's Files

To view the files of a folder, click the folder's name in the TreeView control. As explained earlier, the action of the selection of a new node is detected with the `AfterSelect` event. The code in this event handler, shown in Listing 9.18, displays the selected folder's files on the ListView control.

Listing 9.18: Displaying a Folder's Files

```
    Private Sub TreeView1_AfterSelect(...) _
                Handles TreeView1.AfterSelect
        Dim Nd As TreeNode
        Dim pathName As String
        Nd = TreeView1.SelectedNode
        pathName = Nd.FullPath
        ShowFiles(pathName)
    End Sub
```

The `ShowFiles()` subroutine actually displays the filenames, and some of their properties, in the specified folder on the ListView control. Its code is shown in Listing 9.19.

LISTING 9.19: The *ShowFiles()* Subroutine

```
Sub ShowFiles(ByVal selFolder As String)
    ListView1.Items.Clear()
    Dim files As FileIO.FileCollection
    Dim file As FileIO.File
    files = My.Computer.FileSystem.GetFolder(selFolder).FindFiles("*.*")
    Dim TotalSize As Long
    For Each file In files
        Dim LItem As New ListViewItem
        LItem.Text = file.FileName
        LItem.SubItems.Add(file.Size.ToString("#,###"))
        LItem.SubItems.Add( _
                    FormatDateTime(file.CreatedTime, _
                    DateFormat.ShortDate))
        Item.SubItems.Add( _
                    FormatDateTime(file.AccessedTime, _
                    DateFormat.ShortDate))
        ListView1.Items.Add(LItem)
        TotalSize += file.Size
    Next
    Me.Text = Me.Text & " [" & TotalSize.ToString("#,###") & " bytes]"
End Sub
```

The `ShowFiles()`subroutine creates a ListItem for each file. The item's caption is the file's name, the first subitem is the file's length, and the other two subitems are the file's creation and last access times. You can add more subitems, if needed, in your application. The ListView control in this example uses the Details view to display the items. As mentioned earlier, the ListView control will not display any items unless you specify the proper columns through the `Columns` collection. The columns, along with their widths and captions, were set at design time through the ColumnHeader Collection Editor.

Additional Topics

The discussion of the CustomExplorer sample project concludes the presentation of the TreeView and ListView controls. However, there are a few more interesting topics you might like to read about, which weren't included in this chapter. Like all Windows controls, the ListView control doesn't provide a `Print` method, which I think is essential for any application that displays data on this control. In Chapter 20, ''Printing with Visual Basic 2008,'' you will find the code for printing the items of the ListView control. The printout we'll generate will have columns, just like the control, but it will display long cells (items or subitems with long captions) in multiple text lines. Finally, in Chapter 16 you'll learn how to save the nodes of a TreeView control to a disk file between sessions by using a technique known as serialization. In that chapter, you'll find the code behind the Load Nodes and Save Nodes buttons of the Globe project and a thorough explanation of their function.

The Bottom Line

Create and present hierarchical lists by using the TreeView control. The TreeView control is used to display a list of hierarchically structured items. Each item in the TreeView control is represented by a TreeNode object. To access the nodes of the TreeView control, use the `TreeView.Nodes` collection. The nodes under a specific node (in other words, the child nodes) form another collection of Node objects, which you can access by using the expression `TreeView.Nodes(i).Nodes`. The basic property of the Node object is the `Text` property, which stores the node's caption. The Node object exposes properties for manipulating its appearance (its foreground/background color, its font, and so on).

Master It How will you set up a TreeView control with a book's contents at design time?

Create and present lists of structured items by using the ListView control. The ListView control stores a collection of ListViewItem objects, the `Items` collection, and can display them in several modes, as specified by the `View` property. Each ListViewItem object has a `Text` property and the `SubItems` collection. The subitems are not visible at runtime unless you set the control's `View` property to Details and set up the control's `Columns` collection. There must be a column for each subitem you want to display on the control.

Master It How will you set up a ListView control with three columns to display names, emails, and phone numbers at design time?

Master It How would you populate the same control with the same data at runtime?

Chapter 10

Building Custom Classes

Classes are practically synonymous with objects and they're at the very heart of programming with Visual Basic. The controls you use to build the visible interface of your application are objects, and the process of designing forms consists of setting the properties of these objects, mostly with point-and-click operations. The Framework itself is an enormous compendium of classes, and you can import any of them into your applications and use them as if their members were part of the language. You simply declare a variable of the specific class type, initialize it, and then use it in your code.

You have already worked with ListViewItem objects; they're the items that make up the contents of a ListView control. You declare an object of this type, then set its properties, and finally add it to the control's `Items` collection:

```
Dim LI As New ListViewItem
LI.Text = "Item 1"
LI.Font = New Font("Verdana", 12, FontStyle.Regular)
```

`LI` is an object of the ListViewItem type. The `New` keyword creates a new instance of the ListViewItem class; in other words, a new object. The following two statements set the basic properties of the *`LI`* variable. The `Font` property is also an object: it's an instance of the Font class. You can also add a few subitems to the *`LI`* variable and set their properties. When you're finished, you can add the *`LI`* variable to the ListView control:

```
ListView1.Items.Add(LI)
```

Controls are also objects; they differ from other classes in that controls provide a visual interface, whereas variables don't. However, you manipulate all objects by setting their properties and calling their methods.

In this chapter, you'll learn how to do the following:

- Build your own classes
- Use custom classes in your projects
- Customize the usual operators for your classes

Classes and Objects

When you create a variable of any type, you're creating an instance of a class. The variable lets you access the functionality of the class through its properties and methods. Even the base data types are implemented as classes (the System.Integer class, System.Double, and so on). An integer value, such as 3, is an instance of the System.Integer class, and you can call the properties and methods of this class by using its instance. Expressions such as `CDec(3).MinValue` and `#1/1/2000#.Today` are odd but valid. The first expression returns the minimum value you can represent with the Decimal data type, whereas the second expression returns the current date. The DataTime data type exposes the `Today` property, which returns the current date. The expression `#1/1/2000#` is a value of the DataTime type, so you can find out the current date by calling its `Today` property. If you enter either one of the preceding expressions in your code, you'll get a warning, but they will be executed.

Classes are used routinely in developing applications, and you should get into the habit of creating and using custom classes, even with simple projects. In team development, classes are a necessity, because they allow developers to share their work easily. If you're working in a corporate environment, in which different programmers code different parts of an application, you can't afford to repeat work that someone else has already done. You should be able to get their code and use it in your application as is. That's easier said than done, because you can guess what will happen as soon as a small group of programmers start sharing code — they'll end up with dozens of different versions for each function, and every time a developer upgrades a function, he or she will most likely break the applications that were working with the old version. Or each time they revise a function, they must update all the projects by using the old version of the function and test them. It just doesn't work.

The major driving force behind *object-oriented programming (OOP)* is *code reuse*. Classes allow you to write code that can be reused in multiple projects. You already know that classes don't expose their source code. In other words, you can use a class without having access to its code, and therefore you can't affect any other projects that use the class. You also know that classes implement complicated operations and make these operations available to programmers through properties and methods. The Array class exposes a `Sort` method, which sorts its elements. This is not a simple operation, but fortunately you don't have to know anything about sorting. Someone else has done it for you and made this functionality available to your applications. This is called *encapsulation*. Some functionality has been built into the class (or *encapsulated* into the class), and you can access it from within your applications by using a simple method call.

The Framework is made up of thousands of classes, which allow you to access all the functionality of the operating system. You don't have to see the code, and you don't have to know anything about sorting to sort your arrays, just as you don't need to know anything about encryption to encrypt a string by using the System.Security.Cryptography class. In effect, you're reusing code that Microsoft has already written. It is also possible to *extend* these classes by adding custom members, and even *override* existing members. When you extend a class, you create a new class based on an existing one. Projects using the original class will keep seeing the original class, and they will work fine. New projects that see the derived class will also work.

What Is a Class?

A *class* is a program that doesn't run on its own; it's a collection of methods that must be used by another application. We exploit the functionality of the class by creating a variable of the same type as the class and then call the class's properties and methods through this variable. The methods and properties of the class, as well as its events, constitute the class's *interface*. It's not a visible interface, like the ones you've learned to design so far, because the class doesn't interact directly with the user. To interact with the class, the application uses the class's interface, just as users will be interacting with your application through its visual interface.

You have already learned how to use classes. Now is the time to understand what goes on behind the scenes when you interact with a class and its members. Behind every object, there's a class. When you declare an array, you're invoking the System.Array class, which contains all the code for manipulating arrays. Even when you declare a simple integer variable, you're invoking a class: the System.Integer class. This class contains the code that implements the various properties (such as `MinValue` and `MaxValue`) and methods (such as `ToString`) of the Integer data type. The first time you use an object in your code, you're *instantiating* the class that implements this object. The class's code is loaded into memory, initializes its variables, and is ready to execute. The image of the class in memory is said to be an *instance* of the class, and this is an object.

CLASSES VERSUS OBJECTS

Two of the most misused terms in OOP are *object* and *class*, and most people use them interchangeably. You should think of the class as the factory that produces objects. There's only one System.Array class, but you can declare any number of arrays in your code. Every array is an instance of the System.Array class. All arrays in an application are implemented by the same code, but they store different data. Each instance of a class is nothing more than a set of variables: the same code acts on different sets of variables; each set of variables is a separate instance of the class.

Consider three TextBox controls on the same form. They are all instances of the System.Windows.Forms.TextBox class, but changing any property of a TextBox control doesn't affect the other two controls. Classes are the blueprints on which objects are based. We use the same blueprint to build multiple buildings with the same structural characteristics, but different properties (wall colors, doors, and so on).

Objects are similar to Windows controls, except that they don't have a visible interface. Controls are instantiated when you place them on a form; classes are instantiated when you use a variable of the same type — not when you declare the variable by using the `Dim` statement. To use a control, you must make it part of the project by adding its icon to the Toolbox, if it's not already there. To use a class in your code, you must import the file that implements the class. (This is a Dynamic Link Library, or DLL, file.) To manipulate a control from within your code, you call its properties and methods. You do the same with classes. Finally, you program the various events raised by the controls to interact with the users of your applications. Most classes don't expose any events because the user can't interact with them, but some classes do raise events, which you can program just as you program the events of Windows controls.

Classes Combine Code with Data

Another way to view classes is to understand how they combine code and data. This simple idea is the very essence of object-oriented programming. Data is data, and traditional procedural languages allow you to manipulate data in any way. Meaningful data, however, is processed in specific ways.

Let's consider accounting data. You can add or subtract amounts to an account, sum similar accounts (such as training and travel expenses), calculate taxes on certain account amounts, and the like. Other types of processing may not be valid for this type of data. We never multiply the amounts of two different accounts or calculate logarithms of account balances. These types of processing are quite meaningful with different data, but not with accounting data.

Because the data itself determines to a large extent the type of processing that will take place on the data, why not ''package'' the data along with the code for processing? Instead of simply creating structures for storing our data, we also write the code to process them. The data and the code are implemented in a single unit, a class, and the result is an object. After the class has been built, we no longer write code for processing the data; we simply create objects of this type and call their methods. To transfer an amount from one account to another, we call a method that knows how to transfer the amount, and it also makes sure that the amount isn't subtracted from one account unless it has been added to the other account (and vice versa).

To better understand how classes combine code with data, let's take a close look at a class we're all too familiar with, the Array class. The role of the array is to store sets of data. In addition to holding data, the Array class also knows how to process data: how to retrieve an element, how to extract a segment of the array, and even how to sort its elements. All these operations require a substantial amount of code. The mechanics of storing data in the array and the code that implements the properties and the methods of the array are hidden from you, the developer. You can instruct the array to perform certain tasks by using simple statements. When you call the `Sort` method, you're telling the array to execute some code that will sort its elements. As a developer, you don't know how the data are stored in the array, or how the `Sort` method works. Classes abstract many operations by hiding the implementation details, and developers can manipulate arrays by calling methods. An instance of the Array class not only holds the elements that make up an array, but also exposes the most common operation one would perform on arrays as methods. Summing the logarithms of the elements of a numeric array is a specialized operation, and you have to provide the code to implement it on your own. If you type **System.Array**., you will see a list of all operations you can perform on an array.

In the following sections, you'll learn how data and code coexist in a class, and how you can manipulate the data through the properties and methods exposed by the class. In Chapter 3, ''Programming Fundamentals,'' you learned how to create Structures to store data. Classes are similar to Structures, in that they represent custom data structures. In this chapter, we'll take the idea of defining custom data structures one step further, by adding properties and methods for manipulating the custom data, something you can't do with structures. Let's start by building a custom class and then using it in our code.

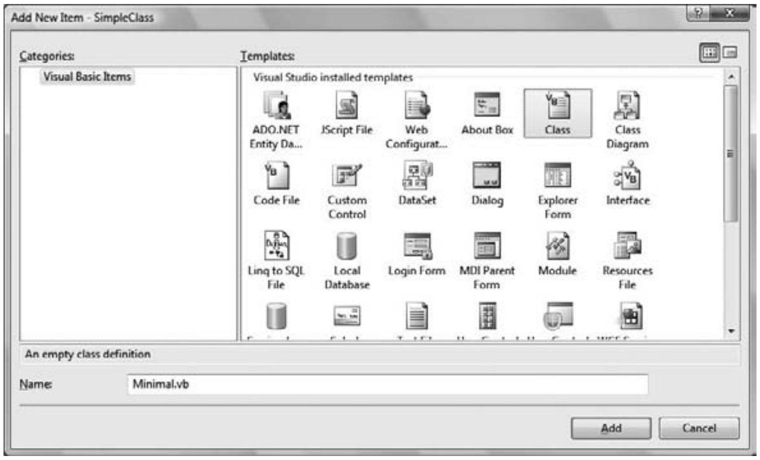
Building the Minimal Class

Our first example is the Minimal class; we'll start with the minimum functionality class and keep adding features to it. The name of the class can be anything — just make sure that it's suggestive of the class's functionality.

A class might reside in the same file as a form, but it's customary to implement custom classes in a separate module, a Class module. You can also create a Class project, which contains one or more classes. However, a class doesn't run on its own, and you can't test it without a form. You can create a Windows application, add the class to it, and then test it by adding the appropriate code to the form. After debugging the class, you can remove the test form and reuse the class with any other project. Because the class is pretty useless outside the context of a Windows application, in this chapter I use Windows applications and add a Class module in the same solution.

Start a new Windows project and name it **SimpleClass** (or open the sample project by that name). Then create a new class by adding a Class item to your project. Right-click the project's name in the Solution Explorer window and choose Add ➢ Class from the context menu. In the dialog box that pops up, select the Class icon and enter a name for the class. Set the class's name to **Minimal**, as shown in Figure 10.1.

FIGURE 10.1
Adding a Class item to a project



The code that implements the class resides in the `Minimal.vb` file, and we'll use the existing form to test our class. After you have tested and finalized the class's code, you no longer need the form and you can remove it from the project.

When you open the class by double-clicking its icon in the Project Explorer window, you will see the following lines in the code window:

```
Public Class Minimal

End Class
```

If you'd rather create a class in the same file as the application's form, enter the `Class` keyword followed by the name of the class, after the existing `End Class` of the form's code window. The editor will insert the matching `End Class` for you. Insert a class's definition in the form's code window if the class is specific to this form only and no other part of the application will use it. At this point, you already have a class, even if it doesn't do anything.

Switch back to the Form Designer, add a button to the test form, and insert the following code in its `Click` event handler:

```
Dim obj1 As Minimal
```

Press Enter and type the name of the variable, `obj1`, followed by a period, on the following line. You will see a list of the methods your class exposes already:

```
Equals
GetHashCode
GetType
ReferenceEqual
ToString
```

If you don't see all of these members, switch to the All Members tab of the IntelliSense drop-down box.

These methods are provided by the Common Language Runtime (CLR), and you don't have to implement them on your own (although you will probably have to provide a new, nongeneric implementation for some of them). They don't expose any real functionality; they simply reflect the way VB handles all classes. To see the kind of functionality that these methods expose, enter the following lines in the Button's `Click` event handler and then run the application:

```
Dim obj1 As New Minimal
Debug.WriteLine(obj1.ToString)
Debug.WriteLine(obj1.GetType)
Debug.WriteLine(obj1.GetHashCode)
Dim obj2 As New Minimal
Debug.WriteLine(obj1.Equals(obj2))
Debug.WriteLine(Minimal.ReferenceEquals(obj1, obj2))
```

The following lines will be printed in the Output window:

```
SimpleClass.Minimal
SimpleClass.Minimal
18796293
False
False
```

The name of the object is the same as its type, which is all the information about your new class that's available to the CLR. Shortly you'll see how you can implement your own `ToString` method and return a more-meaningful string. The hash value of the *`obj1`* variable is an integer value that uniquely identifies the object variable in the context of the current application (it happens to be 18796293, but it is of no consequence).

The next line tells you that two variables of the same type are not equal. But why aren't they equal? We haven't differentiated them at all, yet they're different because they point to two different objects, and the compiler doesn't know how to compare them. All it can do is figure out whether the variables point to the same object. To understand how objects are compared, add the following statement after the line that declares *`obj2`*:

```
obj2 = obj1
```

If you run the application again, the last two statements will print True in the Output window. The `Equals` method compares the two objects and returns a True/False value. Because we haven't told it how to compare two instances of the class yet, it compares their references just like the `ReferenceEquals` method. The `ReferenceEquals` method checks for reference equality; that is, it returns True if both variables point to the same object (the same instance of the class). If you change a property of the *`obj1`* variable, the changes will affect *`obj2`* as well, because both variables point to the same object. We can't modify the object because it doesn't expose any members that we can set to differentiate it from another object of the same type. We'll get to that shortly.

Most classes expose a custom `Equals` method, which knows how to compare two objects of the same type (two objects based on the same class). The custom `Equals` method usually compares the properties of the two instances of the class and returns True if a set of basic properties (or all of them) are the same. You'll learn how to customize the default members of any class later in this chapter.

Notice the name of the class: SimpleClass.Minimal. Within the current project, you can access it as Minimal. Other projects can either import the Minimal class and access it as Minimal, or specify the complete name of the class, which is the name of the project it belongs to followed by the class name.

Adding Code to the Minimal Class

Let's add some functionality to our bare-bones class. We'll begin by adding two trivial properties and two methods to perform simple operations. The two properties are called `strProperty` (a string) and `dblProperty` (a double). To expose these two members as properties, you can simply declare them as Public variables. This isn't the best method of implementing properties, but it really doesn't take more than declaring something as Public to make it available to code outside the class. The following statement exposes the two properties of the class:

```
Public strProperty As String, dblProperty As Double
```

The two methods we'll implement in our sample class are the `ReverseString` and `NegateNumber` methods. The first method reverses the order of the characters in `strProperty` and returns the new string. The `NegateNumber` method returns the negative of `dblProperty`. They're two simple methods that don't accept any arguments; they simply operate on the values of the properties. Methods are exposed as Public procedures (functions or subroutines), just as properties are exposed as Public variables. Enter the function declarations of Listing 10.1 between the `Class Minimal` and `End Class` statements in the class's code window. (I'm showing the entire listing of the class here.)

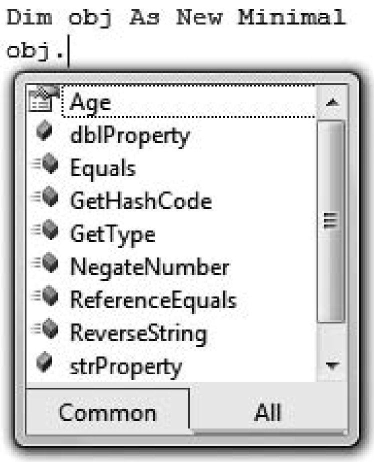
LISTING 10.1: Adding a Few Members to the Minimal Class

```
Public Class Minimal
    Public strProperty As String, dblProperty As Double
    Public Function ReverseString() As String
        Return (StrReverse(strProperty))
    End Function
    Public Function NegateNumber() As Double
        Return (-dblProperty)
    End Function
End Class
```

Let's test the members we've implemented so far. Switch back to your form and enter the lines shown in Listing 10.2 in a new button's `Click` event handler. The ***`obj`*** variable is of the Minimal type and exposes the Public members of the class. You can set and read its properties, and call its methods. In Figure 10.2, you see a few more members than the ones added so far; we'll extend our Minimal class in the following section. Your code doesn't see the class's code, just as it doesn't see any of the built-in classes' code. You trust that the class knows what it is doing and does it right.

LISTING 10.2: Testing the Minimal Class

```
Dim obj As New Minimal
obj.strProperty = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
obj.dblProperty = 999999
Debug.WriteLine(obj.ReverseString)
Debug.WriteLine(obj.NegateNumber)
```



```
Dim obj As New Minimal
obj.
```

FIGURE 10.2
The members of the class are displayed automatically by the IDE, as needed.

Every time you create a new variable of the Minimal type, you're creating a new instance of the Minimal class. The class's code is loaded into memory only the first time you create a variable of this type, but every time you declare another variable of the same type, a new set of variables is created. This is called an *instance of the class*. The code is loaded once, but it can act on different sets of variables. In effect, different instances of the class are nothing more than different sets of local variables.

THE *NEW* KEYWORD

The New keyword tells VB to create a new instance of the Minimal class. If you omit the New keyword, you're telling the compiler that you plan to store an instance of the Minimal class in the `obj` variable, but the class won't be instantiated. All the compiler can do is prevent you from storing an object of any other type in the `obj` variable. You must still initialize the `obj` variable with the New keyword on a separate line:

```
obj = New Minimal
```

It's the New keyword that creates the object in memory. The `obj` variable simply points to this object. If you omit the New keyword, a Null Reference exception will be thrown when the code attempts to use the variable. This means that the variable is Nothing — it hasn't been initialized yet. Even as you work in the editor's window, the name of the variable will be underlined and the following warning will be generated: *Variable 'obj' is used before it has been assigned a value. A null reference exception could result at runtime*. You can compile the code and run it if you want, but everything will proceed as predicted: As soon as the statement that produced the warning is reached, the runtime exception will be thrown.

Using Property Procedures

The `strProperty` and `dblProperty` properties will accept any value, as long as the type is correct and the value of the numeric property is within the acceptable range. But what if the generic properties were meaningful entities, such as email addresses, ages, or zip codes? We should be able to invoke some code to validate the values assigned to each property. To do so, we implement each property as a special type of procedure: the so-called *Property procedure*.

Properties are implemented with a special type of procedure that contains a `Get` and a `Set` section (frequently referred to as the property's *getter* and *setter*, respectively). The `Set` section of the procedure is invoked when the application attempts to set the property's value; the `Get` section is invoked when the application requests the property's value. The value passed to the property is usually validated in the `Set` section and, if valid, is stored to a local variable. The same local variable's value is returned to the application when it requests the property's value, from the property's `Get` section. Listing 10.3 shows what the implementation of an `Age` property would look like.

LISTING 10.3: Implementing Properties with Property Procedures

```
Private m_Age As Integer
Property Age() As Integer
   Get
      Age = m_Age
   End Get
   Set (ByVal value As Integer)
      If value < 0 Or value >= 100 Then
         MsgBox("Age must be positive and less than 100")
      Else
         m_Age = value
      End If
   End Set
End Property
```

`m_Age` is the local variable where the age is stored. When a statement such as the following is executed in the application that uses your class, the `Set` section of the Property procedure is invoked:

```
obj.Age = 39
```

Because the property value is valid, it is stored in the *`m_Age`* local variable. Likewise, when a statement such as the following one is executed, the `Get` section of the Property procedure is invoked, and the value 39 is returned to the application:

```
Debug.WriteLine(obj.Age)
```

The *`value`* argument of the `Set` procedure represents the actual value that the calling code is attempting to assign to the property. The *`m_Age`* variable is declared as private because we don't want any code outside the class to access it directly. The `Age` property is, of course, Public, so that other applications can set it.

FIELDS VERSUS PROPERTIES

Technically, any variables that are declared as Public in a class are called *fields*. Fields behave just like properties in the sense that you can assign values to them and read their values, but there's a critical distinction between fields and properties: When a value is assigned to a field, you can't validate the value from within your code. If the value is of the wrong type, an exception will occur. Properties should be implemented with a Property procedure, so you can validate their values, as you saw in the preceding example. Not only that, but you can set other values from within your code. Consider a class that represents a contract with a starting and ending date. Every time the user changes the starting date, the code can adjust the ending date accordingly (which is something you can't do with fields). If the two dates were implemented as fields, users of the class could potentially specify an ending date prior to the starting date.

Enter the Property procedure for the `Age` property in the Minimal class and then switch to the form to test it. Open the button's `Click` event handler and add the following lines to the existing ones:

```
obj.Age = 39
Debug.WriteLine("after setting the age to 39, age is " & _
                      obj.Age.ToString)
obj.Age = 199
Debug.WriteLine("after setting the age to 199, age is " & _
                       obj.Age.ToString)
```

The value 39 will appear twice in the Output window, which means that the class accepts the value 39. When the third statement is executed, a message box will appear with the error's description:

```
Age must be positive and less than 100
```

The value 39 will appear in the Output window again. The attempt to set the age to 199 failed, so the property retains its previous value.

Throwing Exceptions

Our error-trapping code works fine, but what good is a message box displayed from within a class? As a developer using the Minimal class in your code, you'd rather receive an exception and handle it from within your code. So let's change the implementation of the `Age` property a little. The Property procedure for the `Age` property (Listing 10.4) throws an `InvalidArgument` exception if an attempt is made to assign an invalid value to it. The `InvalidArgument` exception is one of the existing exceptions, and you can reuse it in your code. Later in this chapter, you'll learn how to create and use custom exceptions.

Listing 10.4: Throwing an Exception from within a Property Procedure

```
Private m_Age As Integer
Property Age() As Integer
   Get
      Age = m_Age
   End Get
   Set (ByVal value As Integer)
      If value < 0 Or value >= 100 Then
```

```
            Dim AgeException As New ArgumentException()
            Throw AgeException
         Else
            M_Age = value
         End If
      End Set
   End Property
```

You can test the revised property definition in your application; switch to the test form, and enter the statements of Listing 10.5 in a new button's `Click` event handler. (This is the code behind the Handle Exceptions button on the test form.)

Listing 10.5: Catching the *Age* Property's Exception

```
   Dim obj As New Minimal
   Dim userAge as Integer
   UserAge = InputBox("Please enter your age")
   Try
      obj.Age = userAge
   Catch exc as ArgumentException
      MsgBox("Can't accept your value, " & userAge.ToString & VbCrLf & _
             "Will continue with default value of 30")
      obj.Age = 30
   End Try
```

This is a much better technique for handling errors in your class. The exceptions can be intercepted by the calling application, and developers using your class can write robust applications by handling the exceptions in their code. When you develop custom classes, keep in mind that you can't handle most errors from within your class because you don't know how other developers will use your class.

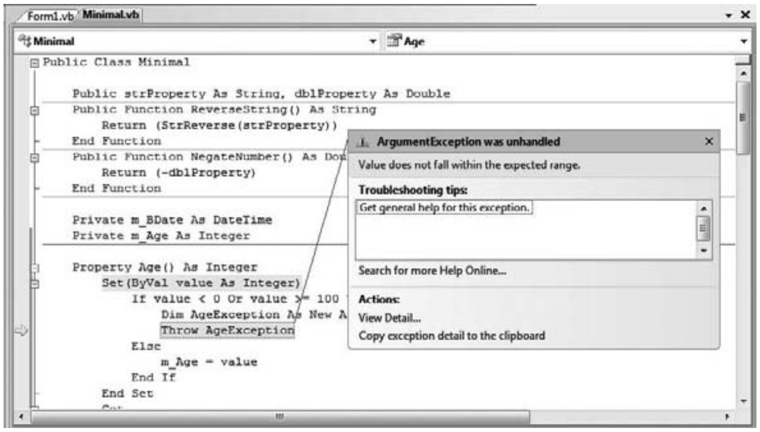
Real World Scenario

Handling Errors in a Class

When you design classes, keep in mind that you don't know how another developer may use them. In fact, you may have to use your own classes in a way that you didn't consider when you designed the class. A typical example is using an existing class with a web application. If your class displays a message box, it will work fine as part of a Windows Forms application. In the context of a web application, however, the message box will be displayed on the monitor of the server that hosts the application, and no one will see it. As a result, the application will keep waiting for a response to a message box before it continues; however, there's no user to click the OK button in the message box, because the code is executing on a server. Even if you don't plan to use a custom class with a web application, never interact with the user from within the class's code. Make your code as robust as you can, but don't hesitate to throw exceptions for all conditions you can't handle from within your code (Figure 10.3). In general, a class's code should detect abnormal conditions, but it shouldn't attempt to remedy them.

The application that uses your class may inform the user about an error condition and give the user a chance to correct the error by entering new data, disable some options on the interface, and so on. As a class developer, you can't make this decision — another developer might prompt the user for another value, and a sloppy developer might let his or her application crash (but this isn't your problem). To throw an exception from within your class's code, call the `Throw` statement with an Exception object as an argument. To play well with the Framework, you should try to use one of the existing exceptions (and the Framework provides quite a few of them). You can also throw custom exceptions by using a statement such as the following:

```
Throw New Exception("your exception's description")
```



```
Public Class Minimal

    Public strProperty As String, dblProperty As Double
    Public Function ReverseString() As String
        Return (StrReverse(strProperty))
    End Function
    Public Function NegateNumber() As Dou
        Return (-dblProperty)
    End Function

    Private m_BDate As DateTime
    Private m_Age As Integer

    Property Age() As Integer
        Set(ByVal value As Integer)
            If value < 0 Or value >= 100
                Dim AgeException As New A
                Throw AgeException
            Else
                m_Age = value
            End If
        End Set
```

ArgumentException was unhandled

Value does not fall within the expected range.

Troubleshooting tips:

Get general help for this exception.

Search for more Help Online...

Actions:

View Detail...

Copy exception detail to the clipboard

FIGURE 10.3
Raising an exception in the class's code

IMPLEMENTING READ-ONLY PROPERTIES

Let's make our class a little more complicated. Age is not usually requested on official documents, because it's valid only for a year after filling out a questionnaire. Instead, you must furnish your date of birth, from which your current age can be calculated at any time. We'll add a `BDate` property in our class and make `Age` a read-only property.

To make a property read-only, you simply declare it as `ReadOnly` and supply the code for the `Get` procedure only. Revise the `Age` property's code in the Minimal class, as seen in Listing 10.6. Then enter the Property procedure from Listing 10.7 for the `BDate` property.

LISTING 10.6: Implementing a Read-Only Property

```
Private m_Age As Integer
ReadOnly Property Age() As Integer
   Get
      Age = m_Age
   End Get
End Property
```

LISTING 10.7: The *BDate* Property

```
Private m_BDate As DateTime
Private m_Age As Integer
Property BDate() As DateTime
    Get
        BDate = m_BDate
    End Get
    Set(ByVal value As Date)
        If Not IsDate(value) Then
            Dim DataTypeException As _
                New Exception("Invalid date value")
            Throw DataTypeException
        End If
        If value > Now() Or _
                DateDiff(DateInterval.Year, value, Now()) >= 100 Then
            Dim AgeException As New Exception _
                     ("Can't accept the birth date you specified")
            Throw AgeException
        Else
            m_BDate = value
            m_Age = DateDiff(DateInterval.Year, value, Now())
        End If
    End Set
End Property
```

As soon as you enter the code for the revised `Age` property, two error messages will appear in the Error List window. The code in the application's form is attempting to set the value of a read-only property, and the editor produces the following error message twice: *Property 'Age' is 'ReadOnly.'* As you probably figured out, we must set the `BDate` property in the code, instead of the `Age` property. The two errors are the same, but they refer to two different statements that attempt to set the `Age` property.

There are two types of errors that can occur while setting the `BDate` property: an invalid date or a date that yields an unreasonable age. First, the code of the `BDate` property makes sure that the value passed by the calling application is a valid date. If not, it throws an exception. If the *`value`* variable is a valid date, the code calls the `DateDiff()` function, which returns the difference between two dates in a specified interval — in our case, years. The expression `DateInterval.Year` is the name of a constant, which tells the `DateDiff()` function to calculate the difference between the two dates in years. You don't have to memorize the constant names — you simply select them from a list as you type.

So, the code checks the number of years between the date of birth and the current date. If it's negative (which means that the person hasn't been born yet) or more than 100 years (we'll assume that people over 100 will be treated as being 100 years old), it rejects the value. Otherwise, it sets the value of the *`m_BDate`* local variable and calculates the value of the *`m_Age`* local variable.

Calculating Property Values on the Fly

There's still a serious flaw in the implementation of the `Age` property. Can you see it? The person's age is up-to-date the moment the birth date is entered, but what if we read it back from a file or database three years later? It will still return the original value, which is no longer the correct age. The `Age` property's value shouldn't be stored anywhere; it should be calculated from the person's birth date as needed. If we avoid storing the age to a local variable and calculate it on the fly, users will always see the correct age. Revise the `Age` property's code to match Listing 10.8, which calculates the difference between the date of birth and the current date, and returns the correct person's age every time it's called.

Listing 10.8: A Calculated Property

```
ReadOnly Property Age() As Integer
   Get
      Age = Convert.ToInt32(DateDiff(DateInterval.Year, m_BDate , Now()))
   End Get
End Property
```

Notice also that you no longer need the *`m_Age`* local variable because the age is calculated on the fly when requested, so remove its declaration from the class. As you can see, you don't always have to store property values to local variables. A property that returns the number of files in a directory, for example, also doesn't store its value in a local variable. It retrieves the requested information on the fly and furnishes it to the calling application. By the way, the calculations might still return a negative value if the user has changed the system's date, but this is a rather far-fetched scenario.

You can implement write-only properties with the `WriteOnly` keyword and a `Set` section only, but write-only properties are rarely used (in my experience, only for storing passwords).

Our Minimal class is no longer so minimal. It exposes some functionality, and you can easily add more. Add properties for name, profession, and income, and add methods to calculate insurance rates based on a person's age and anything you can think of. Experiment with a few custom members, add the necessary validation code in your Property procedures, and you'll soon find out that building and reusing custom classes is a simple and straightforward process. Of course, there's a lot more to learn about classes, but you already have a good understanding of the way classes combine code with data.

Customizing Default Members

As you recall, when you created the Minimal class for the first time, before adding any code, the class already exposed a few members — the default members, such as the `ToString` method (which returns the name of the class) and the `Equals` method (which compares two objects for reference equality). You can (and should) provide your custom implementation for these members; this is what we're going to do in this section.

Customizing the *ToString* Method

The custom `ToString` method is implemented as a Public function, and it must override the default implementation. The implementation of a custom `ToString` method is shown next:

```
Public Overrides Function ToString() As String
   Return "The infamous Minimal class"
End Function
```

As soon as you enter the keyword Overrides, the editor will suggest the names of the three members you can override: ToString, Equals, and GetHashCode. Select the ToString method, and the editor will insert a default implementation for you. The default implementation returns the string MyBase.ToString. This is the default implementation. (You'll see later in this chapter what the MyBase keyword is; it basically references the default class implementation.) Just replace the statement inserted by the editor with the one shown in the preceding statement. It's that simple.

The Overrides keyword tells the compiler that this implementation overwrites the default implementation of the class. The original method's code isn't exposed, and you can't revise it. The Overrides keyword tells the compiler to ''hide'' the original implementation and use your custom ToString method instead. After you override a method in a class, the application using the class can no longer access the original method. Ours is a simple method, but you can return any string you can build in the function. For example, you can incorporate the value of the BDate property in the string:

```
Return("MINIMAL: " & m_BDate.ToShortDateString)
```

The value of the local variable *m_BDate* is the value of the BDate property of the current instance of the class. Change the BDate property, and the ToString method will return a different string.

When called through different variables, the ToString method will report different values. Let's say that you created and initialized two instances of the Minimal class by using the following statements:

```
Dim obj1 As New Minimal()
Obj1.Bdate = #1/1/1963#
Dim obj2 As New Minimal()
Obj2.Bdate = #12/31/1950#
Debug.WriteLine(obj1.ToString)
Debug.WriteLine(obj2.ToString)
```

The last two statements will print the following lines in the Output window:

```
MINIMAL: 1963-01-01
MINIMAL: 1950-12-31
```

Customizing the *Equals* Method

The Equals method exposed by most of the built-in objects can compare values, not references. Two Rectangle objects, for example, are equal if their dimensions and origins are the same. The following two rectangles are equal:

```
Dim R1 As New Rectangle(0, 0, 30, 60)
Dim R2 As New Rectangle
R2.X = 0
R2.Y = 0
R2.Width = 30
R2.Height = 60
If R1.Equals(R2) Then
   MsgBox("The two rectangles are equal")
End If
```

If you execute these statements, a message box confirming the equality of the two objects will pop up. The two variables point to different objects (that is, different instances of the same class), but the two objects are equal, because they have the same origin and same dimensions. The Rectangle class provides its own `Equals` method, which knows how to compare two Rectangle objects. If your class doesn't provide a custom `Equals` method, all the compiler can do is compare the objects referenced by the two variables of the specific type. In the case of our Minimal class, the `Equals` method returns True if the two variables point to the same object (which is the same instance of the class). If the two variables point to two different objects, the default `Equals` method will return False, even if the two objects are equal.

You're probably wondering what makes two objects equal. Is it all their properties or perhaps some of them? Two objects are equal if the `Equals` method says so. You should compare the objects in a way that makes sense, but you're in no way limited as to how you do this. In a very specific application, you might decide that two rectangles are equal because they have the same area, or perimeter, regardless of their dimensions and origin, and override the Rectangle object's `Equals` method. In the Minimal class, for example, you might decide to compare the birth dates and return True if they're equal. Listing 10.9 is the implementation of a possible custom `Equals` method for the Minimal class.

Listing 10.9: A Custom *Equals* Method

```
Public Overrides Function Equals(ByVal obj As Object) As Boolean
   Dim O As Minimal = CType(obj, Minimal)
   If O.BDate = m_BDate Then
      Equals = True
   Else
      Equals = False
   End If
End Function
```

Notice that the `Equals` method is prefixed with the `Overrides` keyword, which tells the compiler to use our custom `Equals` method in the place of the original one. To test the new `Equals` method, place a new button on the form and insert the statements of Listing 10.10 in its `Click` event handler.

Listing 10.10: Testing the Custom *Equals* Method

```
Dim O1 As New Minimal
Dim O2 As New Minimal
O1.BDate = #3/1/1960#
O2.BDate = #3/1/1960#
O1.strProperty = "object1"
O2.strProperty = "OBJECT2"
If O1.Equals(O2) Then
   MsgBox("They're equal")
End If
```

If you run the application, you'll see the message confirming that the two objects are equal, despite the fact that their `strProperty` properties were set to different values. The `BDate` property is the same, and this is the only setting that the `Equals` method examines. So, it's up to you to decide which properties fully and uniquely identify an object and to use these properties to determine when two objects are equal. In a class that represents persons, you'd probably use Social Security numbers, or a combination of names and birth dates. In a class that represents cars, you'd use the maker, model, and year, and so on.

Know What You Are Comparing

The `Equals` method shown in Listing 10.10 assumes that the object you're trying to compare to the current instance of the class is of the same type. Because you can't rely on developers to catch all their mistakes, you should know what you're comparing before you attempt to perform the comparison. A more-robust implementation of the `Equals` method is shown in Listing 10.11. This implementation tries to convert the argument of the `Equals` method to an object of the Minimal type and then compares it to the current instance of the Minimal class. If the conversion fails, an `InvalidCastException` is thrown and no comparison is performed.

Listing 10.11: A More-Robust *Equals* Method

```
Public Overrides Function Equals(ByVal obj As Object) As Boolean
   Dim O As New Minimal()
   Try
      O = DirectCast(obj, Minimal)
   Catch typeExc As InvalidCastException
      Throw typeExc
      Exit Function
   End Try
   If O.BDate = m_BDate Then
      Equals = True
   Else
      Equals = False
   End If
End Function
```

The *Is* Operator

The equals (=) operator can be used in comparing all built-in objects. The following statement is quite valid, as long as the *R1* and *R2* variables were declared of the Rectangle type:

```
If R1 = R2 Then
    MsgBox("The two rectangles are equal")
End If
```

This operator, however, can't be used with the Minimal custom class. Later in this chapter, you'll learn how to customize operators in your class. In the meantime, you can use only the `Is` operator, which compares for reference equality (whether the two variables reference the same object), and the `Equals` method. If the two variables *R1* and *R2* point to the same object, the following statement will return True:

```
If obj1 Is obj2 Then
    MsgBox("The two variables reference the same object")
End If
```

The `Is` operator tells you that the two variables point to a single object. There's no comparison here; the compiler simply figures out whether they point to same object in memory. It will return True if a statement such as the following has been executed before the comparison:

```
obj2 = obj1
```

If the `Is` operator returns True, there's only one object in memory and you can set its properties through either variable.

Custom Enumerations

Let's add a little more complexity to our class. Because we're storing birth dates to our custom objects, we can classify persons according to their age. Most BASIC developers will see an opportunity to use constants here. Instead of using constants to describe the various age groups, we'll use an enumeration with the following group names:

```
Public Enum AgeGroup
   Infant
   Child
   Teenager
   Adult
   Senior
   Overaged
End Enum
```

These statements must appear outside any procedure in the class, and we usually place them at the beginning of the file, right after the declaration of the class. `Public` is an access modifier (we want to be able to access this enumeration from within the application that uses the class). `Enum` is a keyword: It specifies the beginning of the declaration of an enumeration and it's followed by the enumeration's name. The enumeration itself is a list of integer values, each one mapped to a name. In our example, the name `Infant` corresponds to 0, the name `Child` corresponds to 1, and so on. The list of the enumeration's members ends with the `End Enum` keyword. You don't really care about the actual values of the names because the very reason for using enumerations is to replace numeric constants with more-meaningful names. You'll see shortly how enumerations are used both in the class and the calling application.

As you already know, the Framework uses enumerations extensively, and this is how you can add an enumeration to your custom class. You should provide an enumeration for any property with a relatively small number of predetermined settings. The property's type should be the name of the enumeration, and the editor will open a drop-down box with the property's settings as needed.

Now add to the class the `GetAgeGroup` method (Listing 10.12), which returns the name of the age group to which the person represented by an instance of the Minimal class belongs. The name of the group is a member of the `AgeGroup` enumeration.

LISTING 10.12: Using an Enumeration

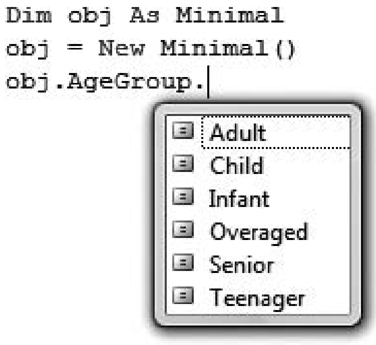
```
Public Function GetAgeGroup() As AgeGroup
   Select Case m_Age
      Case Is < 3 : Return (AgeGroup.Infant)
      Case Is < 10 : Return (AgeGroup.Child)
      Case Is < 21 : Return (AgeGroup.Teenager)
      Case Is < 65 : Return (AgeGroup.Adult)
      Case Is < 100 : Return (AgeGroup.Senior)
      Case Else : Return (AgeGroup.Overaged)
   End Select
End Function
```

The `GetAgeGroup` method returns a value of the `AgeGroup` type. Because the `AgeGroup` enumeration was declared as Public, it's exposed to any application that uses the Minimal class. Let's see how we can use the same enumeration in our application. Switch to the form's code window, add a new button, and enter the statements from Listing 10.13 in its event handler.

LISTING 10.13: Using the Enumeration Exposed by the Class

```
Protected Sub Button1_Click(...) _
                 Handles Button1.Click
   Dim obj As Minimal
        obj = New Minimal()
        Try
            obj.BDate = InputBox("Please Enter your birthdate")
        Catch ex As ArgumentException
            MsgBox(ex.Message)
            Exit Sub
        End Try
        Debug.WriteLine(obj.Age)
        Dim discount As Single
        If obj.GetAgeGroup = Minimal.AgeGroup.Infant Or _
           obj.GetAgeGroup = Minimal.AgeGroup.Child Then discount = 0.4
        If obj.GetAgeGroup = Minimal.AgeGroup.Senior Then discount = 0.5
        If obj.GetAgeGroup = Minimal.AgeGroup.Teenager Then discount = 0.25
        MsgBox("You age is " & obj.Age.ToString & _
           " and belong to the " & _
           obj.GetAgeGroup.ToString & _
           " group" & vbCrLf & "Your discount is " & _
           Format(discount, "Percent"))
End Sub
```

This routine calculates discounts based on the person's age. Notice that we don't use numeric constants in our code, just descriptive names. Moreover, the possible values of the enumeration are displayed in a drop-down list by the IntelliSense feature of the IDE as needed (Figure 10.4), and you don't have to memorize them or look them up as you would with constants. I've used an implementation with multiple `If` statements in this example, but you can perform the same comparisons by using a `Select Case` statement.



```
Dim obj As Minimal
obj = New Minimal()
obj.AgeGroup.
```

FIGURE 10.4
The members of an enumeration are displayed automatically in the IDE as you type.

You could also call the `GetAgeGroup` method once, store its result to a variable, and then use this variable in the comparisons. This approach is slightly more efficient, because you don't have to call the member of the class repeatedly. The variable, as you can guess, should be of the `AgeGroup` type. Here's an alternate code of the statements of Listing 10.13 using a temporary variable, the *`grp`* variable, and a `Select Case` statement:

```
Dim grp As AgeGroup = obj.GetAgeGroup
Select Case grp
    Case Minimal.AgeGroup.Infant, Minimal.AgeGroup.Child ...
    Case Minimal.AgeGroup.Teenager ...
    Case Minimal.AgeGroup.Senior ...
    Case Else
End Select
```

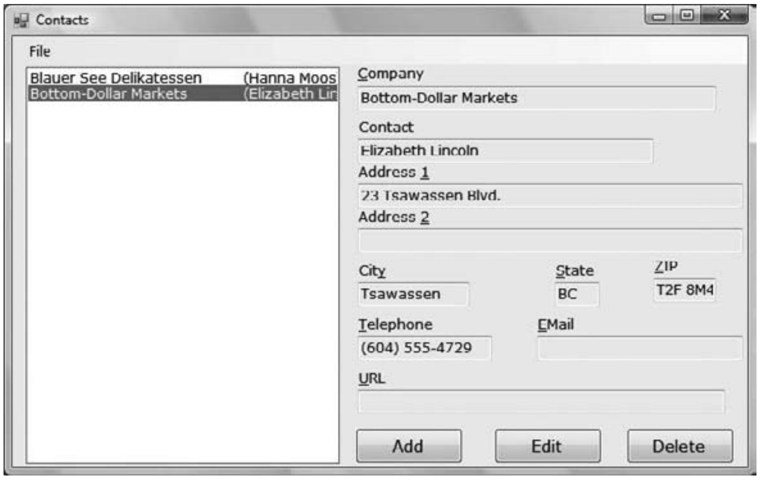
You've seen the basics of working with custom classes in a VB application. Let's switch to a practical example that demonstrates not only the use of a real-world class, but also how classes can simplify the development of a project.

VB 2008 AT WORK: THE CONTACTS PROJECT

In Chapter 7, ''Working with Forms,'' I discussed briefly the Contacts application. This application uses a custom Structure to store the contacts and provides four navigational buttons to allow users to move to the first, last, previous, and next contact. Now that you have learned how to program the ListBox control and how to use custom classes in your code, we'll revise the Contacts application. First, we'll implement a class to represent each contact. The fields of each contact (company and contact names, addresses, and so on) will be implemented as properties and they will be displayed in the TextBox controls on the form.

We'll also improve the user interface of the application. Instead of the rather simplistic navigational buttons, we'll place all the company names in a sorted ListBox control. The user can easily locate the desired company and select it from the list to view the fields of the selected company. The editing buttons at the bottom of the form work as usual, but we no longer need the navigational buttons. Figure 10.5 shows the revised Contacts application.

FIGURE 10.5
The interface of the Contacts application is based on the ListBox control.



Blauer See Delikatessen (Hanna Moos
Bottom-Dollar Markets (Elizabeth Lin

Company	Bottom-Dollar Markets
Contact	Elizabeth Lincoln
Address 1	23 Tsawassen Blvd.
Address 2	
City	Tsawassen
State	BC
ZIP	T2F 8M4
Telephone	(604) 555-4729
EMail	
URL	

Make a copy of the Contacts folder from Chapter 7 into a new folder. First, delete the declaration of the Contact structure and add a class to the project. Name the new class **Contact** and enter the code from Listing 10.14 into it. The names of the private members of the class are the same as the actual property names, and they begin with an underscore. (This is a good convention that lets you easily distinguish whether a variable is private, and the property value it stores.) The implementation of the properties is trivial, so I'm not showing the code for all of them.

LISTING 10.14: The Contact Class

```
<Serializable()> Public Class Contact
    Private _companyName As String
    Private _contactName As String
    Private _address1 As String
    Private _address2 As String
    Private _city As String
    Private _state As String
    Private _zip As String
    Private _tel As String
    Private _email As String
    Private _URL As String

    Property CompanyName() As String
        Get
            CompanyName = _companyName
        End Get
        Set(ByVal value As String)
            If value Is Nothing Or value = "" Then
                Throw New Exception("Company Name field can't be empty")
                Exit Property
            End If
            _companyName = value
        End Set
    End Property

    Property ContactName() As String
        Get
            ContactName = _contactName
        End Get
        Set(ByVal value As String)
            _contactName = value
        End Set
    End Property

    Property Address1() As String
        Get
            Address1 = _address1
        End Get
        Set(ByVal value As String)
            _address1 = value
        End Set
    End Property
    Property Address2() As String
        ...
    End Property

    Property City() As String
        ...
    End Property

    Property State() As String
        ...
    End Property

    Property ZIP() As String
        ...
    End Property

    Property tel() As String
        ...
       End Property

    Property EMail() As String
    Get
            EMail = _email
        End Get
        Set(ByVal value As String)
            If value.Contains("@") Or value.Trim.Length = 0 Then
                _email = Value
            Else
                Throw New Exception("Invalid e-mail address!")
            End If
        End Set
    End Property

    Property URL() As String
        Get
            URL = _URL
        End Get
        Set(ByVal value As String)
            _URL = value
        End Set
    End Property

    Overrides Function ToString() As String
        If _contactName = "" Then
            Return _companyName
        Else
            Return _companyName & vbTab & "(" & _contactName & ")"
        End If
    End Function

    Public Sub New()
        MyBase.New()
    End Sub

    Public Sub New(ByVal CompanyName As String, _
                   ByVal LastName As String, ByVal FirstName As String)
        MyBase.New()
        Me.ContactName = LastName & ", " & FirstName
        Me.CompanyName = CompanyName
    End Sub

    Public Sub New(ByVal CompanyName As String)
        MyBase.New()
        Me.CompanyName = CompanyName
    End Sub
End Class
```

The first thing you'll notice is that the class's definition is prefixed by the <`Serializable()`> keyword. The topic of serialization is discussed in Chapter 16, ''XML and Object Serialization,'' but for now all you need to know is that the .NET Framework can convert objects to a text or binary format and then store them in files. Surprisingly, this process is quite simple; as you will see, we'll be able to dump an entire collection of Contact objects to a file with a single statement. The <`Serializable()`> keyword is an attribute of the class, and (as you will see later in this book) there are more attributes you can use with your classes — or even with your methods. The most prominent method attribute is the <`WebMethod`> attribute, which turns a regular function into a web method.

The various fields of the Contact structure are now properties of the Contact class. The implementation of the properties is trivial except for the `CompanyName` and `EMail` properties, which contain some validation code. The Contact class requires that the `CompanyName` property have a value; if it doesn't, the class throws an exception. Likewise, the `EMail` property must contain the symbol `@`. Finally, the class provides its own `ToString` method, which returns the name of the company followed by the contact name in parentheses.

The ListBox control, in which we'll store all contacts, displays the value returned by the object's `ToString` method, which is why you have to provide your own implementation of the method to describe each contact. The company name should be adequate, but if there are two companies by the same name, you can use another field to differentiate them. I used the contact name, but you can use any of the other properties (the URL would be a good choice).

The ListBox displays a string, but it stores the object itself. In essence, it's used not only as a navigational tool, but also as a storage mechanism for our contacts. Now, we must change the code of the main form a little. Start by removing the navigational buttons; we no longer need them. Their function will be replaced by a few lines of code in the ListBox control's `SelectedIndexChanged` event. Every time the user selects another item on the list, the statements shown in Listing 10.15 display the contact's properties in the various TextBox controls on the form.

LISTING 10.15: Displaying the Fields of the Selected Contact Object

```
Private Sub ListBox1_SelectedIndexChanged(...) _
              Handles ListBox1.SelectedIndexChanged
   currentContact = ListBox1.SelectedIndex
   ShowContact()
End Sub
```

The `ShowContact()` subroutine reads the object stored at the location specified by the *`currentContact`* variable and displays its properties in the various TextBox controls on the form. The TextBox controls are normally read-only, except when editing a contact. This action is signaled by clicking the Edit or the Add button on the form.

When a new contact is added, the code reads its fields from the controls on the form, creates a new Contact object, and adds it to the ListBox control. When a contact is edited, a new Contact object replaces the currently selected object on the control. The code is similar to the code of the Contacts application. I should mention that the ListBox control is locked while a contact is being added or edited, because it doesn't make sense to select another contact at that time.

Adding, Editing, and Deleting Contacts

To delete a contact (Listing 10.16), we simply remove the currently selected object from the ListBox control. In addition, we must select the next contact on the list (or the first contact if the deleted one was last in the list).

LISTING 10.16: Deleting an Object from the ListBox

```
Private Sub bttnDelete_Click(...) Handles bttnDelete.Click
   If currentContact > -1 Then
      ListBox1.Items.RemoveAt(currentContact)
      currentContact = ListBox1.Items.Count - 1
      If currentContact = -1 Then
         ClearFields()
```

```
            MsgBox("There are no more contacts")
         Else
            ShowContact()
         End If
      Else
         MsgBox("No current contacts to delete")
      End If
   End Sub
```

When you add a new contact, the following code is executed in the Add button's `Click` event handler:

```
Private Sub bttnAdd_Click(...) Handles bttnAdd.Click
   adding = True
   ClearFields()
   HideButtons()
   ListBox1.Enabled = False
End Sub
```

The controls are cleared in anticipation of the new contact's fields, and the `adding` variable is set to True. The OK button is clicked to end either the addition of a new record or an edit operation. The code behind the OK button is shown in Listing 10.17.

LISTING 10.17: Committing a New or Edited Record

```
Private Sub bttnOK_Click(...) Handles bttnOK.Click
    If SaveContact() Then
        ListBox1.Enabled = True
        ShowButtons()
    End If
End Sub
```

As you can see, the same subroutine handles both the insertion of a new record and the editing of an existing one. All the work is done by the `SaveContact()` subroutine, which is shown in Listing 10.18.

LISTING 10.18: The *SaveContact()* Subroutine

```
Private Function SaveContact() As Boolean
    Dim contact As New Contact
    Try
        contact.CompanyName = txtCompany.Text
        contact.ContactName = txtContact.Text
        contact.Address1 = txtAddress1.Text
        contact.Address2 = txtAddress2.Text
        contact.City = txtCity.Text
        contact.State = txtState.Text
        contact.ZIP = txtZIP.Text
```

```
            contact.tel = txtTel.Text
            contact.EMail = txtEMail.Text
            contact.URL = txtURL.Text
        Catch ex As Exception
            MsgBox(ex.Message)
            Return False
        End Try
        If adding Then
            ListBox1.Items.Add(contact)
        Else
            ListBox1.Items(currentContact) = contact
        End If
        Return True
    End Function
```

The `SaveContact()` function uses the `adding` variable to distinguish between an add and an edit operation, and either adds the new record to the ListBox control or replaces the current item in the ListBox with the values on the various controls. Because the ListBox is sorted, new contacts are automatically inserted in the correct order. If an error occurs during the operation, the `SaveContact()` function returns False to alert the calling code that the operation failed (most likely because one of the assignment operations caused a validation error in the class's code).

The last operation of the application is the serialization and deserialization of the items in the ListBox control. *Serialization* is the process of converting an object to a stream of bytes for storing to a disk file, and *deserialization* is the opposite process. To serialize objects, we first store them into an ArrayList object, which is a dynamic array that stores objects and can be serialized as a whole. Likewise, the disk file is deserialized into an ArrayList to reload the persisted data back to the application; then each element of the ArrayList is moved to the `Items` collection of the ListBox control. ArrayLists and other Framework collections are discussed in Chapter 14, ''Storing Data in Collections,'' and object serialization is discussed in Chapter 16. You can use these features to test the application and examine the corresponding code after you read about ArrayLists and serialization. I'll discuss the code of the Load and Save operations of the Contacts sample project in Chapter 16.

Real World Scenario

Making the Most of the ListBox Control

This section's sample application demonstrates an interesting technique for handling a set of data at the client. We usually need an efficient mechanism to store data at the client, where all the processing takes place — even if the data comes from a database. In this example, we used the ListBox control, because each item of the control can be an arbitrary object. Because the control displays the string returned by the object's `ToString` method, we're able to customize the display by providing our own implementation of the `ToString` method. As a result, we're able to use the ListBox control both as a data-storage mechanism and as a navigational tool. As long as the strings displayed on the control are meaningful descriptions of the corresponding objects and the control's items are sorted, the ListBox control can be used as an effective navigational tool. If you have too many items to display on the control, you should also provide a search tool to help users quickly locate an item in the list, without having to scroll up and down a long list of items. Review the ListBoxFind project of Chapter 6 for information on searching the contents of the ListBox control.

When data are being edited, you have to cope with another possible problem. The user may edit the data for hours and forget to save the edits every now and then. If the computer (or, even worse, the application) crashes, a lot of work will be wasted. Sure the application provides a Save command, but you should always try to protect users from their mistakes. It would be nice if you could save the data to a temporary file every time the user edits or adds an item to the list. This way, if the computer crashes, users won't lose their edits. When the application starts, it should automatically detect the presence of the temporary file and reload it. Every time the user saves the data by using the application's Save command, or terminates the application, the temporary file should be removed.

Object Constructors

Let's switch to a few interesting topics in programming with objects. Objects are instances of classes, and classes are instantiated with the `New` keyword. The `New` keyword can be used with a number of arguments, which are the initial values of some of the object's basic properties. To construct a rectangle, for example, you can use these two statements:

```
Dim shape1 As Rectangle = New Rectangle()
shape1.Width = 100
shape1.Height = 30
```

or the following one:

```
Dim shape1 As Rectangle = New Rectangle(100, 30)
```

The objects in the Minimal class can't be initialized to specific values of their properties and they expose the simple form of the `New` constructor — the so-called parameterless constructor. Every class has a *parameterless* constructor, even if you don't specify it. You can implement *parameterized* constructors, which allow you to pass arguments to an object as you declare it. These arguments are usually the values of the object's basic properties. Parameterized constructors don't pass arguments for all the properties of the object; they expect only enough parameter values to make the object usable.

Parameterized constructors are implemented via Public subroutines that have the name `New`. You can have as many overloaded forms of the `New()` subroutine as needed. Most of the built-in classes provide a parameterless constructor, but the purists of OOP will argue against parameterless constructors. Their argument is that you shouldn't allow users of your class to create invalid instances of it. A class for describing customers, for example, should expose at least a `Name` property. A class for describing books should expose a `Title` and an `ISBN` property. If the corresponding constructor requires that these properties be specified before creating an instance of the class, you'll never create objects with invalid data.

Let's add a parameterized constructor to our Contact class. Each contact should have at least a name; here's a parameterized constructor for the Contact class:

```
Public Sub New(ByVal CompanyName As String)
    MyBase.New()
    Me.CompanyName = CompanyName
End Sub
```

The code is trivial, with the exception of the statement that calls the `MyBase.New()` subroutine. `MyBase` is an object that lets you access the members of the base class (a topic that's discussed in detail later in this chapter). The reason you must call the New method of the base class is that the base class might have its own constructor, which can't be called directly. You must always insert this statement in your constructors to make sure that any initialization tasks that must be performed by the base class will not be skipped.

The Contact class's constructor accepts a single argument: the company name (this property can't be a blank string). Another useful constructor for the same class accepts two additional arguments, the contact's first and last names, as follows:

```
Public Sub New(ByVal CompanyName As String, _
               ByVal LastName As String, ByVal FirstName As String)
    MyBase.New()
    Me.ContactName = LastName & ", " & FirstName
    Me.CompanyName = CompanyName
End Sub
```

With the two parameterized constructors in place, you can create new instances of the Contact class by using statements such as the following:

```
Dim contact1 As New Contact("Around the Horn")
```

Or the following:

```
Dim contact1 As New Contact("Around the Horn", "Hardy", "Thomas")
```

Notice the lack of the Overloads (or Overrides) keyword. Constructors can have multiple forms and don't require the use of Overloads — just supply as many implementations of the New() subroutine as you need.

One last, but very convenient technique to initialize objects was introduced with Visual Basic 2008. This technique allows you to supply values for as many properties of the new object as you wish, using the With keyword. The following statements create two new instances of the Person class, and they initialize each one differently:

```
Dim P1 As New Person With _
       {.LastName = "Doe", .FirstName = "Joe"})
Dim P2 As New Person With _
       {.LastName = "Doe", .Email = "Doe@xxx.com"})
```

This syntax allows you to quickly initialize new objects, regardless of their constructors; in effect, you can create your own constructor for any class. This technique will be handy when combining object initialization with other statements, such as in the following example, which adds a new object to a list:

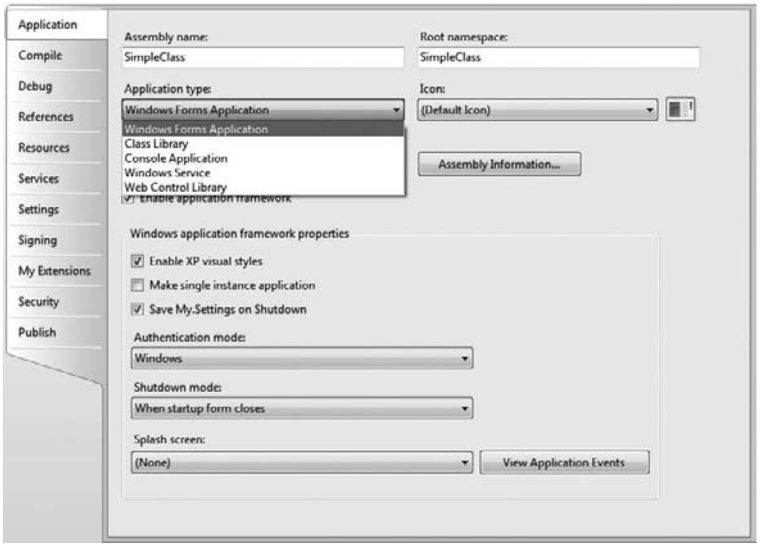
```
Persons.Add(New Person With {.LastName = "Doe", .FirstName = "Joe"})
Persons.Add(New Person With {.LastName = "Doe"})
```

Using the SimpleClass in Other Projects

The projects we built in this section are Windows applications that contain a Class module. The class is contained within the project, and it's used by the project's main form. What if you want to use this class in another project?

First, you must change the type of the project. A Windows project can't be used as a component in another project. Right-click the SimpleClass project and choose Properties. In the project's Property Pages dialog box, switch to the Application tab, locate the Application Type drop-down list, and change the project's type from Windows Forms Application to Class Library, as shown in Figure 10.6. Then close the dialog box. When you return to the project, right-click the TestForm and select Exclude From Project. A class doesn't have a visible interface, and there's no reason to include the test form in your project.

FIGURE 10.6
Setting a project's properties through the Property Pages dialog box



Assembly name: SimpleClass
Root namespace: SimpleClass
Application type: Windows Forms Application
Icon: (Default Icon)
[✓] 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: (None)

From the main menu, choose Build ➢ Build SimpleClass. This command will compile the SimpleClass project and create a DLL file (the file that contains the class's code and the file you must use in any project that needs the functionality of the SimpleClass class). The DLL file will be created in the `\bin\Release` folder under the project's folder.

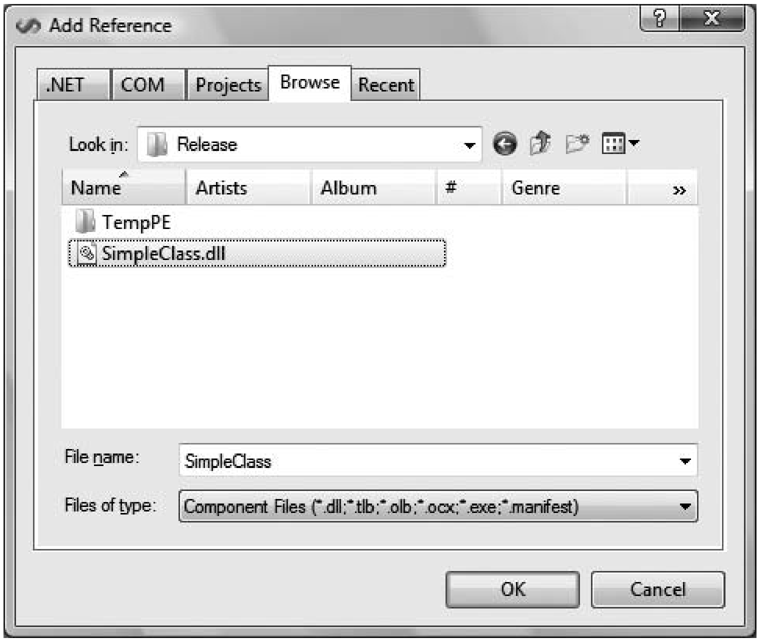
Let's use the `SimpleClass.dll` file in another project. Start a new Windows application, open the Project menu, and add a reference to the SimpleClass. Choose Project ➢ Add Reference and switch to the Projects tab in the dialog box that appears. Click the Browse button and locate the `SimpleClass.dll` file (see Figure 10.7). Select the name of the file and click OK to close the dialog box.

The SimpleClass component will be added to the project. You can now declare a variable of the `SimpleClass.Minimal` type and call its properties and methods:

```
Dim obj As New SimpleClass.Minimal
obj.BDate = #10/15/1992#
obj.property2 = 5544
MsgBox(obj.Negate())
```

If you want to keep testing the SimpleClass project, add the TestForm to the original project (right-click the project's name, choose Add ➢ Add Existing Item, and select the TestForm in the project's folder). Change the project's type back to Windows Forms Application and then change its configuration from Release to Debug.

FIGURE 10.7
Adding a reference to an existing class to a new project



Look in: Release

TempPE
SimpleClass.dll

File name: SimpleClass
Files of type: Component Files (*.dll;*.tlb;*.olb;*.ocx;*.exe;*.manifest)

Firing Events

In addition to methods and properties, classes can also fire events. It's possible to raise events from within your classes, although not quite as common. Controls have many events because they expose a visible interface and the user interacts through this interface (clicks, drags and drops, and so on). But classes can also raise events. Class events can come from three different sources:

Progress events A class might raise an event to indicate the progress of a lengthy process or indicate that an internal variable or property has changed value. The `PercentDone` event is a typical example. A process that takes a while to complete reports its progress to the calling application with this event, which is fired periodically. These events, which are called *progress events*, are the most common type of class events.

Time events *Time events* are based on a timer. They're not very common, but you can implement alarms, job schedulers, and similar applications. You can set an alarm for a specific time or an alarm that will go off after a specified interval.

External events *External events*, such as the completion of an asynchronous operation, can also fire events. A class might initiate a file download and notify the application when the file arrives.

To fire an event from within a class, you must do the following:

1. First you must declare the event and its signature in your class. The declaration must appear in the form, not in any procedure. A simple event, with no arguments, should be declared as follows (`ShiftEnd` is the name of the event — an event that signals the end of a shift every eight hours):

```
Public Event ShiftEnd()
```

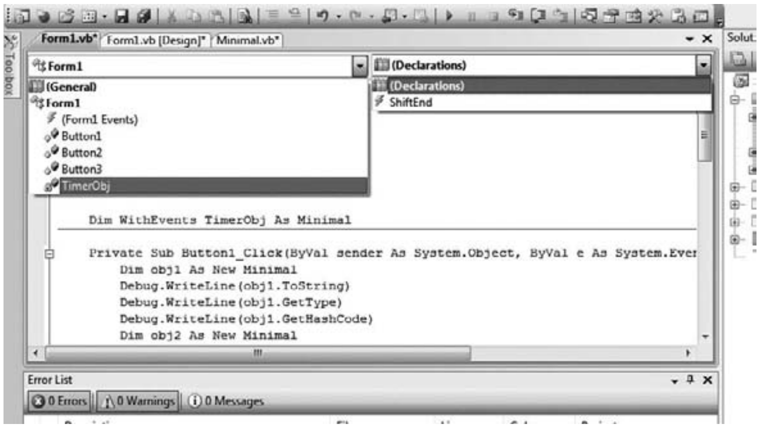
2. Fire the event from within your class's code with the RaiseEvent method:

```
RaiseEvent  ShiftEnd()
```

3. That's all as far as the class is concerned.
4. The application that uses the custom class must declare it with the WithEvents keyword. Otherwise, it will still be able to use the class's methods and properties, but the events raised by the class will go unnoticed. The following statement creates an instance of the class and listens for any event:

```
Dim WithEvents obj As New Minimal
```

5. Finally, the calling application must provide a handler for the specific event. Because the class was declared with the WithEvents keyword, its name will appear in the list of objects in the editor window and its ShiftEnd event will appear in the list of events (Figure 10.8). Insert the code you want to handle this event in the procedure obj.ShiftEnd.



```
    Dim WithEvents TimerObj As Minimal

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.Ever
        Dim obj1 As New Minimal
        Debug.WriteLine(obj1.ToString)
        Debug.WriteLine(obj1.GetType)
        Debug.WriteLine(obj1.GetHashCode)
        Dim obj2 As New Minimal
```

FIGURE 10.8
Programming a custom class's event

Events usually pass information to the calling application. In VB, all events pass two arguments to the application: a reference to the object that fired the event, and another argument (which is an object and contains information specific to the event).

The arguments of an event are declared just like the arguments of a procedure. The following statement declares an event that's fired when the class completes the download of a file. The event passes three parameter values to the application that intercepts it:

```
Public Event CompletedDownload(ByVal fileURL As String, _
                ByVal fileName As String, ByVal fileLength As Long)
```

The parameters passed to the application through this event are the URL from which the file was downloaded, the path of a file where the downloaded information was stored, and the length of the file. To raise this event from within a class's code, call the RaiseEvent statement as before, passing three values of the appropriate type, as shown next:

```
RaiseEvent CompletedDownload("http://www.server.com/file.txt", _
                    "d:\temp\A90100.txt", 144329)
```

When coding the event's handler, you can access these arguments and use them as you wish. Alternatively, you could create a new object, the DownloadedFileArgument type, and expose these arguments as properties:

```
Public Class DownloadedFileArgument
    Public FileName as String
    Public TempLocation As String
    Public FileSize As Long
End Class
```

Then you can declare the event's signature by using the DownLoadedFileArgument type in the argument list:

```
Public Event CompletedDownload(ByVal sender As Object, _
             ByVal e As DownloadedFileArgument)
```

To fire the `CompletedDownload` event from within your class's code, create an instance of the DownLoadedFileArgument class, set its properties and then call the `RaiseEvent` method, as shown here:

```
Dim DArgument As New DownloadedFileArgument
DArgument.FileName = "http://www.server.com/file.txt"
DArgument.TempLocation = "d:\temp\A90100.txt"
DArgument.FileSize = 144329
RaiseEvent Fired(Me, DArgument)
```

To intercept this event in your test application, declare an object of the appropriate type with the `WithEvents` keyword and write an event handler for the `CompletedDownload` event:

```
Public WithEvents obj As New EventFiringClass
Private Sub obj_Fired(ByVal sender As Object, _
          ByVal e As Firing.DownloadedFileArgument) _
          Handles obj.CompletedDownload
    MsgBox("Event fired" & vbCrLf & _
           e.FileName & vbCrLf & _
           e.TempLocation & vbCrLf & _
           e.FileSize.ToString)
End Sub
```

That's all it takes to fire an event from within your custom class. In Chapter 12, ''Building Custom Windows Controls,'' you will find several examples of custom events.

Instance and Shared Methods

As you have seen in earlier chapters, some classes allow you to call some of their members without first creating an instance of the class. The DateTime class, for example, exposes the `IsLeapYear` method, which accepts as an argument a numeric value and returns a True/False value that indicates whether the year is a leap year. You can call this method through the DateTime (or Date) class without having to create a variable of the DateTime type, as shown in the following statement:

```
If DateTime.IsLeapYear(1999) Then
   { process a leap year}
End If
```

A typical example of classes that can be used without explicit instances is the Math class. To calculate the logarithm of a number, you can use an expression such as this one:

```
Math.Log(3.333)
```

The properties and methods that don't require you to create an instance of the class before you call them are called *shared methods*. Methods that must be applied to an instance of the class are called *instance methods*. By default, all methods are instance methods. To create a shared method, you must prefix the corresponding function declaration with the `Shared` keyword, just like a shared property.

Why do we need shared methods, and when should we create them? If a method doesn't apply to a specific instance of a class, make it shared. In other words, if a method doesn't act on the properties of the current instance of the class, it should be shared. Let's consider the DateTime class. The `DaysInMonth` method returns the number of days in the month (of a specific year) that is passed to the method as an argument. You don't really need to create an instance of a Date object to find out the number of days in a specific month of a specific year, so the `DaysInMonth` method is a shared method and can be called as follows:

```
DateTime.DaysInMonth(2004, 2)
```

Think of the `DaysInMonth` method this way: Do I need to create a new date to find out if a specific month has 30 or 31 days? If the answer is no, then the method is a candidate for a shared implementation.

The `AddDays` method, on the other hand, is an instance method. We have a date to which we want to add a number of days and construct a new date. In this case, it makes sense to apply the method to an instance of the class — the instance that represents the date to which we add the number of days.

If you spend a moment to reflect on shared and instance members, you'll come to the conclusion that all members could have been implemented as shared members and accept the data they act upon as arguments. The idea behind classes, however, is to combine data with code. If you implement a class with shared members, you lose one of the major advantages of OOP. Building a class with shared members only is equivalent to a collection of functions, and the Math class of the Framework is just that.

The SharedMembers sample project is a simple class that demonstrates the differences between a shared and an instance method. Both methods do the same thing: They reverse the characters in a string. The `IReverseString` method is an instance method; it reverses the current instance of the class, which is a string. The `SReverseString` method is a shared method; it reverses its argument. Listing 10.19 shows the code that implements the SharedMembersClass component.

Listing 10.19: A Class with a Shared and an Instance Method

```
Public Class SharedMembersClass
   Private strProperty As String

   Sub New(ByVal str As String)
      strProperty = str
   End Sub

   Public Function IReverseString() As String
```

```
        Return (StrReverse(strProperty))
     End Function

     Public Shared Function SReverseString(ByVal str As String) As String
        Return (StrReverse(str))
     End Function
  End Class
```

The instance method acts on the current instance of the class. This means that the class must be initialized to a string, and this is why the `New` constructor requires a string argument. To test the class, add a form to the project, make it the Startup object, and add two buttons to it. The code behind the two buttons is shown next:

```
Private Sub Button1_Click(...) Handles Button1.Click
   Dim testString As String = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
   Dim obj As New SharedMembersClass(testString)
   Debug.WriteLine(obj.IReverseString)
End Sub

Private Sub Button2_Click(...) Handles Button2.Click
   Dim testString As String = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
   Debug.WriteLine(SharedMembersClass.SReverseString(testString))
End Sub
```

The code behind the first button creates a new instance of the SharedMembersClass and calls its `IReverseString` method. The second button calls the `SReverseString` method through the class's name and passes the string to be reversed as an argument to the method.

A class can also expose shared properties. There are situations in which you want all instances of a class to see the same property value. Let's say you want to keep track of the users currently accessing your class. You can declare a method that must be called to enable the class, and this method signals that another user has requested your class. This method could establish a connection to a database or open a file. We'll call it the `Connect` method. Every time an application calls the `Connect` method, you can increase an internal variable by one. Likewise, every time an application calls the `Disconnect` method, the same internal variable is decreased by one. This internal variable can't be private because it will be initialized to zero with each new instance of the class. You need a variable that is common to all instances of the class. Such a variable should be declared with the `Shared` keyword.

Let's add a shared variable to our Minimal class. We'll call it `LoggedUsers`, and it will be read-only. Its value is reported via the `Users` property, and only the `Connect` and `Disconnect` methods can change its value. Listing 10.20 is the code you must add to the Minimal class to implement a shared property.

Listing 10.20: Implementing a Shared Property

```
  Shared LoggedUsers As Integer
  ReadOnly Property Users() As Integer
     Get
        Users = LoggedUsers
```

```
      End Get
   End Property

   Public Function Connect() As Integer
      LoggedUsers = LoggedUsers + 1
      { your own code here}
   End Function

   Public Function Disconnect() As Integer
      If LoggedUsers > 1 Then
         LoggedUsers = LoggedUsers - 1
      End If
      { your own code here}
   End Function
```

To test the shared variable, add a new button to the form and enter the code in Listing 10.21 in its `Click` event handler. (The lines with the **bold** numbers are the values reported by the class; they're not part of the listing.)

LISTING 10.21: Testing the *LoggedUsers* Shared Property

```
Protected Sub Button5_Click(ByVal sender As Object, _
                 ByVal e As System.EventArgs)
   Dim obj1 As New SharedMemberClass
   obj1.Connect()
   Debug.WriteLine(obj1.Users)
1
   obj1.Connect()
   Debug.WriteLine(obj1.Users)
2
   Dim obj2 As New SharedMemberClass
   obj2.Connect()
   Debug.WriteLine(obj1.Users)
3
   Debug.WriteLine(obj2.Users)
3
   Obj2.Disconnect()
   Debug.WriteLine(obj2.Users)
2
End Sub
```

If you run the application, you'll see the values displayed under each `Debug.WriteLine` statement in the Output window. As you can see, both the `obj1` and `obj2` variables access the same value of the `Users` property. Shared variables are commonly used in classes that run on a server and service multiple applications. In effect, they're the class's global variables, which can be shared among all the instances of a class. You can use shared variables to keep track of the total number of rows accessed by all users of the class in a database, connection time, and other similar quantities.

A ''Real'' Class

This section covers a more-practical class that exposes three methods for manipulating strings. I have used these methods in many projects, and I'm sure many readers will have good use for them — at least one of them. The first two methods are the `ExtractPathName` and `ExtractFileName` methods, which extract the filename and pathname from a full filename. If the full name of a file is `C:\Documents\Recipes\Chinese\Won Ton.txt`, the `ExtractPathName` method will return the substring `C:\Documents\Recipes\Chinese\`, and the `ExtractFileName` method will return the substring `Won Ton.txt`.

You can use the `Split` method of the String class to extract all the parts of a delimited string. Extracting the pathname and filename of a complete filename is so common in programming that it's a good idea to implement the corresponding functions as methods in a custom class. You can also use the Path object, which exposes a similar functionality. (The Path object is discussed in Chapter 15, ''Accessing Folders and Files.'')

The third method, which is called `Num2String`, converts numeric values (amounts) to the equivalent strings. For example, it can convert the amount $12,544 to the string `Twelve Thousand, Five Hundred And Forty Four dollars`. No other class in the Framework provides this functionality, and any program that prints checks can use this class.

Parsing a Filename

Let's start with the two methods that parse a complete filename. These methods are implemented as Public functions, and they're quite simple. Start a new project, rename the form to **TestForm**, and add a Class to the project. Name the class and the project **StringTools**. Then enter the code of Listing 10.22 in the Class module.

Listing 10.22: The *ExtractFileName* and *ExtractPathName* Methods

```
Public Function ExtractFileName(ByVal PathFileName As String) As String
   Dim delimiterPosition As Integer
   delimiterPosition = PathFileName.LastIndexOf("\")
   If delimiterPosition > 0 Then
      Return PathFileName.Substring(delimiterPosition + 1)
   Else
      Return PathFileName
   End If
End Function

Public Function ExtractPathName(ByVal PathFileName As String) As String
   Dim delimiterPosition As Integer
   delimiterPosition = PathFileName.LastIndexOf("\")
   If delimiterPosition > 0 Then
      Return PathFileName.Substring(0, delimiterPosition)
   Else
      Return ""
   End If
End Function
```

These are two simple functions that parse the string passed as an argument. If the string contains no delimiter, it's assumed that the entire argument is just a filename.

Converting Numbers to Strings

The Num2String method is far more complicated, but if you can implement it as a regular function, it doesn't take any more effort to turn it into a method. The listing of Num2String is shown in Listing 10.23. First, it formats the billions in the value (if the value is that large); then it formats the millions, thousands, units, and finally the decimal part, which can contain no more than two digits.

LISTING 10.23: Converting Numbers to Strings

```
Public Function Num2String(ByVal number As Decimal) As String
   Dim biln As Decimal, miln As Decimal, _
       thou As Decimal, hund As Decimal
   Dim ten As Integer, units As Integer
   Dim strNumber As String
   If number > 999999999999.99 Then
      Return ("***")
      Exit Function
   End If
   biln = Math.Floor(number / 1000000000)
   If biln > 0 Then _
       strNumber = FormatNum(biln) & " Billion" & Pad()
   miln = Math.Floor((number - biln * 1000000000) / 1000000)
   If miln > 0 Then _
      strNumber = strNumber & FormatNum(miln) & " Million" & Pad()
   thou = Math.Floor((number - biln * 1000000000 - miln * 1000000) / 1000)
   If thou > 0 Then _
      strNumber = strNumber & FormatNum(thou) & " Thousand" & Pad()
   hund = Math.Floor(number - biln * 1000000000 - miln * 1000000 - thou * 1000)
   If hund > 0 Then strNumber = strNumber & FormatNum(hund)
   If Right(strNumber, 1) = "," Then _
      strNumber = Left(strNumber, Len(strNumber) - 1)
   If Left(strNumber, 1) = "," Then _
      strNumber = Right(strNumber, Len(strNumber) - 1)
   If number <> Math.Floor(number) Then
      strNumber = strNumber & _
                  FormatDecimal(CInt((number - Int(number)) * 100))
   Else
      strNumber = strNumber & " dollars"
   End If
   Return (Delimit(SetCase(strNumber)))
End Function
```

Each group of three digits (million, thousand, and so on) is formatted by the FormatNum() function. Then the appropriate string is appended (Million, Thousand, and so on). The FormatNum() function, which converts a numeric value less than 1,000 to the equivalent string, is shown in Listing 10.24.

Listing 10.24: The *FormatNum()* Function

```
Private Function FormatNum(ByVal num As Decimal) As String
   Dim digit100 As Decimal, digit10 As Decimal, digit1 As Decimal
   Dim strNum As String
   digit100 = Math.Floor(num / 100)
   If digit100 > 0 Then strNum = Format100(digit100)
   digit10 = Math.Floor((num - digit100 * 100))
   If digit10 > 0 Then
      If strNum <> "" Then
         strNum = strNum & " And " & Format10(digit10)
      Else
         strNum = Format10(digit10)
      End If
   End If
   Retutn (strNum)
End Function
```

The `FormatNum()` function formats a three-digit number as a string. To do so, it calls the `Format100()` function to format the hundreds, and the `Format10()` function formats the tens. The `Format10()` function calls the `Format1()` function to format the units. I will not show the code for these functions; you can find it in the StringTools project. You'd probably use similar functions to implement the `Num2String` method as a function. Instead, I will focus on a few peripheral issues, such as the enumerations used by the class as property values.

To make the `Num2String` method more flexible, the class exposes the `Case`, `Delimiter`, and `Padding` properties. The `Case` property determines the case of the characters in the string returned by the method. The `Delimiter` property specifies the special characters that should appear before and after the string. Finally, the `Padding` property specifies the character that will appear between groups of digits. The values each of these properties can take on are members of the appropriate enumeration:

PaddingEnum	**DelimiterEnum**	**CaseEnum**
paddingCommas	delimiterNone	caseCaps
paddingSpaces	delimiterAsterisk	caseLower
paddingDashes	delimiter3Asterisks	caseUpper

The values under each property name are implemented as enumerations, and you need not memorize their names. As you enter the name of the property followed by the equal sign, the appropriate list of values will pop up, and you can select the desired member. Listing 10.25 presents the `UseCaseEnum` enumeration and the implementation of the `UseCase` property.

Listing 10.25: The *CaseEnum* Enumeration and the *UseCase* Property

```
Enum CaseEnum
   caseCaps
```

```
      caseLower
      caseUpper
   End Enum

   Private varUseCase As CaseEnum
   Public Property [Case]() As CaseEnum
      Get
         Return (varUseCase)
      End Get
      Set
         varUseCase = Value
      End Set
   End Property
```

Notice that the name of the `Case` property is enclosed in square brackets. This is necessary when you're using a reserved keyword as a variable, property, method, or enumeration member name. Alternatively, you can use a different name for the property to avoid the conflict altogether. After the declaration of the enumeration and the Property procedure are in place, the coding of the rest of the class is simplified a great deal. The `Num2String()` function, for example, calls the `Pad()` method after each three-digit group. The separator is specified by the `UseDelimiter` property, whose type is `clsPadding`. The `Pad()` function uses the members of the `UsePaddingEnum` enumeration to make the code easier to read. As soon as you enter the `Case` keyword, the list of values that can be used in the `Select Case` statement will appear automatically, and you can select the desired member. Here's the code of the `Pad()` function:

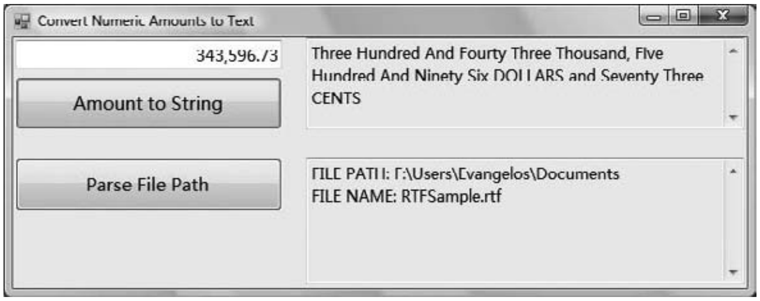
```
Private Function Pad() As String
   Select Case varUsePadding
      Case PaddingEnum.paddingSpaces : Return ("")
      Case PaddingEnum.paddingDashes : Return ("-")
      Case PaddingEnum.paddingCommas : Return (",")
   End Select
End Function
```

To test the StringTools class, create a test form like the one shown in Figure 10.9. Then enter the code from Listing 10.26 in the `Click` event handler of the two buttons.

Listing 10.26: Testing the StringTools Class

```
   Protected Sub Button1_Click(...) Handles Button1.Click
      TextBox1.Text = Convert.ToDecimal( _
                      TextBox1.Text).ToString("#,###.00")
        Dim objStrTools As New StringTools()
        objStrTools.Case = StringTools.CaseEnum.CaseCaps
        objStrTools.Delimiter = StringTools.DelimitEnum.DelimiterNone
        objStrTools.Padding = StringTools.PaddingEnum.PaddingCommas
        TextBox2.Text = objStrTools.Num2String(Convert.ToDecimal(TextBox1.Text))
   End Sub

   Protected Sub Button2_Click(...) Handles Button2.Click
```

```
        Dim objStrTools As New StringTools()
        openFileDialog1.ShowDialog()
        Dim fName as String
        fName = OpenFileDialog1.FileName
        Debug.writeline(objStrTools.ExtractPathName(fName))
        Debug.WriteLine(objStrTools.ExtractFileName(fName))
    End Sub
```



```
343,596.73
```

Three Hundred And Fourty Three Thousand, Five Hundred And Ninety Six DOLLARS and Seventy Three CENTS

```
FILE PATH: F:\Users\Evangelos\Documents
FILE NAME: RTFSample.rtf
```

FIGURE 10.9
The test form of the StringTools class

Operator Overloading

In this section you'll learn about an interesting (but quite optional) feature of class design: how to customize the usual operators. Some operators in Visual Basic act differently on various types of data. The addition operator (+) is the most typical example. When used with numbers, the addition operator adds them. When used with strings, however, it concatenates the strings. The same operator can perform even more complicated calculations with the more-elaborate data types. When you add two variables of the TimeSpan type, the addition operator adds their durations and returns a new TimeSpan object. If you execute the following statements, the value 3882 will be printed in the Output window (the number of seconds in a time span of 1 hour, 4 minutes, and 42 seconds):

```
Dim TS1 As New TimeSpan(1, 0, 30)
Dim TS2 As New TimeSpan(0, 4, 12)
Debug.WriteLine((TS1 + TS2).TotalSeconds.ToString)
```

The TimeSpan class is discussed in detail in Chapter 13, ''Handling Strings, Characters, and Dates,'' but for the purposes of the preceding example, all you need to know is that variable *TS1* represents a time span of 1 hour and 30 seconds, while *TS2* represents a time span of 4 minutes and 12 seconds. Their sum is a new time span of 1 hour, 4 minutes and 42 seconds. So far you saw how to overload methods and how the overloaded forms of a method can simplify development. Sometimes it makes sense to alter the default function of an operator. Let's say you designed a class for representing lengths in meters and centimeters, something like the following:

```
Dim MU As New MetricUnits
MU.Meters = 1
MU.Centimeters = 78
```

The MetricUnits class allows you to specify lengths as an integer number of meters and centimeters (presumably, you don't need any more accuracy). The most common operation you'll perform with this class is to add and subtract lengths. However, you can't directly add two objects of the MetricUnits type by using a statement such as this:

```
TotalLength = MU1 + MU2
```

Wouldn't it be nice if you could add two custom objects by using the addition operator? For this to happen, you should be able to overload the addition operator, just as you can overload a method. Indeed, it's possible to overload an operator for your custom classes and write statements like the preceding one. Let's design a class to express lengths in metric and English units, and then overload the basic operators for this class.

To overload an operator, you must create an `Operator` procedure, which is basically a function with an odd name: the name of the operator you want to overload. The `Operator` procedure accepts as arguments two values of the custom type (the type for which you're overloading the operator) and returns a value of the same type. Here's the outline of an `Operator` procedure that overloads the addition operator:

```
Public Shared Operator + ( _
           ByVal length1 As MetricUnits, _
           ByVal length2 As MetricUnits) As MetricUnits
End Operator
```

The procedure's body contains the statements that add the two arguments as units of length, not as numeric values. Overloading operators is a straightforward process that can help you create elegant classes that can be manipulated with the common operators.

VB 2008 at Work: The LengthUnits Class

To demonstrate the overloading of common operators, I included the LengthUnits project, which is a simple class for representing distances in English and metric units. Listing 10.27 shows the definition of the MetricUnits class, which represents lengths in meters and centimeters.

Listing 10.27: The MetricUnits Class

```
Public Class MetricUnits
    Private _Meters As Integer
    Private _Centimeters As Integer

    Public Sub New()

    End Sub

    Public Sub New(ByVal meters As Integer, ByVal centimeters As Integer)
        Me.Meters = meters
        Me.Centimeters = centimeters
    End Sub

    Public Property Meters() As Integer
```

```
        Get
            Return _Meters
        End Get
        Set(ByVal Value As Integer)
            _Meters = Value
        End Set
    End Property

    Public Property Centimeters() As Integer

        Get
            Return _Centimeters
        End Get
        Set(ByVal Value As Integer)
            If value > 100 Then
                _Meters += Convert.ToInt32(Math.Floor(Value / 100))
                _Centimeters = (Value Mod 100)
            Else
                _Centimeters = value
            End If
        End Set
    End Property

    Public Overloads Function Tostring() As String
        Dim str As String = Math.Abs(_Meters).ToString & " meters, " & _
                    Math.Abs(_Centimeters).ToString & " centimeters"
        If _Meters < 0 Or (_Meters = 0 And _Centimeters < 0) Then
            str = "-" & str
        End If
        Return str
    End Function
End Class
```

The class uses the private variables *_Meters* and *_Centimeters* to store the two values that determine the length of the current instance of the class. These variables are exposed as the Meters and Centimeters properties. Notice the two forms of the constructor and the custom ToString method. Because the calling application may supply a value that exceeds 100 for the Centimeters property, the code that implements the Centimeters property checks for this condition and increases the Meters property, if needed. It allows the calling application to set the Centimeters property to 252, but internally it increases the *_Meters* local variable by 2 and sets the *_Centimenters* local variable to 52. The ToString method returns the value of the current instance of the class as a string such as 1.98, but it inserts a minus sign in front of it if it's negative. If you open the sample project, you'll find the implementation of the EnglishUnits class, which represents lengths in feet and inches. The code is quite similar.

There's nothing out of the ordinary so far; it's actually a trivial class. We can turn it into a highly usable class by overloading the basic operators for the MetricUnits class: namely the addition and subtraction operators. Add the Operator procedures shown in Listing 10.28 to the class's code.

Listing 10.28: Overloading Operators for the MetricUnits Class

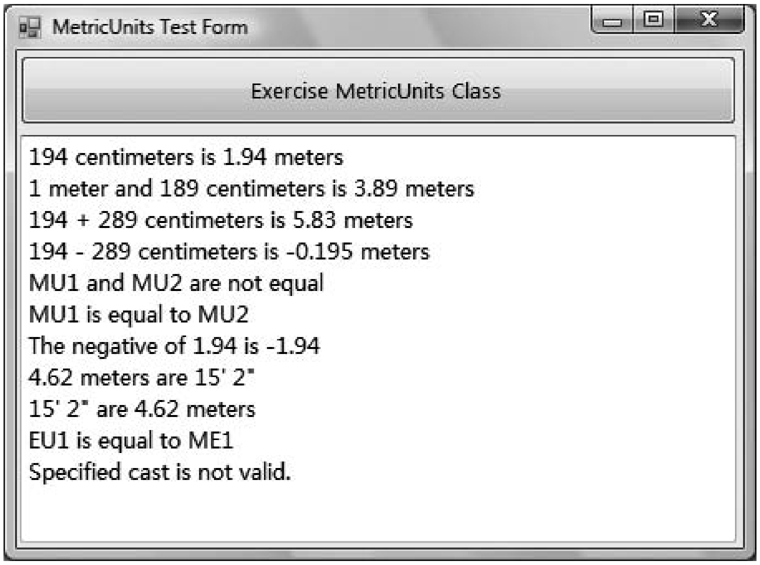
```
Public Shared Operator + ( _
                ByVal length1 As MetricUnits, _
                ByVal length2 As MetricUnits) As MetricUnits
    Dim result As New metricUnits
    result.Meters = 0
    result.Centimeters = _
        length1.Meters * 100 + length1.Centimeters + _
        length2.Meters * 100 + length2.Centimeters
    Return result
End Operator

Public Shared Operator - ( _
                ByVal length1 As MetricUnits, _
                ByVal length2 As MetricUnits) As MetricUnits
    Dim result As New MetricUnits
    result.Meters = 0
    result.Centimeters = _
         length1.Meters * 100 + length1.Centimeters - _
         length2.Meters * 100 - length2.Centimeters
    Return result
End Operator
```

These two procedures turned an ordinary class into an elegant custom data type. You can now create MetricUnits variables in your code and manipulate them with the addition and subtraction operators as if they were simple numeric data types. The following code segment exercises the MetricUnits class:

```
Dim MU1 As New MetricUnits
MU1.Centimeters = 194
Debug.WriteLine("194 centimeters is " & MU1.Tostring & " meters")
194 centimeters is 1.94 meters
Dim MU2 As New MetricUnits
MU2.Meters = 1
MU2.Centimeters = 189
Debug.WriteLine("1 meter and 189 centimeters is " & MU2.Tostring & " meters")
1 meter and 189 centimeters is 2.89 meters
Debug.WriteLine("194 + 289 centimeters is " & (MU1 + MU2).Tostring & " meters")
194 + 289 centimeters is 4.83 meters
Debug.WriteLine("194 - 289 centimeters is " & (MU1 - MU2).Tostring & " meters")
The negative of 1.94 is -1.94
MU1.Meters = 4
MU1.Centimeters = 63
Dim EU1 As EnglishUnits = CType(MU1, EnglishUnits)
Debug.WriteLine("4.62 meters are " & EU1.Tostring)
4.62 meters are 15' 2"
MU1 = CType(EU1, MetricUnits)
```

```
Debug.WriteLine(EU1.Tostring & " are " & MU1.Tostring & " meters")
15' 2" are 4.62 meters
```

If you execute the preceding statements, the highlighted values will appear in the Output window. (The LengthUnits sample project uses a TextBox control to display its output.) Figure 10.10 shows the test project for the MetricUnits and EnglishUnits classes. The last few statements convert values between metric and English units, and you'll see the implementation of these operations momentarily.



```
194 centimeters is 1.94 meters
1 meter and 189 centimeters is 3.89 meters
194 + 289 centimeters is 5.83 meters
194 - 289 centimeters is -0.195 meters
MU1 and MU2 are not equal
MU1 is equal to MU2
The negative of 1.94 is -1.94
4.62 meters are 15' 2"
15' 2" are 4.62 meters
EU1 is equal to ME1
Specified cast is not valid.
```

FIGURE 10.10
Exercising the members of the MetricUnits class

IMPLEMENTING UNARY OPERATORS

In addition to being the subtraction operator, the minus symbol is also a *unary operator* (it negates the following value). If you attempt to negate a MetricUnits variable, an error will be generated because the subtraction operator expects two values — one on either side of it. In addition to the subtraction operator (which is a binary operator because it operates on two values), we must define the negation operator (which is a unary operator because it operates on a single value). The unary minus operator negates the following value, so a new definition of the subtraction `Operator` procedure is needed. This definition will overload the existing one, as follows:

```
Public Overloads Shared Operator -( _
          ByVal length1 As MetricUnits) As MetricUnits
    Dim result As New MetricUnits
    result.Meters = -length1.Meters
    result.Centimeters = -length1.Centimeters
    Return result
End Operator
```

To negate a length unit stored in a variable of the MetricUnits type, use statements such as the following:

```
MU2 = -MU1
Debug.Write(MU2.Tostring)
Debug.Write((-MU1).Tostring)
```

Both statements will print the following in the Output window:

```
-1 meters, -94 centimeters
```

There are several unary operators, which you can overload in your custom classes as needed. There's the unary + operator (not a common operator), and the `Not`, `IsTrue`, and `IsFalse` operators, which are logical operators. The last unary operator is the `CType` operator, which is exposed as a method of the custom class and is explained next.

Handling Variants

To make your custom data type play well with the other data types, you must also provide a `CType()` function that can convert a value of the MetricUnits type to any other type. It doesn't make much sense to convert MetricUnits to dates or any of the built-in objects, but let's say you have another class: the EnglishUnits class. This class is similar to the MetricUnits class, but it exposes the `Inches` and `Feet` properties in place of the `Meters` and `Centimeters` properties. The `CType()` function of the MetricUnits class, which will convert MetricUnits to EnglishUnits, is shown next:

```
Public Overloads Shared Widening Operator _
                 CType(ByVal MU As MetricUnits) As EnglishUnits
    Dim EU As New EnglishUnits
    EU.Inches = Convert.ToInt32( _
                (MU.Meters * 100 +  MU.Centimeters) / 2.54)
    Return EU
End Operator
```

Do you remember the implicit narrowing and widening conversions we discussed in Chapter 2? An attempt to assign an integer value to a decimal variable will produce a warning, but the statement will be executed because it's a widening conversion (no loss of accuracy will occur). The opposite is not true. If the Strict option is on, the compiler won't allow narrowing conversions because not all Decimal values can be mapped to Integers. To help the compiler enforce strict types, you can use the appropriate keyword to specify whether the `CType()` function performs a widening or a narrowing conversion. The `CType()` procedure is shared and overloads the default implementation, which explains all the keywords prefixing its declaration. The following statements exercise the `CType` method of the MetricUnits class:

```
Debug.Write(MU1.Tostring)
1 meters, 94 centimeters
Debug.WriteLine(CType(MU1, EnglishUnits).Tostring)
6 feet, 4 inches
```

The output of the two statements is highlighted. Both classes expose integer properties, so the `Widening` or `Narrowing` keyword isn't really important. In other situations, you must carefully specify the type of the conversion to help the compiler generate the appropriate warnings (or exceptions, if needed).

The `CType` operator we added to the MetricUnits class can only convert values of the MetricUnit type to values of the EnglishUnit type. If it makes sense to convert MetricUnits variables to other types, you must provide more overloaded forms of the `CType()` procedure. For example, you can convert them to numeric values (the numeric value could be the length in centimeters or a double value that represents the same length in meters). The compiler sees the return type(s) of the various overloaded forms of the `CType` operator, knows whether the requested conversion is possible, and generates the appropriate exception.

In short, operator overloading isn't complicated, but it adds a touch of elegance to a custom class and enables variables of this type to mix well with the other data types. If you like math, you could implement classes to represent matrices, or complex numbers, and overload the usual operators for addition, multiplication, and so on.

The Bottom Line

Build your own classes. Classes contain code that executes without interacting with the user. The class's code is made up of three distinct segments: the declaration of the private variables, the property procedures that set or read the values of the private variables, and the methods, which are implemented as Public subroutines or functions. Only the Public entities (properties and methods) are accessible by any code outside the class. Optionally, you can implement events that are fired from within the class's code. Classes are referenced through variables of the appropriate type, and applications call the members of the class through these variables. Every time a method is called, or a property is set or read, the corresponding code in the class
is executed.

Master It How do you implement properties and methods in a custom class?

Master It How would you use a constructor to allow developers to create an instance of your class and populate it with initial data?

Use custom classes in your projects. To use a custom class in your project, you must add to the project a reference to the class you want to use. If the class belongs to the same project, you don't have to do anything. If the class belongs to another project, you must right-click the project's name in the Solution Explorer and select Add Reference from the shortcut menu. In the Add Reference dialog box that appears, switch to the Browse tab and locate the DLL file with the class's implementation (it will be a DLL file in the project's Bin folder). Select the name of this file and click OK to add the reference and close the dialog box.

Master It How will you call the two constructors of the preceding Master It sections in an application that uses the custom class to represent books?

Customize the usual operators for your classes. Overloading is a common theme in coding classes (or plain procedures) with Visual Basic. In addition to overloading methods, you can overload operators. In other words, you can define the rules for adding or subtracting two custom objects, if this makes sense for your application.

Master It When should you overload operators in a custom class, and why?

Chapter 11

Working with Objects

This chapter continues the discussion of object-oriented programming (OOP) and covers some of its more-advanced, but truly useful, concepts: inheritance and polymorphism. Instead of jumping to the topic of inheritance, I'll start with a quick overview of what you learned in the previous chapter and how to apply this knowledge.

Inheritance is discussed later in this chapter, along with polymorphism, another powerful OOP technique, and interfaces. But first make sure that you understand the basics of OOP because things aren't always as simple as they look (but are quite often simpler than you think).

In this chapter, you'll learn how to:

- Extend existing classes using inheritance
- Develop flexible classes using polymorphism

Issues in Object-Oriented Programming

Building classes and using them in your code is fairly simple, but there are a few points about OOP that can cause confusion. To help you make the most of OOP and get up to speed, I'm including a list of related topics that are known to cause confusion to programmers — and not only beginners. If you understand the topics of the following paragraphs and how they relate to the topics discussed in the previous chapter, you're more than familiar with the principles of OOP and you can apply them to your projects immediately.

Classes versus Objects

Classes are templates that we use to create new objects. In effect, they're the blueprints used to manufacture objects in our code. Another way to think of classes is as custom types. After you add the class Customer to your project (or a reference to the DLL that implements the Customer class), you can declare variables of the Customer type, just as you declare integers and strings. The code of the class is loaded into the memory, and a new set of local variables is created. This process is referred to as *class instantiation*: Creating an object of a custom type is the same as instantiating the class that implements the custom type. For each object of the Customer type, there's a set of local variables, as they're declared in the class's code. The various procedures of the class are invoked as needed by the Common Language Runtime (CLR) and they act on the set of local variables that correspond to the current instance of the class. Some of the local variables may be common among all instances of a class: These are the variables that correspond to shared properties (properties that are being shared by all instances of a class).

When you create a new variable of the Customer type, the `New()` procedure of the Customer class is invoked. The `New()` procedure is known as the class's *constructor*. Each class has a default constructor that accepts no arguments, even if the class doesn't contain a New() subroutine. This default constructor is invoked every time a statement similar to the following is executed:

```
Dim cust As New Customer
```

You can overload the New() procedure by specifying arguments, and you should try to provide one or more parameterized constructors. Parameterized constructors allow you (or any developer using your class) to create meaningful instances of the class. Sure, you can create a new Customer object with no data in it, but a Customer object with a name and company makes more sense. The parameterized constructor initializes some of the most characteristic properties of the object.

Objects versus Object Variables

All variables that refer to objects are called *object variables*. (The other type of variables are *value variables*, which store base data types, such as characters, integers, strings, and dates.) In declaring object variables, we usually use the New keyword, which is the only way to create a new object. If you omit this keyword from a declaration, only a variable of the Customer type will be created, but no instance of the Customer class will be created in memory, and the variable won't point to an actual object. The following statement declares a variable of the Customer type, but doesn't create an object:

```
Dim Cust As Customer
```

If you attempt to access a member of the Customer class through the *Cust* variable, the infamous NullReferenceException will be thrown. The description of this exception is *Object reference not set to an instance of an object*, which means that the Cust variable doesn't point to an instance of the Customer class. Actually, the editor will catch this error and will underline the name of the variable. If you hover the mouse pointer over the name of the variable in question, the following explanation will appear on a ToolTip box: *Variable Cust is used before it has been assigned a value. A Null Reference exception could result at runtime.* Why bother declaring variables that don't point to specific objects? The *Cust* variable can be set later in the code to reference an existing instance of the class:

```
Dim Cust As Customer
Dim Cust2 As New Customer
Cust = Cust2
```

After the execution of the preceding statements, both variables point to the same object in memory, and you can set the properties of this object through either variable. You have two object variables, but only one object in memory because only one of them was declared with the New keyword. To set the Company property, you can use either one of the following statements, because they both point to the same object in memory:

```
Cust.CompanyName = "New Company Name"
```

or

```
Cust2.CompanyName = "New Company Name"
```

The `Cust` variable is similar to a shortcut. When you create a shortcut to a specific file on your desktop, you're creating a reference to the original file. You do not create a new file or a copy of the original file. You can use the shortcut to access the original file, just as we can use the `Cust` variable to manipulate the properties of the `Cust2` object in the preceding code sample.

It's also common to declare object variables without the `New` keyword when we know we're going to use them later in our code, as shown in the following loop:

```
Dim LI As ListViewItem
For row = 0 To 20
    LI = New ListViewItem
    LI.Text = "....."
    ' more statements to set up the LI variable
    ListView1.Items.Add(LI)
Next
```

The `LI` variable is declared once, and we initialize it many times in the following loop. The first statement in the loop creates a new ListViewItem object, and the last statement adds it to the ListView control. Another common scenario is to declare an object variable without initializing it at the form's level and initialize it in a procedure, while using its value in other procedures.

When to Use the *New* Keyword

Many programmers are confused by the fact that most object variables must be declared with the `New` keyword, whereas some types don't support the `New` keyword. If you want to create a new object in memory (which is an instance of a class), you must use the `New` keyword. When you declare a variable without the `New` keyword, you're creating a reference to an object, but not a new object. Only shared classes must be declared without the `New` keyword. If in doubt, use the `New` keyword anyway, and the compiler will let you know immediately whether the class you're instantiating has a constructor. If the `New` keyword is underlined in error, you know that you must delete the `New` keyword from the declaration.

Exploring Value Types

Okay, if the variables that represent objects are called *object variables* and the types they represent are called reference types, what other variables are there? They're the regular variables that store the basic data types, and they're called *value variables* because they store values. An integer, or a string, is not stored as an object for efficiency. An Integer variable contains an actual value, not a pointer to the value. Imagine if you had to instantiate the Integer class every time you needed to use an Integer value in your code. Not that it's a bad idea, but it would scare away most VB developers. Value variables are so common in programming and they're not implemented as classes for efficiency. Whereas objects require complicated structures in memory, the basic data types are stored in a few bytes and are manipulated much faster than objects.

Consider the following statements:

```
Dim age1, age2 As Integer
age2 = 29
age1 = age2
age2 = 40
```

When you assign a value variable to another, the actual value stored in the variable overwrites the current value of the other variable. The two variables have the same value after the statement that assigns the value of *age2* to the variable *age1*, but they're independent of one another. After the execution of the last statement, the values of *age1* and *age2* are different again. If they were object variables, they would point to the same object after the assignment operation, and you wouldn't be able to set their values separately. You'd be setting the properties of the same object.

Value types are converted to objects as soon as you treat them as objects. As soon as you enter a statement like the following, the intValue variable is converted to an object:

```
intValue.MinValue
```

You'll rarely use the methods of the base types, but you can turn value variables into object variables at any time. This process is known as *boxing* (the conversion of a value type to an object).

Exploring Reference Types

To better understand how reference types work, consider the following statements that append a new row with two subitems to a ListView control. (The control's item is an object of the ListView Item type.):

```
ListView1.Items.Clear
Dim LI As New ListViewItem
LI.Text = "Item 1"
LI.SubItems.Add("Item 1 SubItem 1.a")
LI.SubItems.Add("Item 1 SubItem 1.b")
ListView1.Items.Add(LI)
```

After the execution of the preceding statements, the ListView control contains a single row. This row is an object of the ListViewItem type and exists in memory on its own. Only after the execution of the last statement is the ListViewItem object referenced by the *LI* variable associated with the ListView1 control.

To change the text of the first item, or its appearance, you can manipulate the control's Items collection directly, or change the *LI* variable's properties. The following pairs of statements are equivalent:

```
ListView1.Items(0).Text = "Revised Item 1"
ListView1.Items(0).BackColor = Color.Cyan
```

and

```
LI.Text = "Revised Item 1"
LI.BackColor = Color.Cyan
```

There's yet another method to access the ListView control's items. Create an object variable that references a specific item and set the item's properties through this variable:

```
Dim selItem As ListViewItem
selItem = ListView1.Items(0)
selItem.Text = "new caption"
selItem.BackColor = Color.Silver
```

These statements are not new to you; you’ve seen these techniques in action in Chapter 9, ‘‘The TreeView and ListView Controls,’’ where you learned to program the ListView control. Now you can really understand how they work.

A final question for testing your OOP skills: What do you think will happen if you set the *LI* variable to nothing? Should the control’s row disappear? The answer is no. If you thought otherwise, take a moment now to think about why deleting a variable doesn’t remove the object from memory. The *LI* variable points to an object in memory; it’s not the object. The New keyword created a new ListViewItem object in memory and assigned its address to the variable *LI*. The statement that added the *LI* variable to the control’s Items collection associated the object in memory with the control. By setting the *LI* variable to nothing, we simply removed the pointer to the ListViewItem object in memory, not the object itself. To actually remove the control’s first item, you must call the Remove method of the *LI* variable:

```
LI.Remove
```

This statement will remove the ListViewItem object from the control’s Items collection, but the actual object still lives in the memory. If you execute the following statement, the item will be added again to the control:

```
ListView1.Items.Add(LI)
```

So to sum up, the ListViewItem object exists in memory and is referenced by the *LI* variable. The Remove method removes the item from the control; it doesn’t delete it from the memory. If you remove the item from the control and then set the *LI* variable to Nothing, the object will also be removed from memory.

By the way, the ListViewItem object won’t be deleted instantly. The CLR uses a special mechanism to remove objects from memory, the Garbage Collector (GC). The GC runs every so often and removes from memory all objects that are not referenced by any variable. These objects eventually will be removed from memory, but we can’t be sure when. (There’s no way to force the GC to run on demand.) The CLR will start the GC based on various criteria (the current CPU load, the amount of available memory, and so on). Because objects are removed automatically by the CLR, we say that the lifetime of an object is *nondeterministic*. However, you can rest assured that the object will eventually be removed from memory. After you set the *LI* variable to Nothing and remove the corresponding item from the ListView control, you’re left with a ListViewItem object in memory that’s not referenced by any other entity. This object will live a little longer in the memory, until the GC gets a chance to remove it and reclaim the resources allocated to the object.

Here are the statements I’ve used for this experiment:

```
' Create a new ListViewItem object
Dim LI As New ListViewItem
LI.Text = "Item 1"
LI.SubItems.Add("Item 1 SubItem 1.a")
LI.SubItems.Add("Item 1 SubItem 1.b")
' add it to the ListView control
ListView1.Items.Add(LI)
MsgBox("Item added to the list." & vbCrLf & _
       "Click OK to modify the appearance " & _
       "of the top item through the LI variable.")
```

```
' Edit the object's properties
' The new settings will affect the appearance of the
' item on the control immediately
LI.Text = "ITEM 1"
LI.Font = New Font("Verdana", 10, FontStyle.Regular)
LI.BackColor = Color.Azure
MsgBox("Item's text and appearance modified." & _
         vbCrLf & "Click OK to modify the " & _
         "appearance of the top item through " & _
         "the ListView1.Items collection.")
' Change the first item on the control directly
' Changes also affect the object in memory
ListView1.Items(0).BackColor = Color.LightCyan
LI.SubItems(2).Text = "Revised Subitem"
' Remove the top item from the control
MsgBox("Will remove the top item from the control.")
LI.Remove()
MsgBox("Will restore the deleted item")
' The item was removed from list, but not deleted
' We can add it to the control's Items collection
ListView1.Items.Add(LI)
MsgBox("Will remove object from memory")
' Remove it again from the control
LI.Remove()
' and set it to Nothing
LI = Nothing
' We can no longer access the LI object.
MsgBox("Can I access it again? " & vbCrLf & _
       "NO, YOU'LL GET AN EXCEPTION WHEN THE " & _
       "FOLLOWING STATEMENT IS EXECUTED!")
ListView1.Items.Add(LI)
```

Properties versus Fields

When you set or read a property's value, the corresponding `Get` or `Set` segment of the Property procedure is executed. The following statement invokes the Property `Set` segment of the `EMail` public property of the class:

```
cust.EMail = "Evangelos.P@Sybex.com"
```

Obviously, every time you call one of the class's properties, the corresponding public procedure in the class is invoked. The following statement invokes both the `Set` and `Get` property procedures of the Customer class's `Balance` property:

```
cust.Balance = cust.Balance + 429.25
```

Trivial properties can also be implemented as public variables. These variables, which are called fields, behave like properties, but no code is executed when the application sets or reads their value. We often implement properties of the enumeration type as *fields* because they can be set only to valid values. If the `Set` method of a property doesn't contain any validation code and simply assigns a new value to the local variable that represents the specific property, there's no difference between the property and a field.

Shared versus Instance Members

To really understand classes and appreciate them, you must visualize the way classes combine code and data. Properties contain the data that live along with the code, which determines the object's behavior — its functionality. The functionality of the object is implemented as a number of methods and events. The properties, methods, and events constitute the class's interface. Each instance of the class acts on its own data, and there's no interference between two objects of the same type unless they contain shared properties. A *shared property* is common to all instances of the class. In other words, there's no local variable for this property, and all instances of the class access the same variable. Shared properties are not common — after all, if many of the properties are common to all instances of the class, why create many objects? Shared methods, on the other hand, are quite common. The Math class is a typical example. To calculate the logarithm of a number, you call the `Log` method of the Math class:

```
Math.Log(123)
```

You need not create an instance of the Math class before calling any of its methods (which are the common math functions). Actually, you can't create a new instance of the Math class because the entire class is marked as shared.

Let's say you're building a class to represent customers, the Customer class. This class should expose properties that correspond to the columns of the Customers table in a database. Each instance of the Customer class stores information about a specific customer. In addition to the properties, the Customer class should expose a few methods to get data from the database and commit changes or new customers to the database. The `GetCustomerByID` method, for example, should accept the ID of a customer as an argument, retrieve the corresponding customer's data from the database, and use them to populate the current instance's properties. Here's how you use this class in your code:

```
Dim cust As New Customer
cust.GetCustomerByID("ALFKI")
Debug.WriteLine cust.CompanyName
Debug.WriteLine cust.ContactName & "   " & cust.ContactTitle
```

The `GetCustomerByID` method can retrieve the customer data from a local database, a remote web service, or even an XML file. The idea is that a single method call gets the data and uses it to populate the properties of the current instance of the class. This method is an instance method because it requires an instance of the class. It populates the properties of this instance, or object.

You could have implemented the `GetCustomerByID` method as a shared method, but then the method should return an object of the Customer type. The shared method can't populate any object's properties, because it can't be applied to an instance of the class. Here's how you'd use the Customer class if the `GetCustomerByID` method were shared:

```
Dim cust As New Customer
cust = Customer.GetCustomerByID("ALFKI")
Debug.WriteLine cust.CompanyName
Debug.WriteLine cust.ContactName & "   " & cust.ContactTitle
```

As you can see, we call the method of the Customer class, not the method of an object. You could also call the method with the following statement, but the code becomes obscure (at the very least, it's not elegant):

```
cust = cust.GetCustomerByID("ALFKI")
```

The background compiler will detect that you're attempting to access a shared method through an instance of the class and will generate the following warning. (The expression will be evaluated at runtime, in spite of the warning.)

```
Access of shared member, constant member,
enum member or nested type through an instance;
 qualifying expression will not be evaluated.
```

Because the class needs to know the database in which the data is stored, you can provide a `Connection` property that's shared. Shared properties are usually set when the class is initialized, or from within a method that's called before we attempt to access any other methods, or any of the class's properties. All the methods in the class use the `Connection` property to connect to the database. There's no reason to change the setting of this property in the course of an application, but if you change it, all subsequent operations should switch to the new database.

In summary, any class may expose a few shared properties, if all instances of the class should access the same property value. It may also expose a few shared methods, which can be called through the class name, if there's no need to create an instance of the class in order to call a method. In extreme situations, you can create a shared class: All properties and methods of this class are shared by default.

Type Casting

The data type used most in earlier versions of the language up to VB 6 was the Variant (which was replaced in subsequent versions by the Object type). A variable declared as Object can store anything, and any variable that hasn't been declared explicitly is an Object variable. Even if you turn on the Strict option, which forces you to declare the type of each variable (and you should always have this option on), you will eventually run into Object variables. When you retrieve an item from a ListBox control, for example, you get back an object, not a specific data type. In the previous chapter, we used the ListBox control to store Contact objects. Every time we retrieved a contact from the control's `Items` collection, however, we got back an Object variable. To use this object in our code, we had to convert it to a more specific type, the Contact type, with the `CType()` or `DirectCast` functions. The same is true for an ArrayList, which stores objects, and we usually cast its members to specific types.

Variables declared without a specific type are called *untyped variables*. Untyped variables should be avoided — and here's why. The following expression represents a ListBox item, which is an object:

```
ListBox1.Items(3)
```

Even if you add a Customer or a Product object to the list, when you retrieve the same item, it's returned as a generic Object variable. If you type the preceding expression followed by a period, you will see in the IntelliSense drop-down list the members of the generic Object variable, which you hardly ever need. If you cast this item to a specific type, the IntelliSense box will show the members of the appropriate type.

The action of changing a variable's type is known as *casting*, and there are two methods for casting variable types — the old VB 6 CType() function and the new DirectCast() function:

```
Dim currentCustomer As Customer
currentCustomer = CType(ListBox1.Items(3), Customer)
```

From now on, you can access the members of the *currentCustomer* object as usual.

THE *TRYCAST()* FUNCTION

If the specified type conversion can't be carried out, the CType() function will throw an Invalid-CastException exception. As a reminder, a variation of the CType() and DirectCast() functions is the TryCast() function, which attempts to convert a variable into another type. If the conversion is not possible, the TryCast() function doesn't throw an exception, but returns the Nothing value. Here's how the TryCast() function is used:

```
Dim o As Object
o = New Customer("Evangelos Petroutsos", "SYBEX")
c = TryCast(o, Contact)
If c Is Nothing Then
    MsgBox("Can't convert " & o.GetType.Name & " to Contact")
    Exit Sub
End If
' statements to process the c object variable
```

Early versus Late Binding

Untyped variables can't be resolved at compile time; these variables are said to be *late-bound*. An expression such as the following can't be resolved at compile time because the compiler has no way of knowing whether the object retrieved from the ListBox control is of the Customer type (or any other type that exposes the LastName property):

```
ListBox1.Items(3).LastName
```

The preceding statement will compile and execute fine if the fourth item on the ListBox control is of the Customer type or any other type that provides a LastName property. If not, it will compile all right, but a runtime exception will be thrown. Moreover, you won't see any members of interest in the IntelliSense box, because the editor doesn't know the exact type of the object retrieved from the ListBox control.

If you cast the object to a specific type, the compiler won't let you reference a nonexisting member, therefore eliminating the chances of runtime exceptions. The last expression in the following code segment is said to be *early-bound* because the compiler knows its type and won't compile a statement that references nonexisting members:

```
Dim currentCustomer As Customer
currentCustomer = CType(ListBox1.Items(3), Customer)
Debug.WriteLine currentCustomer.LastName
```

Casting an object to the desired type won't help you, unless you know that the object is of the same type or can be cast to the desired type. Make your code as robust as it can be by using the `TryCast()` function to make sure that the conversion succeeded before attempting to use the *`currentCustomer`* object in your code.

Discovering a Variable's Type

Sometimes we need to figure out the type of a variable in our code. Even if you declare explicitly all the variables in your code, you might have to discover a specific variable's type at runtime.

The Form object exposes the `ActiveControl` property, which is the control that has the focus. The `ActiveControl` property returns a Control object, and you will have to find out its exact type (whether it's a TextBox, a ComboBox, or a Button, for example) from within your code.

All classes, including custom ones, expose the `GetType()` function, which returns the type of the corresponding object. The `GetType()` function's return value isn't a string; it is an object that exposes a large number of properties. You can call the `IsEnum` and `IsClass` properties to find out whether it's been implemented as an enumeration or as a class, as well as the `Name` property to retrieve the variable's type name.

Consider an event handler that handles the same event for multiple controls on a form. The control that raised the event is passed to the event handler through the *`sender`* argument, and you can determine the type of the control that raised the event by using a statement such as the following:

```
If sender.GetType Is GetType(System.Windows.Forms.Button) Then
  ' process a button control
End If
```

You can also retrieve the type's name with the `TypeName()` function, which returns a string:

```
If TypeName(newContact).ToUpper="CONTACT" Then
```

Because the `TypeName()` function returns a string, you don't have to use the `Is` operator, but it's a good idea to convert this value to uppercase before attempting any comparisons.

Notice that you can't use the equals operator to compare types. To compare an object's type to another type, you must use the `Is` and `IsNot` keywords, as shown in the preceding example.

By now you should have a good understanding of developing with objects. In the following section, you're going to learn about a powerful concept in OOP, namely how to write new classes that inherit the functionality of existing ones.

Inheritance

Here's a scenario we're all too familiar with: You've written some code, perhaps a collection of functions, which you want to reuse in another project. The key word here is *reuse*: *write once, use many times*. For years, VB developers were reusing code, even sharing it with others, with a very simple method: copying from one project and pasting it into another. The copy/paste approach to code reuse has never really worked because the code was never left untouched at its destination. In the process of reusing the original code in another project, we make changes to better accommodate the new project. In the process, we also improve the code. At some point, we decide that we should ''return'' the improved code to the original project and enhance it. Unfortunately, the improved code doesn't always fit nicely into a different project. Some of the improvements break applications that used to work with the not-so-good code. If this has happened to you, imagine what a mess code sharing can be in a large environment with dozens of programmers. On a corporate level, this form of code reuse is a nightmare.

The promise of OOP is code reuse. The functionality you place into a class is there for your projects, and any other developer can access it as well. *Inheritance* is a technique for reusing and improving code without breaking the applications that use it. The idea is to export the code we want to reuse in a format that doesn't allow editing. If more than two people can edit the same code (or even a single person is allowed to edit the same code in two different projects), any benefits of code reuse evaporate immediately. The code to be shared must be packaged as a DLL, which exposes all the functionality without the risk of being modified in a haphazard way. Only the original creator of the DLL can edit the code, and it's likely that this person will make sure that the interface of the class doesn't change. However, we should still be able to enhance the code in different projects. That's where inheritance comes into the picture. Instead of getting a copy of the code, we inherit a class. The functionality of the class can't change. The code in the DLL is well protected, and there's no way to edit the executable code; it's the class's functionality we inherit.

However, it's possible to add new functionality to the inherited code or even override some of the existing functionality. We can add new functionality to the code by adding new members to the existing classes. This doesn't break any existing applications that use the original DLL. We can also override some of the functionality by creating a new method that replaces an existing one. Applications that use the original version of the DLL won't see the new members because they work with the old DLL. Newer projects can use the enhanced functionality of the DLL. The current solution to the problem of code reuse is inheritance. It's not a panacea, but it's a step forward.

How to Apply Inheritance

Let me give a simple but quite practical example. A lot of functionality has been built into Windows itself, and we constantly reuse it in our applications. The various Windows Forms controls are a typical example. The functionality of the TextBox control, which we all take for granted, is packaged in a DLL (the System.Windows.Forms.TextBox class). Yet, many of us enhance the functionality of the TextBox control to address specific application requirements. Many developers add a few statements in the control's `Enter` and `Leave` events to change the color of the TextBox control that has the focus. With VB 2008, it's possible to write just two event handlers that react to these two events and control the background color of the TextBox with the focus. These two handlers handle the corresponding events of all TextBox controls on the form.

A better approach is to design a ''new'' TextBox control that incorporates all the functionality of the original TextBox control, and also changes its background color while it has the focus. The code that implements the TextBox control is hidden from us, but we can reuse it by building a new control that inherits from the TextBox control. As you will see in Chapter 12, ''Building Custom Windows Controls,'' this is not only possible, but almost trivial; we'll build an enhanced TextBox control with a few lines of code. Actually, it'll be more convincing if I show you the code right now, so here's the code that implements an enhanced TextBox control, the FocusedTextBox control. (I copied from an example in Chapter 12.):

```
Public Class FocusedTextBox
    Inherits System.Windows.Forms.TextBox
    Private Sub FocusedTextBox_Enter(ByVal sender As Object, _
                 ByVal e As System.EventArgs) Handles Me.Enter
```

```
        Me.BackColor = _enterFocusColor
    End Sub

    Private Sub FocusedTextBox_Leave(ByVal sender As Object, _
                ByVal e As System.EventArgs) Handles Me.Leave
        Me.BackColor = _leaveFocusColor
    End Sub
End Class
```

As you understand, the two Color variables are properties of the control (implemented with the usual setters and getters), so that different applications can use different colors for the active TextBox control on the form.

With the `Inherits` statement, we include all the functionality of the original TextBox control without touching the control's code. (The `Inherits` statement is stored in a different file from the rest of the code, but this a technicality I'll address in the following chapter.) Any project that uses the FocusedTextBox control can take advantage of the extra functionality, yet all existing projects will continue to work with the original version of the control. We can easily upgrade a project to take advantage of the enhanced TextBox control by replacing all the instances of the TextBox control on a form with instances of the new control. Some projects may use the new control, yet not take advantage of the new functionality and leave the default colors — in which case the enhanced control behaves just like the original TextBox control.

Inheritance is simply the ability to create a new class based on an existing one. The existing class is the *parent class*, or *base class*. The new class is said to *inherit* the base class and is called a *subclass*, or *derived class*. The derived class inherits all the functionality of the base class and can add new members and replace existing ones. The replacement of existing members with other ones is called *overriding*. When you replace a member of the base class, you're overriding it. Or, you can overload a method by providing multiple forms of the same method that accept different arguments.

To understand how useful inheritance is to team development, I'll start with an example of extending an existing class, which is part of the Framework. As you can guess, I will inherit the functionality of an existing class, because I can't touch the code of the Framework and introduce any ''improvements.''

Inheriting Existing Classes

To demonstrate the power of inheritance, we'll extend an existing class: the ArrayList class. This class comes with the Framework and is a dynamic array. (See Chapter 14, ''Storing Data in Collections,'' for a detailed description of the ArrayList class.) The ArrayList class maintains a list of objects, similar to an array, but it's dynamic. The class we'll develop in this section will inherit all the functionality of ArrayList, plus it will expose a custom method we'll implement here: the `EliminateDuplicates` method. The project described in this section is the CustomArrayList sample project.

Let's call the new class myArrayList. The first line in the new class must be the `Inherits` statement, followed by the name of the class we want to inherit, ArrayList. Start a new project, name it **CustomArrayList**, and add a new class to it. Name the new class **myArrayList**:

```
Class myArrayList
Inherits ArrayList

End Class
```

If you don't add a single line of code to this class, the myArrayList class will expose exactly the same functionality as the ArrayList class. If you add a public function to the class, it will become a method of the new class, in addition to the methods of ArrayList. Add the code of the `EliminateDuplicates()` subroutine (see Listing 11.1) to the myArrayList class; this subroutine will become a method of the new class.

LISTING 11.1: *EliminateDuplicates* Method for the ArrayList Class

```
Public Sub EliminateDuplicates()
   Dim i As Integer = 0
   Dim delEntries As ArrayList
   While i <= MyBase.Count - 2
      Dim j As Integer = i + 1
      While j <= MyBase.count - 1
         If MyBase.Item(i).ToString = MyBase.item(j).ToString Then
            MyBase.RemoveAt(j)
         End If
         j = j + 1
      End While
      i = i + 1
   End While
End Sub
```

The code compares each item with all following items and removes any duplicates. The duplicate items are the ones whose `ToString` property returns the same value. You might wish to perform specific comparisons, but the `ToString` method will do for our demo. Notice that the code accesses the members of the ArrayList class through the `MyBase` keyword. `MyBase` is a keyword that represents the base class, from which the custom class inherits. To test the derived class, place a button on the test form and insert the code presented by Listing 11.2 in its `Click` event handler.

LISTING 11.2: Testing the *EliminateDuplicates* Method

```
Private Sub bttnTest_Click(...) Handles bttnTest.Click
   Dim mlist As New myArrayList()
   mlist.Add(" 10")
   mlist.Add("A")
   mlist.Add("20")
   mlist.Add("087")
   mlist.Add("c")
   mlist.Add("A")
   mlist.Add("b")
   mlist.Add("a")
   mlist.Add("A")
   mlist.Add("87")
   mlist.Add(10)
```

```
        mlist.Add(100)
        mlist.Add(110)
        mlist.Add("1001")
        Console.WriteLine(mlist.GetString())
        mlist.EliminateDuplicates()
        Console.WriteLine(mlist.GetString())
    End Sub
```

Table 11.1 shows the contents of the ArrayList before and after the elimination of the duplicates. Notice that the second list contains the item 10 twice. One of the items is a string, and the other one is a numeric value; therefore, they're not duplicates.

TABLE 11.1: The *mList* ArrayList before and after the Elimination of Duplicates

ORIGINAL LIST	AFTER ELIMINATION OF DUPLICATES
10	10
A	A
20	20
087	087
C	C
A	B
B	A
A	87
A	10
87	100
10	110
100	1001
110	
1001	

`GetString` (see Listing 11.3) is not a method of the ArrayList; it's a method of the extended ArrayList class, which returns the values of all the items in the list. (It uses each item's `ToString` method to retrieve the string representation of the individual items and concatenates them with a line feed separator.)

LISTING 11.3: *GetString* Method

```
Function GetString() As String
   Dim i As Integer
   Dim strValue As String
   strValue = MyBase.Item(0).ToString
   For i = 1 To MyBase.Count - 1
      strValue = strValue & vbCrLf & MyBase.Item(i).ToString
   Next
   GetString = strValue
End Function
```

Another problem with the ArrayList class is that it can't sort its elements if they're not of the same type. You can always provide a custom comparer for custom types, but it's impossible to write a comparer that can handle all objects. Sometimes, however, we need to know the smallest or largest numeric element, or the alphabetically first or last element. These methods apply to numeric or string elements only; if some of the collection's elements are objects, we can ignore them. Let's implement two more custom methods for the myArrayList class (see Listing 11.4). The `Min` method returns the alphabetically smallest value; the `NumMin` method returns the numerically smallest value.

LISTING 11.4: *Min* and *NumMin* Methods of the ArrayList Class

```
Function Min() As String
   Dim i As Integer
   Dim minValue As String
      minValue = MyBase.Item(0).ToString
      For i = 1 To MyBase.Count - 1
         If MyBase.Item(i).ToString < minValue Then _
            minValue = MyBase.Item(i).ToString
      Next
      Min = minValue
End Function

Function NumMin() As Double
   Dim i As Integer
   Dim minValue As Double
      minValue = 1E+230
      For i = 1 To MyBase.Count - 1
         If IsNumeric(MyBase.item(i)) And _
            val(MyBase.Item(i).tostring) < minValue Then _
               minValue = val(MyBase.Item(i).tostring)
      Next
      NumMin = minValue
End Function
```

You can populate the `myArrayList` collection with strings and integers and call the `Min` and `NumMin` methods to retrieve the smaller string or numeric value in the list.

What have we done in this section, really? We took an existing class, a powerful one, and extended it. We did that by writing simple procedures that could have appeared in any application. We just inserted the `Inherits` keyword followed by the name of an existing class on which we want to base our class, and provided the implementation of the new methods. A few more keywords to learn, and you can practically customize any class that comes with the Framework. Existing applications won't break (the ArrayList class is actually used by some system services, which will keep working fine); they see the original class, not the customized class. Some of your new applications will see the enhanced ArrayList. Another developer might further extend the functionality of your derived class. The old applications will work because ArrayList is still around, your applications will also work because myArrayList hasn't been modified, and someone else's applications will work with another class derived from yours.

This type of inheritance, in which we inherit an existing class and add new members and/or revise existing ones, is called *implementation inheritance*. Implementation inheritance is a powerful feature and can be used in many situations, besides enhancing an existing class. You can design base classes that address a large category of objects and then subclass them for specific objects. The typical example is the Person class, from which classes such as Contact, Customer, Employee, and so on can be derived. Inheritance is used with large-scale projects to ensure consistent behavior across the application. Later in this chapter, you'll see an interesting application of inheritance. We'll build classes that describe related objects (shapes), all of which will be based on a single class that encapsulates the basic characteristics of all derived classes.

Inheriting Custom Classes

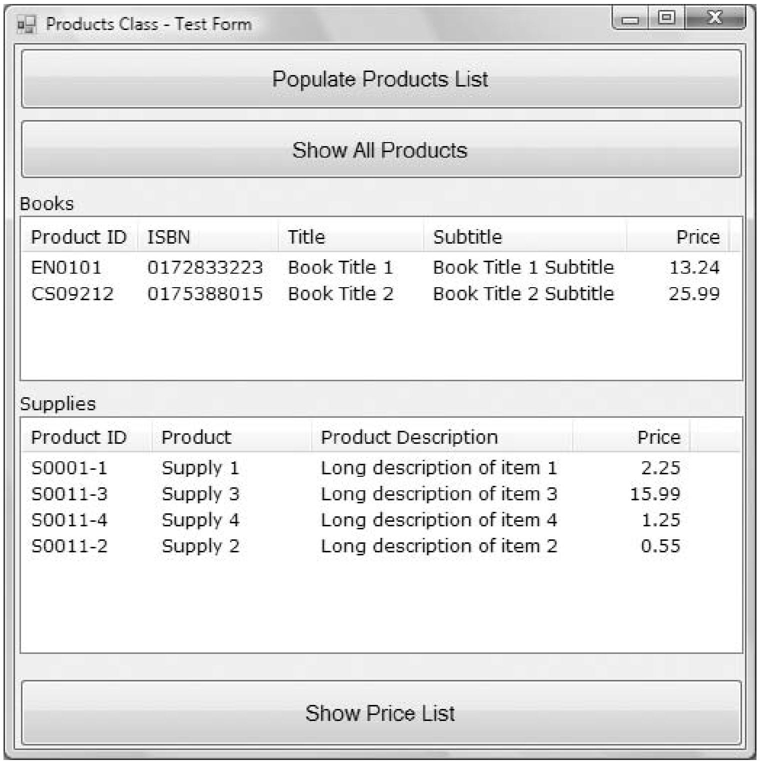
In this example, we'll tackle a very real problem by using inheritance. Consider a structure for storing product information; in most applications, this structure is optimized for a specific product type. In my consulting days, I've seen designs that try to capture the "global" product: a structure that can store products of any type. This approach leads to unnecessarily large database tables, name conflicts, and all kinds of problems that surface after the program has been installed at customers with different product types. Here's my suggestion for handling multiple types of products.

Every company makes money by selling products and services, and every company has different requirements. Even two bookstores don't store the same information in their databases. However, there are a few pieces of information that any company uses to sell its products: the product's code, its description, and its price. This is the minimum information you need to sell something (it's the information that's actually printed in the invoice). The price is usually stored to a different table, along with the company's pricing policies. Without being too specific, these are the three pieces of information for ordering and selling products. We use these items to maintain a list of orders and invoices, and keep track of the stock, customer balances, and so on. The specifics of a product can be stored to different tables in the database, and these tables will be implemented upon request. If your customer is a book seller, you'll design tables for storing data such as publisher and author names, book descriptions, ISBNs, and the like.

You'll also have to write applications to maintain all this information. To sell the same application to an electronics store, you must write another module for maintaining a different type of product, but the table with the basic data remains the same. Clearly, you can't design a program for handling all types of products, nor can you edit the same application to fit different products. You just have to write different applications for different types of products, but the parts of the application that deal with buying and selling products, customers, suppliers, and other peripheral entities won't change.

Let's look at a custom class for storing products, which is part of the Products sample project. The application's main form is shown in Figure 11.1.

FIGURE 11.1
Exercising the Book and Supply inherited classes



Books

Product ID	ISBN	Title	Subtitle	Price
EN0101	0172833223	Book Title 1	Book Title 1 Subtitle	13.24
CS09212	0175388015	Book Title 2	Book Title 2 Subtitle	25.99

Supplies

Product ID	Product	Product Description	Price
S0001-1	Supply 1	Long description of item 1	2.25
S0011-3	Supply 3	Long description of item 3	15.99
S0011-4	Supply 4	Long description of item 4	1.25
S0011-2	Supply 2	Long description of item 2	0.55

The most basic class stores the information we'll need in our ordering and invoicing applications: the product's ID, its name, and its price. Here's the implementation of a simple Product class:

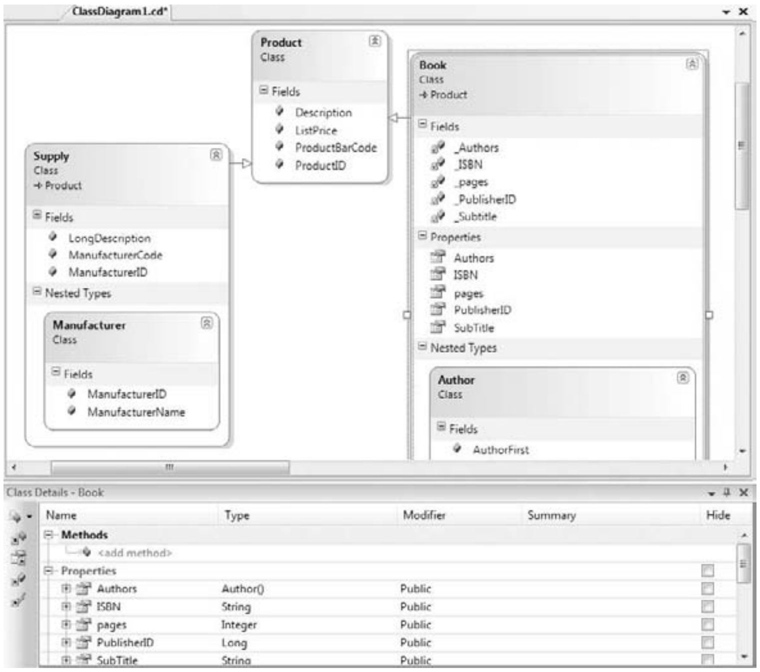
```
Public Class Product
    Public Description As String
    Public ProductID As String
    Public ProductBarCode As String
    Public ListPrice As Decimal
End Class
```

I included the product's bar code because this is how products are usually sold at cash registers. This class can represent any product for the purposes of buying and selling it. Populate a collection with objects of this type and you're ready to write a functional interface for creating invoices and purchase orders.

Now we'll take into consideration the various types of products. To keep the example simple, consider a store that sells books and supplies. Each type of product is implemented with a different class, which inherits from the Product class. Supplies don't have ISBNs, and books don't have manufacturers — they have authors and publishers; don't try to fit everything into a single object, or (even worse) into a single database table.

Figure 11.2 shows the base class, Product, and the two derived classes, Supply and Book, in the Class Diagram Designer. The arrows (if they exist) point to the base class of a derived class, and nested classes (such as the Author and Publisher classes) are contained in the box of their parent class.

FIGURE 11.2
Viewing a hierarchy of classes with the Class Diagram Designer



Name	Type	Modifier	Summary	Hide
Methods				
<add method>				
Properties				☐
Authors	Author()	Public		☐
ISBN	String	Public		☐
pages	Integer	Public		☐
PublisherID	Long	Public		☐
SubTitle	String	Public		☐

Listing 11.5 is a simple class for representing books, the Book class.

LISTING 11.5: Simple Class for Representing Books

```
Public Class Book
    Inherits Product
    Public Subtitle As String
    Public ISBN As String
    Public pages As Integer
    Public PublisherID As Long
    Public Authors() As Author

    Public Class Author
        Public AuthorID As Long
        Public AuthorLast As String
        Public AuthorFirst As String
    End Class
```

```
        Public Class Publisher
            Public PublisherID As Long
            Public PublisherName As String
            Public PublisherPhone As String
        End Class
    End Class
```

In addition to its own properties, the Book class exposes the properties of the Product class as well. Because the book industry has a universal coding scheme (the ISBN), the product's code is the same as its ISBN. This, however, is not a requirement of the application. You will probably add some extra statements to make sure that the *`ProductID`* field of the Product class and the *`ISBN`* field of the Book class always have the same value.

The class that represents supplies is shown in Listing 11.6.

Listing 11.6: Simple Class for Representing Supplies

```
    Public Class Supply
        Inherits Product
        Public LongDescription As String
        Public ManufacturerCode As String
        Public ManufacturerID As Long

        Public Class Manufacturer
            Public ManufacturerID As Long
            Public ManufacturerName As String
        End Class
    End Class
```

To make sure that this class can accommodate all pricing policies for a company, you can implement a `GetPrice` method, which returns the product's sale price (which can be different at different outlets or for different customers). The idea is that some piece of code accepts the product's list (or purchase) price and the ID of the customer who buys it. This code can perform all kinds of calculations, look up tables in the database, or perform any other action, and return the product's sale price: the price that will appear on the customer's receipt. We'll keep our example simple and sell with the list price. To implement any other pricing policy, I recommend implementing a procedure that accepts as arguments the ID of the product being sold and the ID of the buyer, and returns a price. This procedure can be a class method or even a stored procedure in the database.

Let's write some code to populate a few instances of the Book and Supply classes. The following statements populate a HashTable with books and supplies. The HashTable is a structure for storing objects along with their keys. In this case, the keys are the IDs of the products. The HashTable can locate items by means of their keys very quickly, and this is why I chose this type of collection to store the data. HashTables, as well as other collections, are discussed in detail in Chapter 14.

```
    Dim P1 As New Book
    P1.ListPrice = 13.24D
```

```
P1.Description = "Book Title 1"
P1.ProductID = "EN0101"
P1.ISBN = "0172833223"
P1.Subtitle = "Book Title 1 Subtitle"
Products.Add(P1.ProductID, P1)

Dim P2 As New Supply
P2.Description = "Supply 1"
P2.ListPrice = 2.25D
P2.LongDescription = "Long description of item 1"
P2.ProductID = "S0001-1"
Products.Add(P2.ProductID, P2)
```

`Products` is the name of the collection in which the products are stored, and is declared as follows:

```
Dim Products As New Hashtable
```

Each item in the *`Products`* collection is either of the Book or of the Supply type, and you can find out its type with the following expression:

```
If TypeOf Products.Item(key) Is Book ...
```

Listing 11.7 shows the code behind the Display Products button on the sample application's form. The code iterates through the items of the collection, determines the type of each item, and adds the product's fields to the appropriate ListView control.

Listing 11.7: Iterating through a Collection of Book and Supply Products

```
Private Sub Button2_Click(...) Handles bttnDisplay.Click
    Dim key As String
    Dim LI As ListViewItem
    For Each key In Products.Keys
        LI = New ListViewItem
        Dim bookItem As Book, supplyItem As Supply
        If TypeOf Products.Item(key) Is Book Then
            bookItem = CType(Products.Item(key), Book)
            LI.Text = bookItem.ISBN
            LI.SubItems.Add(bookItem.Description)
            LI.SubItems.Add("")
            LI.SubItems.Add( bookItem.ListPrice.ToString("#,##0.00"))
            ListView1.Items.Add(LI)
        End If
        If TypeOf Products.Item(key) Is Supply Then
            supplyItem = CType(Products.Item(key), Supply)
            LI.Text = supplyItem.ProductID
            LI.SubItems.Add(supplyItem.Description)
```

```
                LI.SubItems.Add(supplyItem.LongDescription)
                LI.SubItems.Add( supplyItem.ListPrice.ToString("#,##0.00"))
                ListView2.Items.Add(LI)
            End If
        Next
    End Sub
```

It's fairly easy to take advantage of inheritance in your projects. The base class encapsulates the functionality that's necessary for multiple classes. All other classes inherit from the base class and add specific members that don't apply to other classes (at least, not all of them). The actual data resides in a database, and you'll have to write code that populates the `Products` collection from the database, but this is a topic we'll discuss in Chapter 21, ''Basic Concepts of Relational Databases.''

As I mentioned earlier, for the purposes of selling products, you can use the Product class. You can search for both books and supplier with their ID or bar code and use the product's description and price to generate an invoice.

The following statements retrieve a product by its ID and print its description and price:

```
Dim id As String
id = InputBox("ID")
If Products.Contains(id) Then
    Dim selProduct As Product
    selProduct = CType(Products(id), Product)
    Debug.WriteLine("The price of " & selProduct.Description & _
                    " is" & selProduct.ListPrice)
End If
```

If executed, the preceding statements will print the following in the Output window. This is all the information you need to prepare invoices and orders, and it comes from the Product class, which is the base class for all products.

```
The price of Supply 2 is 5.99
```

Before ending this section, I should point out that you can convert the type of an inherited class only to that of the parent class. You can convert instances of the Book and Supply class to objects of the Product type, but not the opposite. The only valid type conversion is a widening conversion (from a narrower to a wider type).

You won't be hard-pressed to come up with real-world situations that call for inheritance. Employees, customers, and suppliers can all inherit from the Person class. Checking and savings accounts can inherit from the Account class, which stores basic information such as customer info and balances. Later in this chapter, you'll develop a class that represents shapes and you'll use it as a basis for classes that implement specific shapes such as circles, rectangles, and so on.

Polymorphism

A consequence of inheritance is another powerful OOP technique: *polymorphism*, which is the capability of a base type to adjust itself to accommodate many different derived types. Let's make it simpler by using some analogies in the English language. Take the word *run*, for example.

This verb can be used to describe what athletes, cars, or refrigerators do; they all run. In different sentences, the same word takes on different meanings. When you use it with a person, it means going a distance at a fast pace. When you use it with a refrigerator, it means that it's working. When you use it with a car, it may take on both meanings. So, in a sense the word *run* is polymorphic (and so are many other English words): Its exact meaning is differentiated by the context.

To apply the same analogy to computers, think of a class that describes a basic object such as a Shape. This class would be very complicated if it had to describe and handle all shapes. It would be incomplete, too, because the moment you released it to the world, you'd come up with a new shape that can't be described by your class. To design a class that describes all shapes, you build a simple class to describe shapes at large, and then you build a separate class for each individual shape: a Triangle class, a Square class, a Circle class, and so on. As you can guess, all these classes inherit the Shape class. Let's also assume that all the classes that describe individual shapes have an `Area` method, which calculates the area of the shape they describe. The name of the `Area` method is the same for all classes, but it calculates a different formula for different shapes.

Developers, however, shouldn't have to learn a different syntax of the `Area` method for each shape; they can declare a Square object and calculate its area with the following statements:

```
Dim shape1 As New Square(5)' statements to initialize the square
Dim area As Double = shape1.Area
```

If *`shape2`* represents a circle, the same method will calculate the circle's area. (I'm assuming that the constructors accept as an argument the square's side and the circle's radius, respectively.)

```
Dim shape2 As New Circle(9.90)(
Dim area As Double = shape2.Area
```

You can go through a list of objects derived from the Shape class and calculate their areas by calling the `Area` method. No need to know what shape each object represents — you just call its `Area` method. Let's say you created an ArrayList with various shapes. You can go through the collection and calculate the total area with a loop like the following:

```
Dim shapeEnum As IEnumerator
Dim totalArea As Double = 0.0
shapeEnum = aList.GetEnumerator
While shapeEnum.MoveNext
   totalArea = totalArea + CType(shapeEnum.Current, Shape).Area
End While
```

The `CType()` function converts the current element of the collection to a Shape object; it's necessary only if the Strict option is on, which prohibits VB from late-binding the expression. (Strict is off by default.)

One rather obvious alternative is to build a separate function to calculate the area of each shape (`SquareArea`, `CircleArea`, and so on). It will work, but why bother with so many function names, not to mention the overhead in your code? You must first figure out the type of shape described by a specific variable, such as *`shape1`*, and then call the appropriate method. The code will not be as easy to read, and the longer the application gets, the more `If` and `Case` statements you'll be coding. Not to mention that each method would require different arguments for its calculations.

This approach clearly offsets the benefits of object-oriented programming by reducing classes to collections of functions.

The second, even less-efficient method is a really long `Area()` function that would be able to calculate the area of all shapes. This function should be a very long `Case` statement, such as the following one:

```
Public Function Area(ByVal shapeType As String) As Double
   Select Case shapeType
      Case "Square": { calculate the area of a square }
      Case "Circle": { calculate the area of a circle }
      { . . . more Case statements }
   End Select
End Function
```

The real problem with this approach is that every time you want to add a new segment to calculate the area of a new shape to the function, you'd have to edit it. If other developers wanted to add a shape, they'd be out of luck.

In the following section, we'll build the Shape class, which we'll extend with individual classes for various shapes. You'll be able to add your own classes to implement additional shapes, and any code written using the older versions of the Shape class will keep working.

Building the Shape Class

In this section, you'll build a few classes to represent shapes to demonstrate the advantages of implementing polymorphism. Let's start with the Shape class, which will be the base class for all other shapes. This is a really simple class that's pretty useless on its own. Its real use is to expose two methods that can be inherited: `Area` and `Perimeter`. Even the two methods don't do much — actually, they do absolutely nothing. All they really do is provide a naming convention. All classes that will inherit the Shape class will have an `Area` and a `Perimeter` method, and they must provide the implementation of these methods.

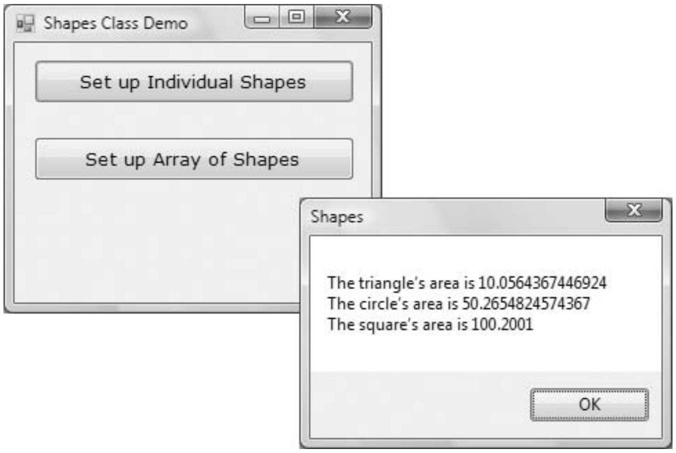
The code shown in Listing 11.8 comes from the Shapes sample project. The application's main form, which exercises the Shape class and its derived classes, is shown in Figure 11.3.

Listing 11.8: Shape Class

```
Class Shape
   Overridable Function Area() As Double
   End Function
   Overridable Function Perimeter() As Double
   End Function
End Class
```

If there are properties common to all shapes, you place the appropriate Property procedures in the Shape class. If you want to assign a color to your shapes, for instance, insert a `Color` property in this class. The `Overridable` keyword means that a class that inherits from the Shape class can override the default implementation of the corresponding methods or properties. As you will see shortly, it is possible for the base class to provide a few members that can't be overridden in the derived class. The methods that are declared but not implemented in the parent class are called *virtual methods*, or *pure virtual methods*.

FIGURE 11.3
The main form of the Shapes project



```
The triangle's area is 10.0564367446924
The circle's area is 50.2654824574367
The square's area is 100.2001
```

Next you must implement the classes for the individual shapes. Add another Class module to the project, name it **Shapes**, and enter the code shown in Listing 11.9.

LISTING 11.9: Square, Triangle, and Circle Classes

```
Public Class Square
   Inherits Shape
   Private sSide As Double
   Public Property Side() As Double
      Get
         Return(sSide)
      End Get
      Set
         sSide = Value
      End Set
   End Property

   Public Overrides Function Area() As Double
      Area = sSide * sSide
   End Function

   Public Overrides Function Perimeter() As Double
      Return (4 * sSide)
   End Function
End Class

Public Class Triangle
   Inherits Shape
   Private side1, side2, side3 As Double
   Property SideA() As Double
      Get
         Return(Side1)
      End Get
      Set
         side1 = Value
      End Set
   End Property

   Property SideB() As Double
      Get
         Return(side2)
      End Get
      Set
         side2 = Value
      End Set
   End Property

   Public Property SideC() As Double
      Get
         Return(side3)
      End Get
      Set
         side3 = Value
      End Set
   End Property

   Public Overrides Function Area() As Double
      Dim perim As Double
      perim = Perimeter()
      Return (Math.Sqrt(perim * (perim - side1) * _
             (perim - side2) * (perim - side3)))
   End Function

   Public Overrides Function Perimeter() As Double
      Return (side1 + side2 + side3)
   End Function
End Class

Public Class Circle
   Inherits Shape
   Private cRadius As Double
   Public Property Radius() As Double
      Get
         Return(cRadius)
      End Get
      Set
         cRadius = Value
      End Set
   End Property

   Public Overrides Function Area() As Double
      Return (Math.Pi * cRadius ˆ 2)
   End Function

   Public Overrides Function Perimeter() As Double
      Return (2 * Math.Pi * cRadius)
   End Function
End Class
```

The Shapes.vb file contains three classes: the Square, Triangle, and Circle classes. All three expose their basic geometric characteristics as properties. The Triangle class, for example, exposes the properties SideA, SideB, and SideC, which allow you to set the three sides of the triangle. In a real-world application, you may opt to insert some validation code, because not any three sides produce a triangle. You must also insert parameterized constructors for each shape. The implementation of these constructors is trivial, and I'm not showing it in the listing; you'll find the appropriate constructors if you open the project with Visual Studio. The Area and Perimeter methods are implemented differently for each class, but they do the same thing: They return the area and the perimeter of the corresponding shape. The Area method of the Triangle class is a bit involved, but it's just a formula (the famous Heron's formula for calculating a triangle's area).

Testing the Shape Class

To test the Shape class, all you have to do is create three variables — one for each specific shape — and call their methods. Or, you can store all three variables into an array and iterate through them. If the collection contains Shape variables only, the current item is always a shape, and as such it exposes the Area and Perimeter methods. The code in Listing 11.10 does exactly that. First, it declares three variables of the Triangle, Circle, and Square types. Then it sets their properties and calls their Area method to print their areas.

Listing 11.10: Testing the Shape Class

```
   Protected Sub bttnAreas_Click(...) Handles bttnAreas.Click
      Dim shape1 As New Triangle()
      Dim shape2 As New Circle()
      Dim shape3 As New Square()
   ' Set up a triangle
      shape1.SideA = 3
      shape1.SideB = 3.2
      shape1.SideC = 0.94
      Console.WriteLine("The triangle's area is " & shape1.Area.ToString)
   ' Set up a circle
      shape2.Radius = 4
      Console.WriteLine("The circle's area is " & shape2.Area.ToString)
   ' Set up a square
```

```
        shape3.Side = 10.01
        Console.WriteLine("The square's area is " & shape3.Area.ToString)
        Dim shapes() As Shape
        shapes(0) = shape1
        shapes(1) = shape2
        shapes(2) = shape3
        Dim shapeEnum As IEnumerator
        Dim totalArea As Double
        shapeEnum = shapes.GetEnumerator
        While shapeEnum.MoveNext
           totalArea = totalArea + CType(shapeEnum.Current, shape).Area
        End While
        Console.WriteLine("The total area of your shapes is " & _
                          totalArea.ToString)
    End Sub
```

In the last section, the test code stores all three variables into an array and iterates through its elements. At each iteration, it casts the current item to the Shape type and calls its `Area` method. The expression that calculates areas is `CType(shapeEnum.Current, shape).Area`, and the same expression calculates the area of any shape.

Casting Objects to Their Parent Type

The trick that makes polymorphism work is that objects of a derived type can be cast to their parent type. An object of the Circle type can be cast to the Shape type, because the Shape type contains less information than the Circle type. You can cast objects of a derived type to their parent type, but the opposite isn't true. The methods that are shared among multiple derived classes should be declared in the parent class, even if they contain no actual code. Just don't forget to prefix them with the `Overridable` keyword. There's another related attribute, the `MustOverride` attribute, which forces every derived class to provide its own implementation of a method or property.

Depending on how you will use the individual shapes in your application, you can add properties and methods to the base class. In a drawing application, all shapes have an outline and a fill color. These properties can be implemented in the Shape class because they apply to all derived classes. Any methods with a common implementation for all classes should also be implemented as methods of the parent class. Methods that are specific to a shape must be implemented in one of the derived classes.

Who Can Inherit What?

The Shape base class and the Shapes derived class work fine, but there's a potential problem. A new derived class that implements a new shape may not override the `Area` or the `Perimeter` method. If you want to force all derived classes to implement a specific method, you can specify the `MustInherit` modifier for the class declaration and the `MustOverride` modifier for the member declaration. If some of the derived classes may not provide their implementation of a method, this method of the derived class must also be declared with the `Overridable` keyword.

The Shapes project uses the `MustInherit` keyword in the definition of the Shape class. This keyword tells the CLR that the Shape class can't be used as is; it must be inherited by another class. A class that can't be used as is, is known as an *abstract base class*, or a *virtual class*. The definition of the `Area` and `Perimeter` methods are prefixed with the `MustOverride` keyword, which tells the compiler that derived classes (the ones that will inherit the members of the base class) must provide their own implementation of the two methods:

```
Public MustInherit Class Shape
   Public MustOverride Function Area() As Double
   Public MustOverride Function Perimeter() As Double
End Class
```

Notice that there's no `End Function` statement, just the declaration of the function that must be inherited by all derived classes. If the derived classes may override one or more methods optionally, these methods must be implemented as actual functions. Methods that *must* be overridden need not be implemented as functions — they're just placeholders for a name. You must also specify their parameters, if any. The definitions of the methods you specify are known as the methods' *signature*.

There are other modifiers you can use with your classes, such as the `NotInheritable` modifier, which prevents your class from being used as a base class by other developers. The System.Array class is an example of a Framework class can't be inherited.

In the following section, you'll look at the class-related modifiers and learn when to use them. The various modifiers are keywords, such as the `Public` and `Private` keywords that you can use in variable declarations. These keywords can be grouped according to the entity they apply to, and I used this grouping to organize them in the following sections.

Parent Class Keywords

These keywords apply to classes that can be inherited, and they appear in front of the `Class` keyword. By default, all classes can be inherited, but their members can't be overridden. You can change this default behavior with the following modifiers:

NotInheritable This prevents the class from being inherited. The base data types, for example, are not inheritable. In other words, you can't create a new class based on the Integer data type. The Array class is also not inheritable.

MustInherit This class must be inherited. Classes prefixed with the `MustInherit` attribute are called abstract classes, and the Framework contains quite a few of them. You can't create an object of this class in your code and, therefore, you can't access its methods. The Shape class is nothing more than a blueprint for the methods it exposes and can't be used on its own; that's why it was declared with the `MustInherit` keyword.

Derived Class Keywords

The following keywords may appear in a derived class; they have to do with the derived class's parent class:

Inherits Any derived class must inherit an existing class. The `Inherits` statement tells the compiler which class it derives from. A class that doesn't include the `Inherits` keyword is by definition a base class.

MyBase Use the `MyBase` keyword to access a derived class's parent class from within the derived class's code.

Parent Class Member Keywords

These keywords apply to the members of classes that can be inherited, and they appear in front of the member's name. They determine how derived classes must handle the members (that is, whether they can or must override their properties and methods):

Overridable Every member with this modifier can be overwritten. If a member is declared as `Public` only, it can't be overridden. You should allow developers to override as many of the members of your class as possible, as long as you don't think there's a chance that they might break the code by overriding a member. Members declared with the `Overridable` keyword don't necessarily need to be overridden, so they must provide some functionality.

NotOverridable Every member declared with this modifier can't be overridden in the inheriting class.

MustOverride Every member declared with this modifier must be overridden. You can skip the overriding of a member declared with the `MustOverride` modifier in the derived class, as long as the derived class is declared with the `MustInherit` modifier. This means that the derived class must be inherited by some other class, which then receives the obligation to override the original member declared as `MustOverride`.

The two methods of the Shape class must be overridden, and we've done so in all the derived classes that implement various shapes. Let's also assume that you want to create different types of triangles with different classes (an orthogonal triangle, an isosceles triangle, and a generic triangle). Let's also assume that these classes would inherit the Triangle class. You can skip the definition of the `Area` method in the Triangle class, but you'd have to include it in the derived classes that implement the various types of triangles. Moreover, the Triangle class would have to be marked as `MustInherit`.

Public This modifier tells the CLR that the specific member can be accessed from any application that uses the class. This, as well as the following keywords, are access modifiers and are strictly inheritance related, but I'm listing them here for completeness.

Private This modifier tells the CLR that the specific member can be accessed only in the module in which it was declared. All the local variables must be declared as `Private`, and no other class (including derived classes) or application will see them.

Protected `Protected` members have scope between public and private, and they can be accessed in the derived class, but they're not exposed to applications using either the parent class or the derived classes. In the derived class, they have a private scope. Use the `Protected` keyword to mark the members that are of interest to developers who will use your class as a base class, but not to developers who will use it in their applications.

Protected Friend This modifier tells the CLR that the member is available to the class that inherits the class, as well as to any other component of the same project.

Derived Class Member Keyword

The `Overrides` keyword applies to members of derived classes and indicates whether a member of the derived class overrides a base class member. Use this keyword to specify the member of the parent class you're overriding. If a member has the same name in the derived class as in the parent class, this member must be overridden. You can't use the `Overrides` keyword with members that were declared with the `NotOverridable` or `Protected` keywords in the base class.

VB 2008 At Work: The InheritanceKeywords Project

A few examples are in order. The sample application of this section is the InheritanceKeywords project, and it contains a few classes and a simple test form. Create a simple class by entering the statements of Listing 11.11 in a Class module, and name the module **ParentClass**.

LISTING 11.11: InheritanceKeywords Class

```
Public MustInherit Class ParentClass
   Public Overridable Function Method1() As String
      Return ("I'm the original Method1")
   End Function
   Protected Function Method2() As String
      Return ("I'm the original Method2")
   End Function
   Public Function Method3() As String
      Return ("I'm the original Method3")
   End Function
   Public MustOverride Function Method4() As String
' No code in a member that must be overridden !
' Notice the lack of the matching End Function here
   Public Function Method5() As String
      Return ("I'm the original Method5")
   End Function
   Private prop1, prop2 As String
   Property Property1() As String
      Get
         Property1 = "Original Property1"
      End Get
      Set
         prop1 = Value
      End Set
   End Property
   Property Property2() As String
      Get
         Property2 = "Original Property2"
      End Get
      Set
         prop2 = Value
      End Set
   End Property
End Class
```

This class has five methods and two properties. Notice that `Method4` is declared with the `MustOverride` keyword, which means it must be overridden in a derived class. Notice also the structure of Method4. It has no code, and the End Function statement is missing. Method4 is declared with the MustOverride keyword, so you can't instantiate an object of the ParentClass type. A class that contains even a single member marked as MustOverride must also be declared as MustInherit.

Place a button on the class's test form, and in its code window attempt to declare a variable of the ParentClass type. VB will issue a warning that you can't create a new instance of a class declared with the MustInherit keyword. Because of the MustInherit keyword, you must create a derived class. Enter the lines from Listing 11.12 in the ParentClass module after the end of the existing class.

Listing 11.12: Derived Class

```
Public Class DerivedClass
   Inherits ParentClass
   Overrides Function Method4() As String
      Return ("I'm the derived Method4")
   End Function
   Public Function newMethod() As String
      Console.WriteLine("<This is the derived Class's newMethod " & _
                        "calling Method2 of the parent Class> ")
      Console.WriteLine("    " & MyBase.Method2())
   End Function
End Class
```

The Inherits keyword determines the parent class. This class overrides the Method4 member and adds a new method to the derived class: newMethod. If you switch to the test form's code window, you can now declare a variable of the DerivedClass type:

```
Dim obj As DerivedClass
```

This class exposes all the members of ParentClass except for the Method2 method, which is declared with the Protected modifier. Notice that the newMethod() function calls this method through the MyBase keyword and makes its functionality available to the application. Normally, we don't expose Protected methods and properties through the derived class.

Let's remove the MustInherit keyword from the declaration of the ParentClass class. Because it's no longer mandatory that the ParentClass be inherited, the MustInherit keyword is no longer a valid modifier for the class' members. So, Method4 must be either removed or implemented. Let's delete the declaration of the Method4 member. Because Method4 is no longer a member of the ParentClass, you must also remove the entry in the DerivedClass that overrides it.

MyBase and MyClass

The MyBase and MyClass keywords let you access the members of the base class and the derived class explicitly. To see why they're useful, edit the ParentClass, as shown here:

```
Public Class ParentClass
   Public Overridable Function Method1() As String
      Return (Method4())
   End Function
   Public Overridable Function Method4() As String
      Return ("I'm the original Method4")
   End Function
```

Override `Method4` in the derived class, as shown here:

```
Public Class DerivedClass
   Inherits ParentClass
   Overrides Function Method4() As String
   Return("Derived Method4")
End Function
```

Switch to the test form, add a button, declare a variable of the derived class, and call its `Method4`:

```
Dim objDerived As New DerivedClass()
Debug.WriteLine(objDerived.Method4)
```

What will you see if you execute these statements? Obviously, the string `Derived Method4`. So far, all looks reasonable, and the class behaves intuitively. But what if we add the following method in the derived class?

```
Public Function newMethod() As String
   Return (Method1())
End Function
```

This method calls `Method1` in the ParentClass class because `Method1` is not overridden in the derived class. `Method1` in the base class calls `Method4`. But which `Method4` gets invoked? Surprised? It's the derived `Method4`! To fix this behavior (assuming you want to call the `Method4` of the base class), change the implementation of `Method1` to the following:

```
Public Overridable Function Method1() As String
   Return (MyClass.Method4())
End Function
```

If you run the application again, the statement

```
Console.WriteLine(objDerived.newMethod)
```

will print this string:

```
I'm the original Method4
```

Is it reasonable for a method of the base class to call the overridden method? It is reasonable because the overridden class is newer than the base class, and the compiler tries to use the newest members. If you had other classes inheriting from the DerivedClass class, their members would take precedence.

Use the `MyClass` keyword to make sure that you're calling a member in the same class, and not an overriding member in an inheriting class. Likewise, you can use the keyword `MyBase` to call the implementation of a member in the base class, rather than the equivalent member in a derived class. `MyClass` is similar to `MyBase`, but it treats the members of the parent class as if they were declared with the `NotOverridable` keyword.

The Class Diagram Designer

Classes are quite simple to build and use, and so is OOP. There are even tools to help you design and build your classes, which I'll describe briefly here. You can use the Class Diagram Designer to build your classes with point-and-click operations, but you can't go far on this tool alone. The idea is that you specify the name and the type of a property, and the tool emits the `Get` and `Set` procedures for the property (the getters and setters, as they're known in OOP jargon). The default implementation of *setters* and *getters* is trivial, and you'll have to add your own validation code. You can also create new methods by specifying their names and arguments, but the designer won't generate any code for you; you must implement the methods yourself. Tools such as the Class Diagram Designer or Visio allow you to visualize the classes that make up a large project and the relations between them, and they're a necessity in large projects. Many developers, however, build applications of substantial complexity without resorting to tools for automating the process of building classes. You're welcome to explore these tools, however.

Right-click the name of a class in Solution Explorer and choose View Class Diagram from the context menu. You'll see a diagram of the class on the design surface, showing all the members of the class. You can add new members, select the type of the properties, and edit existing members. The diagram of a trivial class like the Contact class is also trivial, but the class diagram becomes more helpful as you implement more interrelated classes.

Figure 11.2, from earlier in the chapter, shows the Product, Book, and Supply classes in the Class Diagram Designer. You can use the commands of each class's context menu to create new members and edit/remove existing ones. To add a new property, for example, you specify the property's name and type, and the designer generates the outline of the `Set` and `Get` procedures for you. Of course, you must step in and insert your custom validation code in the property's setter.

To add a new class to the diagram, right-click on the designer's surface and choose Add Class from the context menu. You'll be prompted to enter the name of the class and its location: the VB file in which the autogenerated class's code will be stored. You can specify a new name, or select the file of an existing class and add your new class to it. To create a derived class, you must double-click the box that represents the new class and manually insert the `Inherits` statement followed by the name of the base class. After you specify the parent class, a line will be added to the diagram joining the two classes. The end of the line at the parent class has an arrow. In other words, the arrow points to the parent class. In addition to classes, you can add other items, including structures, enumerations, and comments. Experiment with the tools of the Class Diagram Designer to jumpstart the process of designing classes. You can also create class diagrams from existing classes. At the very least, you should use this tool to document your classes, especially in a team environment.

To add members to a class, right-click the box that represents the class and choose Add from the context menu. This will lead to a submenu with the members you can add to a class: Method, Property, Field, and Event. You can also add a constructor (although you will have to supply the arguments and the code for parameterized constructors), a destructor, and a constant. To edit a member, such as the type of a property or the arguments of a method, switch to the Class Details window, where you will see the members of the selected class. Expand any member to see its parameters: the type of a property and the arguments and the return value of a method.

The Bottom Line

Use inheritance. Inheritance, which is the true power behind OOP, allows you to create new classes that encapsulate the functionality of existing classes without editing their code. To inherit from an existing class, use the `Inherits` statement, which brings the entire class into your class.

Master It Explain the inheritance-related attributes of a class's members.

Use polymorphism. Polymorphism is the ability to write members that are common to a number of classes but behave differently, depending on the specific class to which they apply. Polymorphism is a great way of abstracting implementation details and delegating the implementation of methods with very specific functionality to the derived classes.

Master It The parent class Person represents parties, and it exposes the `GetBalance` method, which returns the outstanding balance of a person. The Customer and Supplier derived classes implement the `GetBalance` method differently. How will you use this method to find out the balance of a customer and/or supplier?

Chapter 12

Building Custom Windows Controls

Just as you can design custom classes, you can use Visual Studio to design custom controls. The process is very similar, in the sense that custom controls have properties, methods, and events, which are implemented with code that's identical to the code you'd use to implement these members with classes. The difference is that controls have a visual interface and interact with the user. In short, you must provide the code to draw the control's surface, as well as react to selected user actions from within the control's code.

In this chapter, you'll learn how to enhance the functionality of existing controls, a common practice among developers. You've already seen in the preceding chapter how to inherit an existing class and add custom members. You can do the same with the built-in controls.

There are several methods of designing custom controls. In this chapter, you'll learn how to do the following:

- Extend the functionality of existing Windows Forms controls with inheritance
- Build compound custom controls that combine multiple existing controls
- Build custom controls from scratch
- Customize the rendering of the items in a ListBox control

On Designing Windows Controls

Before I get to the details of how to build custom controls, I want to show you how they relate to other types of projects. I'll discuss briefly the similarities and differences among Windows controls, classes, and Windows projects. This information will help you get the big picture and put together the pieces of the following sections.

A standard application consists of a main form and several (optional) auxiliary forms. The auxiliary forms support the main form because they usually accept user data that are processed by the code in the main form. You can think of a custom control as a form and think of its Properties window as the auxiliary form.

An application interacts with the user through its interface. The developer decides how the forms interact with the user, and the user has to follow these rules. Something similar happens with custom controls. The custom control provides a well-defined interface, which consists of properties and methods. This is the only way to manipulate the control. Just as users of your applications don't have access to the source code and can't modify the application, developers can't see the control's source code and must access it through the interface exposed by the control. After an instance of the custom control is placed on the form, you can manipulate it through its properties and methods, and you never get to see its code.

In preceding chapters, you learned how to implement interfaces consisting of properties and methods and how to raise events from within a class. This is how you build the interface of a custom Windows control: You implement properties with Property procedures, and you implement methods as Public procedures. Although a class can provide a few properties and any number of methods, a control must provide a large number of properties. A developer who places your custom control on a form expects to see the properties that are common to all the controls (properties to set the control's dimensions, its color, the text font, the `Index` and `Tag` properties, and so on). Fortunately, many of the standard properties are exposed automatically. The developer also expects to be able to program all the common events, such as the mouse and keyboard events, as well as some events that are unique to the custom control.

The design of a Windows control is similar to the design of a form. You place controls on a form-like object, called *UserControl*, which is the control's surface. It provides nearly all the methods of a standard form, and you can adjust its appearance with the drawing methods. In other words, you can use familiar programming techniques to draw a custom control or you can use existing controls to build a custom control.

The forms of an application are the windows you see on the desktop when the application is executed. When you design the application, you can rearrange the controls on a form and program how they react to user actions. Windows controls are also windows, only they can't exist on their own and can't be placed on the desktop. They must be placed on forms.

The major difference between forms and custom controls is that custom controls can exist in two runtime modes. When the developer places a control on a form, the control is actually running. When you set a control's property through the Properties window, something happens to the control — its appearance changes or the control rejects the changes. It means that the code of the custom control is executing, even though the project on which the control is used is in design mode. When the developer starts the application, the custom control is already running. However, the control must be able to distinguish when the project is in design or execution mode and behave accordingly. Here's the first property of the UserControl object you will be using quite frequently in your code: the `DesignMode` property. When the control is positioned on a form and used in the Designer, the `DesignMode` property is True. When the developer executes the project that contains the control, the `DesignMode` property is False.

This dual runtime mode of a Windows control is something you'll have to get used to. When you design custom controls, you must also switch between the roles of Windows control developer (the programmer who designs the control) and application developer (the programmer who uses the control).

In summary, a *custom control* is an application with a visible user interface as well as an invisible programming interface. The visible interface is what the developer sees when an instance of the control is placed on the form, which is also what the user sees on the form when the project is placed in runtime mode. The developer using the control can manipulate it through its properties and methods. The control's properties can be set at both design time and runtime, whereas methods must be called from within the code of the application that uses the control. The properties and methods constitute the control's invisible interface (or the *developer interface*, as opposed to the *user interface*). You, the control developer, will develop the visible user interface on a UserControl object, which is almost identical to the Form object; it's like designing a standard application. As far as the control's invisible interface goes, it's like designing a class.

Enhancing Existing Controls

The simplest type of custom Windows control you can build is one that enhances the functionality of an existing control. Fortunately, they're the most common types of custom controls, and many developers have their own collections of ''enhanced'' Windows controls. The Windows controls are quite functional, but you won't be hard-pressed to come up with ideas to make them better.

The TextBox control, for example, is a text editor on its own, and you have seen how easy it is to build a text editor by using the properties and methods exposed by this control. Many programmers add code to their projects to customize the appearance and the functionality of the TextBox control. Let's say you're building data-entry forms composed of many TextBox controls.
To help the user identify the current control on the form, it would be nice to change its color while it has the focus. If the current control has a different color from all others, users will quickly locate the control that has the focus.

Another feature you can add to the TextBox control is to format its contents as soon as it loses focus. Let's consider a TextBox control that must accept dollar amounts. After the user enters a numeric value, the control could automatically format the numeric value as a dollar amount and perhaps change the text's color to red for negative amounts. When the control receives the focus again, you can display the amount without any special formatting, so that users can edit it quickly. As you will see, it's not only possible but actually quite easy to build a control that incorporates all the functionality of a TextBox and some additional features that you provide through the appropriate code. You already know how to add features such as the ones described here to a TextBox from within the application's code. But what if you want to enhance multiple TextBox controls on the same form or reuse your code in multiple applications?

The best approach is to create a new Windows control with all the desired functionality and then reuse it in multiple projects. To use the proper terminology, you can create a new custom Windows control that *inherits* the functionality of the TextBox control. The *derived* control includes all the functionality of the control being inherited, plus any new features you care to add to it. This is exactly what we're going to do in this section.

Building the FocusedTextBox Control

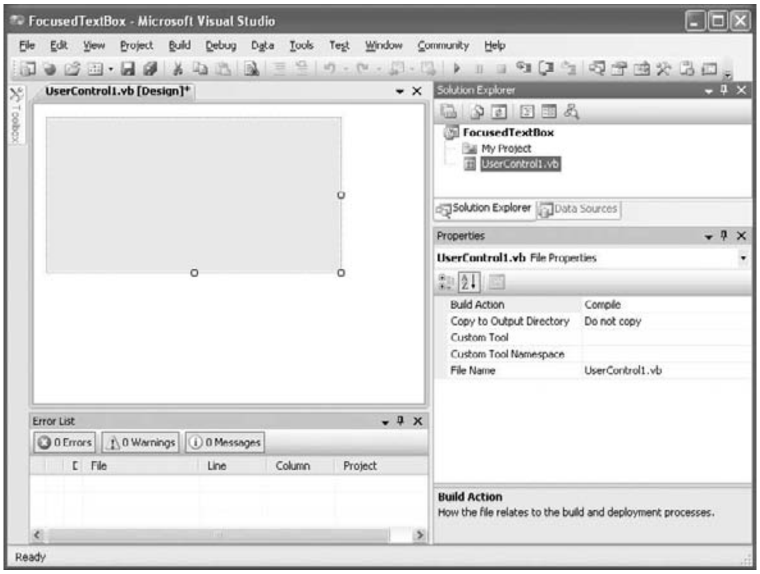
Let's call our new custom control FocusedTextBox. Start a new VB project and, in the New Project dialog box, select the template Windows Control Library. Name the project **FocusedTextBox**. The Solution Explorer for this project contains a single item, the UserControl1 item. UserControl1 (see Figure 12.1) is the control's surface — in a way, it's the control's form. This is where you'll design the visible interface of the new control using the same techniques as for designing a Windows form.

Start by renaming the UserControl1 object to **FocusedTextBox**. Then save the project by choosing File ➢ Save All. To inherit all the functionality of the TextBox control into our new control, we must insert the appropriate `Inherits` statement in the control's code. Click the Show All button in the Solution Explorer to see all the files that make up the project. Under the `FocusedTextBox.vb` file is the `FocusedTextBox.Designer.vb` file. Open this file by double-clicking its name and you'll see that it begins with the following two statements:

```
Partial Public Class FocusedTextBox
Inherits System.Windows.Forms.UserControl
```

The first statement says that the entire file belongs to the FocusedTextBox class; it's the part of the class that contains initialization code and other statements that the user does not need to see because it's left unchanged in most cases. To design an inherited control, we must change the second statement to the following:

```
Inherits System.Windows.Forms.TextBox
```


FocusedTextBox - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Test Window Community Help
Toolbox
UserControl1.vb [Design]*
Solution Explorer
FocusedTextBox
My Project
UserControl1.vb
Solution Explorer
Data Sources
Properties
UserControl1.vb File Properties
Error List
0 Errors
0 Warnings
0 Messages
File
Line
Column
Project
Ready

Build Action	Compile
Copy to Output Directory	Do not copy
Custom Tool	
Custom Tool Namespace	
File Name	UserControl1.vb

Build Action
How the file relates to the build and deployment processes.

FIGURE 12.1
A custom control in design mode

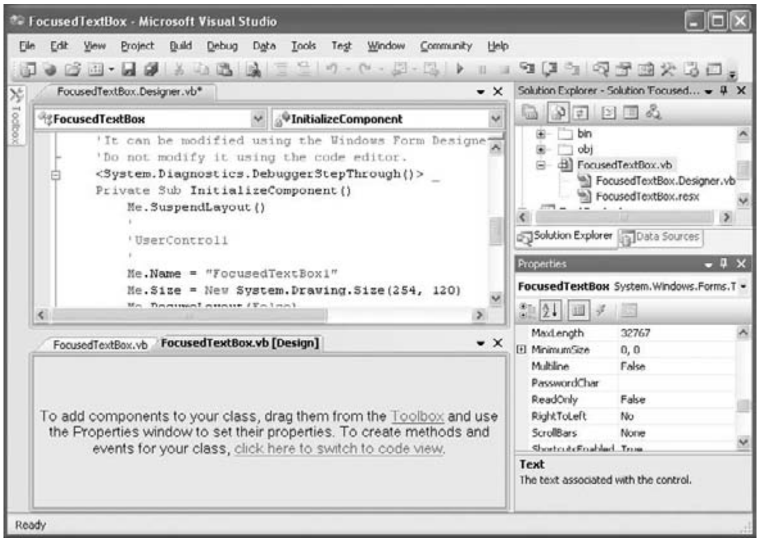
This statement tells the compiler that we want our new control to inherit all the functionality of the TextBox control. You must also modify the `InitializeComponent` method in the `FocusedTextBox.Designer.vb` file by removing the statement that sets the control's `AutoSizeMode` property. This statement applies to the generic UserControl object, but not to the TextBox control.

As soon as you specify that your custom control inherits the TextBox control, the UserControl object will disappear from the Designer. The Designer knows exactly what the new control must look like (it will look and behave exactly like a TextBox control), and you're not allowed to change it its appearance.

If you switch to the `FocusedTextBox.vb` file, you'll see that it's a public class called FocusedTextBox. The Partial class by the same name is part of this class; it contains the code that was generated automatically by Visual Studio. When compiled, both classes will produce a single DLL file. Sometimes we need to split a class's code into two files, and one of them should contain the `Partial` modifier. This keyword signifies that the file contains part of the class. The `FocusedTextBox.vb` file is where you will insert your custom code. The Partial class contains the code emitted by Visual Studio, and you're not supposed to touch it. Inherited controls are an exception to this rule, because we have to be able to modify the `Inherits` statement.

Let's test our control and verify that it exposes the same functionality as the TextBox control. Figure 12.2 shows the IDE while developing an inherited control. Notice that the FocusedTextBox control has inherited all the properties of the TextBox control, such as the `MaxLength` and `PasswordChar` properties.

To test the control, you must add it to a form. A control can't be executed outside the context of a host application. Add a new project to the solution (a Windows Application project) with the File ➢ Add ➢ New Project command. When the Add New Project dialog box appears, select the Windows Application template and set the project's name to **TestProject**. A new folder will be created under the FocusedTextBox folder — the TestProject folder — and the new project will be stored there. The TestProject must also become the solution's startup object. (This is the very reason we added the project to our solution: to have an executable for testing the custom control.) Right-click the test project's name in the Solution Explorer and select Set As StartUp Object in the context menu.


FocusedTextBox - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Test Window Community Help
FocusedTextBox.Designer.vb*
FocusedTextBox
InitializeComponent
Solution Explorer - Solution 'Focused...
bin
obj
FocusedTextBox.vb
FocusedTextBox.Designer.vb
FocusedTextBox.resx
Solution Explorer
Data Sources
Properties
FocusedTextBox System.Windows.Forms.T
FocusedTextBox.vb
FocusedTextBox.vb [Design]
Ready

```
    'It can be modified using the Windows Form Designe
    'Do not modify it using the code editor.
    <System.Diagnostics.DebuggerStepThrough()> _
    Private Sub InitializeComponent()
        Me.SuspendLayout()
        '
        'UserControl1
        '
        Me.Name = "FocusedTextBox1"
        Me.Size = New System.Drawing.Size(254, 120)
```

To add components to your class, drag them from the Toolbox and use the Properties window to set their properties. To create methods and events for your class, click here to switch to code view.

MaxLength	32767
MinimumSize	0, 0
Multiline	False
PasswordChar	
ReadOnly	False
RightToLeft	No
ScrollBars	None
ShortcutsEnabled	True

Text
The text associated with the control.

FIGURE 12.2
The IDE during the design of an inherited control

To test the control you just ''designed,'' you need to place an instance of the custom control on the form of the test project. First, you must build the control. Select the FocusedTextBox item in the Solution Explorer, and from the Build menu, select the Build FocusedTextBox command (or right-click the FocusedTextBox component in the Solution Explorer and select Build from the context menu). The build process will create a DLL file with the control's executable code in the Bin folder under the project's folder.

Then switch to the test project's main form and open the ToolBox. You will see a new tab, the FocusedTextBox Components tab, which contains all the custom components of the current project. The new control has already been integrated into the design environment, and you can use it like any of the built-in Windows controls. Every time you edit the code of the custom control, you must rebuild the control's project for the changes to take effect and update the instances of the custom control on the test form. The icon that appears before the custom control's name is the default icon for all custom Windows controls. You can associate a different icon with your custom control, as explained in the ''Classifying the Control's Properties'' section, later in this chapter.

Place an instance of the FocusedTextBox control on the form and check it out. It looks, feels, and behaves just like a regular TextBox. In fact, it is a TextBox control by a different name. It exposes all the members of the regular TextBox control: You can move it around, resize it, change its `Multiline` and `WordWrap` properties, set its `Text` property, and so on. It also exposes all the methods and events of the TextBox control.

Adding Functionality to Your Custom Control

As you can see, it's quite trivial to create a new custom control by inheriting any of the built-in Windows controls. Of course, what good is a control that's identical to an existing one? Let's add some extra functionality to our custom TextBox control. Switch to the control project and view the

FocusedTextBox object's code. In the code editor's pane, expand the Objects list and select the item FocusedTextBox Events. This list contains the events of the TextBox control because it is the base control for our custom control.

Expand the Events drop-down list and select the `Enter` event. The following event handler declaration will appear:

```
Private Sub FocusedTextBox_Enter(...) Handles Me.Enter

End Sub
```

This event takes place every time our custom control gets the focus. To change the color of the current control, insert the following statement in the event handler:

```
Me.BackColor = Color.Cyan
```

(Or use any other color you like; just make sure it mixes well with the form's default background color. You can also use the members of the `SystemColors` enumeration, to help ensure that it mixes well with the background color.) We must also program the `Leave` event, so that the control's background color is reset to white when it loses the focus. Enter the following statement in the `Leave` event's handler:

```
Private Sub FocusedTextBox_Leave(...) Handles Me.Leave
   Me.BackColor = Color.White
End Sub
```

Having a hard time picking the color that signifies that the control has the focus? Why not expose this value as a property, so that you (or other developers using your control) can set it individually in each project? Let's add the `EnterFocusColor` property, which is the control's background color when it has the focus.

Because our control is meant for data-entry operations, we can add another neat feature. Some fields on a form are usually mandatory, and some are optional. Let's add some visual indication for the mandatory fields. First, we need to specify whether a field is mandatory with the `Mandatory` property. If a field is mandatory, its background color will be set to the value of the `MandatoryColor` property, but only if the control is empty.

Here's a quick overview of the control's custom properties:

EnterFocusColor When the control receives the focus, its background color is set to this value. If you don't want the currently active control to change color, set its `EnterFocusColor` to white.

Mandatory This property indicates whether the control corresponds to a required field if Mandatory is True or to an optional field if `Mandatory` is False.

MandatoryColor This is the background color of the control if its `Mandatory` property is set to True. The `MandatoryColor` overwrites the control's default background color. In other words, if the user skips a mandatory field, the corresponding control is painted with the `MandatoryColor`, and it's not reset to the control's background color. Required fields behave like optional fields after they have been assigned a value.

If you have read the previous chapter, you should be able to implement these properties easily. Listing 12.1 is the code that implements the four custom properties. The values of the properties are stored in the private variables declared at the beginning of the listing. Then the control's properties are implemented as Property procedures.

LISTING 12.1: Property Procedures of the FocusedTextBox

```
Dim _mandatory As Boolean
Dim _enterFocusColor, _leaveFocusColor As Color
Dim _mandatoryColor As Color

Property Mandatory() As Boolean
    Get
        Mandatory = _mandatory
    End Get
    Set(ByVal value As Boolean)
        _mandatory = Value
    End Set
End Property

Property EnterFocusColor() As System.Drawing.Color
    Get
        Return _enterFocusColor
    End Get
    Set(ByVal value As System.Drawing.Color)
        _enterFocusColor = value
    End Set
End Property

Property MandatoryColor() As System.Drawing.Color
    Get
        Return _mandatoryColor
    End Get
    Set(ByVal value As System.Drawing.Color)
        _mandatoryColor = value
    End Set
End Property
```

The last step is to use these properties in the control's `Enter` and `Leave` events. When the control receives the focus, it changes its background color to `EnterFocusColor` to indicate that it's the active control on the form (the control with the focus). When it loses the focus, its background is restored to the usual background color, unless it's a required field and the user has left it blank. In this case, its background color is set to `MandatoryColor`. Listing 12.2 shows the code in the two focus-related events of the UserControl object.

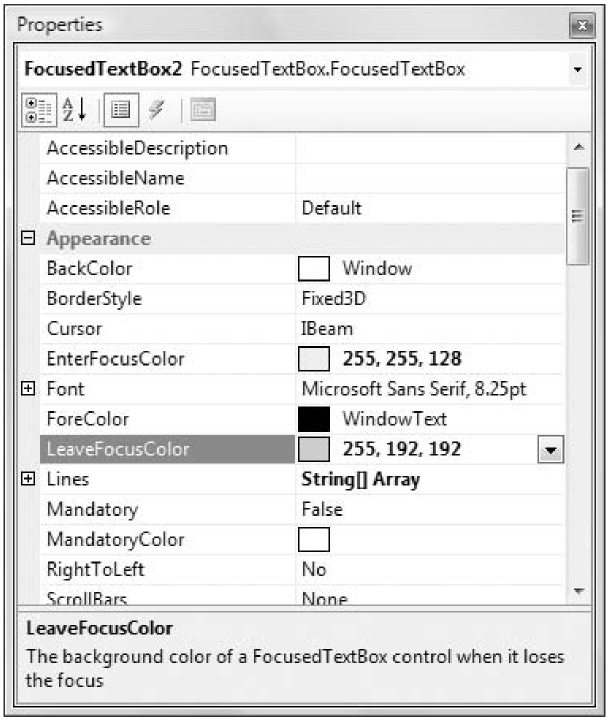
LISTING 12.2: *Enter* and *Leave* Events

```
Private _backColor As Color
Private Sub FocusedTextBox_Enter(...) Handles MyBase.Enter
   _backColor = Me.BackColor
   Me.BackColor = _enterFocusColor
End Sub
```

```
Private Sub FocusedTextBox_Leave(...) Handles MyBase.Leave
   If Trim(Me.Text).Length = 0 And _mandatory Then
      Me.BackColor = _mandatoryColor
   Else
      Me.BackColor = _backColor
   End If
End Sub
```

Testing the FocusedTextBox Control

Build the control again with the Build ➤ Build FocusedTextBox command and switch to the test form. Place several instances of the custom control on the form, align them, and then select each one and set its properties in the Properties window. The new properties are appended at the bottom of the Properties window, on the Misc tab (for miscellaneous properties). You will see shortly how to add each property under a specific category, as shown in Figure 12.3. Set the custom properties of a few controls on the form and then press F5 to run the application. See how the FocusedTextBox controls behave as you move the focus from one to the other and how they handle the mandatory fields.



AccessibleDescription	
AccessibleName	
AccessibleRole	Default
⊟ Appearance	
BackColor	Window
BorderStyle	Fixed3D
Cursor	IBeam
EnterFocusColor	**255, 255, 128**
⊞ Font	Microsoft Sans Serif, 8.25pt
ForeColor	WindowText
LeaveFocusColor	**255, 192, 192**
⊞ Lines	**String[] Array**
Mandatory	False
MandatoryColor	
RightToLeft	No
ScrollBars	None

LeaveFocusColor
The background color of a FocusedTextBox control when it loses the focus

Figure 12.3
Custom properties of the FocusedTextBox control in the Properties window

Pretty impressive, isn't it? I'm certain that many readers will incorporate this custom control in their projects — perhaps you may already be considering new features. Even if you have no use for an enhanced TextBox control, you'll agree that building it was quite simple. Next time you need to enhance one of the Windows controls, you know how to do it. Just build a new control that inherits from an existing control, add some custom members, and use it. Create a project with all the ''enhanced'' controls and use them regularly in your projects. All you have to do is add a reference to the DLL that implements the control in a new project, just like reusing a custom class.

Classifying the Control's Properties

Let's go back to our FocusedTextBox control — there are some loose ends to take care of. First, we must specify the category in the Properties window under which each custom property appears. By default, all the properties you add to a custom control are displayed in the Misc section of the Properties window. To specify that a property be displayed in a different section, use the `Category` attribute of the Property procedure. As you will see, properties have other attributes too, which you can set in your code as you design the control.

Properties have attributes, which appear in front of the property name and are enclosed in a pair of angle brackets. All attributes are members of the System.ComponentModel class, and you must import this class to the module that contains the control's code. The following attribute declaration in front of the property's name determines the category of the Properties window in which the specific property will appear:

```
<Category("Appearance")> Public Property ...
```

If none of the existing categories suits a specific property, you can create a new category in the Properties window by specifying its name in the `Category` attribute. If you have a few properties that should appear in a section called Conditional, insert the following attribute in front of the declarations of the corresponding properties:

```
<Category("Conditional")> Public Property ...
```

When this control is selected, the Conditional section will appear in the Properties window, and all the properties with this attribute under it.

Another attribute is the `Description` attribute, which determines the property's description that appears at the bottom of the Properties window when the property is selected. To specify multiple attributes, separate them with commas, as shown here:

```
<Description("Indicates whether the control can be left blank"), _
 Category("Appearance")> _
Property Mandatory() As Boolean
{ the property procedure's code }
End Property
```

The most important attribute is the `DefaultValue` attribute, which determines the property's default (initial) value. The `DefaultValue` attribute must be followed by the default value in parentheses:

```
<Description("Indicates whether the control can be left blank") _
 Category("Appearance"), DefaultValue(False)> _
 Property Mandatory() As Boolean
{ the property procedure's code }
```

Some attributes apply to the class that implements the custom controls. The `DefaultProperty` and `DefaultEvent` attributes determine the control's default property and event. To specify that `Mandatory` is the default property of the FocusedTextBox control, replace the class declaration with the following:

```
<DefaultProperty("Mandatory")> Public Class FocusedTextBox
```

Events are discussed later in the chapter, but you already know how to raise an event from within a class. Raising an event from within a control's code is quite similar. Open the FocusedTextBox project, examine its code, and experiment with new properties and methods.

As you may have noticed, all custom controls appear in the Toolbox with the same icon. You can specify the icon to appear in the Toolbox with the `ToolboxBitmap` attribute, whose syntax is the following, where *`imagepath`* is a string with the absolute path to a 16 × 16 pixel bitmap:

```
<ToolboxBitmap(imagepath)> Public Class FocusedTextBox
```

The bitmap is actually stored in the control's DLL and need not be distributed along with the control.

Now we're ready to move on to something more interesting. This time, we'll build a control that combines the functionality of several controls, which is another common scenario. You will literally design its visible interface by dropping controls on it, just like designing the visible interface of a Windows form.

Building Compound Controls

A *`compound control`* provides a visible interface that consists of multiple Windows controls. The controls that make up a compound control are known as *`constituent`* controls. As a result, this type of control doesn't inherit the functionality of any specific control. You must implement its properties and methods with custom code. This isn't as bad as it sounds, because a compound control inherits the UserControl object, which exposes quite a few members of its own (the `Anchoring` and `Docking` properties, for example, are exposed by the UserControl object, and you need not implement these properties — thank Microsoft). You will add your own members, and in most cases you'll be mapping the properties and methods of the compound controls to a property or method of one of its constituent controls. If your control contains a TextBox control, for example, you can map the custom control's `WordWrap` property to the equivalent property of the TextBox. The following property procedure demonstrates how to do it:

```
Property WordWrap() As Boolean
   Get
      WordWrap = TextBox1.WordWrap
   End Get
   Set(ByVal Value As Boolean)
      TextBox1.WordWrap = Value
   End Set
End Property
```

You don't have to maintain a private variable for storing the value of the custom control's `WordWrap` property. When this property is set, the Property procedure assigns the property's value to the `TextBox1.WordWrap` property. Likewise, when this property's value is requested, the procedure reads it from the constituent control and returns it. In effect, the custom control's `WordWrap` property affects directly the functionality of one of the constituent controls.

The same logic applies to events. Let's say your compound control contains a TextBox and a ComboBox control, and you want to raise the `TextChanged` event when the user edits the TextBox control, and the `SelectionChanged` event when the user selects another item in the ComboBox control. First, you must declare the two events:

```
Event TextChanged
Event SelectionChanged
```

Then, you must raise the two events from within the appropriate event handlers: the *TextChanged* event from the ***TextBox1*** control's *TextChanged* event handler, and the *SelectionChanged* event from the ***ComboBox1*** control's *SelectedIndexChanged* event handler:

```
Private Sub TextBox1_TextChanged(...) _
            Handles FocusedTextBox1.TextChanged
    RaiseEvent TextChanged()
End Sub

Private Sub ComboBox1_SelectedIndexChanged(...) _
            Handles ComboBox1.SelectedIndexChanged
    RaiseEvent SelectionChanged()
End Sub
```

VB 2008 at Work: The ColorEdit Control

In this section, you're going to build a compound control that's similar to the Color dialog box. The ColorEdit control allows you to specify a color by adjusting its red, green, and blue components with three scroll bars, or to select a color by name. The control's surface at runtime on a form is shown in Figure 12.4.

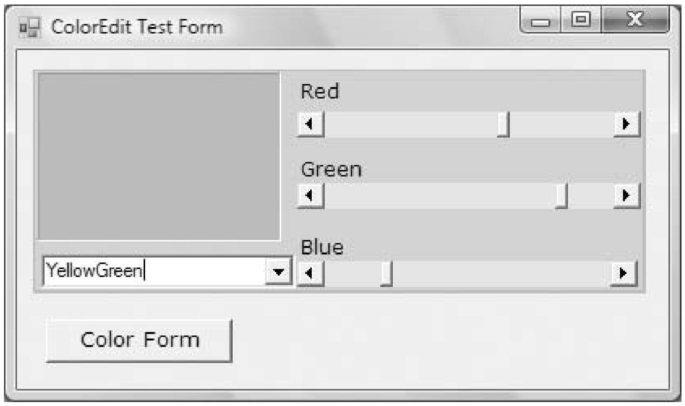


FIGURE 12.4
The ColorEdit control on a test form

Create a new Windows Control Library project, the **ColorEdit** project. Save the solution and then add a new Windows Application project, the TestProject, and make it the solution's startup project, just as you did with the first sample project of this chapter.

Now open the UserControl object and design its interface, as shown in Figure 12.4. Place the necessary controls on the UserControl object's surface and align them just as you would do with a Windows form. The three ScrollBar controls are named `RedBar`, `GreenBar`, and `BlueBar`, respectively. The `Minimum` property for all three controls is 0; the `Maximum` for all three is 255. This is the valid range of values for a color component. The control at the top-left corner is a Label control with its background color set to Black. (We could have used a PictureBox control in its place.) The role of this control is to display the selected color.

The ComboBox at the bottom of the custom control is the `NamedColors` control, which is populated with color names when the control is loaded. The Color class exposes 140 properties, which are color names (Beige, Azure, and so on). Don't bother entering all the color names in the ComboBox control; just open the ColorEdit project and you will find the `AddNamedColors()` subroutine, which does exactly that.

The user can specify a color by sliding the three ScrollBar controls or by selecting an item in the ComboBox control. In either case, the Label control's `Background` color will be set to the selected color. If the color is specified with the ComboBox control, the three ScrollBars will adjust to reflect the color's basic components (red, green, and blue). Not all possible colors that you can specify with the three ScrollBars have a name (there are approximately 16 million colors). That's why the ComboBox control contains the Unknown item, which is selected when the user specifies a color by setting its basic components.

Finally, the ColorEdit control exposes two properties: `NamedColor` and `SelectedColor`. The `NamedColor` property retrieves the selected color's name. If the color isn't selected from the ComboBox control, the value Unknown will be returned. The `SelectedColor` property returns or sets the current color. Its type is Color, and it can be assigned any expression that represents a color value. The following statement will assign the form's `BackColor` property to the `SelectedColor` property of the control:

```
UserControl1.SelectedColor = Me.BackColor
```

You can also specify a color value with the `FromARGB` method of the Color object:

```
UserControl1.SelectedColor = Color.FromARGB(red, green, blue)
```

The implementation of the `SelectedColor` property (shown in Listing 12.3) is straightforward. The `Get` section of the procedure assigns the Label's background color to the `SelectedColor` property. The `Set` section of the procedure extracts the three color components from the value of the property and assigns them to the three ScrollBar controls. Then it calls the `ShowColor` subroutine to update the display. (You'll see shortly what this subroutine does.)

Listing 12.3: *SelectedColor* Property Procedure

```
Property SelectedColor() As Color
   Get
      SelectedColor = Label1.BackColor
   End Get
   Set(ByVal Value As Color)
      HScrollBar1.Value = Value.R
      HScrollBar2.Value = Value.G
```

```
        HScrollBar3.Value = Value.B
        ShowColor()
    End Set
End Property
```

The `NamedColor` property (see Listing 12.4) is read-only and is marked with the `ReadOnly` keyword in front of the procedure's name. This property retrieves the value of the ComboBox control and returns it.

LISTING 12.4: *NamedColor* Property Procedure

```
ReadOnly Property NamedColor() As String
    Get
        NamedColor = ComboBox1.SelectedItem
    End Get
End Property
```

When the user selects a color name in the ComboBox control, the code retrieves the corresponding color value with the `Color.FromName` method. This method accepts a color name as an argument (a string) and returns a color value, which is assigned to the `namedColor` variable. Then the code extracts the three basic color components with the `R`, `G`, and `B` properties. (These properties return the red, green, and blue color components, respectively.) Listing 12.5 shows the code behind the ComboBox control's `SelectedIndexChanged` event, which is fired every time a new color is selected by name.

LISTING 12.5: Specifying a Color by Name

```
Private Sub ComboBox1_SelectedIndexChanged(...) _
                Handles ComboBox1.SelectedIndexChanged
    Dim namedColor As Color
    Dim colorName As String
    colorName = ComboBox1.SelectedItem
    If colorName <> "Unknown" Then
        namedColor = Color.FromName(colorName)
        HScrollBar1.Value = namedColor.R
        HScrollBar2.Value = namedColor.G
        HScrollBar3.Value = namedColor.B
        ShowColor()
    End If
End Sub
```

The `ShowColor`() subroutine simply sets the Label's background color to the value specified by the three ScrollBar controls. Even when you select a color value by name, the control's code sets the three ScrollBars to the appropriate values. This way, we don't have to write additional code to update the display. The `ShowColor`() subroutine is quite trivial:

```
Sub ShowColor()
   Label1.BackColor = Color.FromARGB(255, HScrollBar1.Value, _
           HScrollBar2.Value, HScrollBar3.Value)
End Sub
```

The single statement in this subroutine picks up the values of the three basic colors from the ScrollBar controls and creates a new color value with the `FromARGB` method of the Color object. The first argument is the transparency of the color (the *A*, or alpha channel), and we set it to 255 for a completely opaque color. You can edit the project's code to take into consideration the transparency channel as well. If you do, you must replace the Label control with a PictureBox control and display an image in it. Then draw a rectangle with the specified color on top of it. If the color isn't completely opaque, you'll be able to see the underlying image and visually adjust the transparency channel.

Testing the ColorEdit Control

To test the new control, you must place it on a form. Build the ColorEdit control and switch to the test project (add a new project to the current solution if you haven't done so already). Add an instance of the new custom control to the form. You don't have to enter any code in the test form. Just run it and see how you specify a color, either with the scroll bars or by name. You can also read the value of the selected color through the `SelectedColor` property. The code behind the Color Form button on the test form does exactly that (it reads the selected color and paints the form with this color):

```
Private Sub Button1_Click(...) Handles Button1.Click
    Me.BackColor = ColorEdit1.SelectedColor
End Sub
```

Building User-Drawn Controls

This is the most complicated but most flexible type of control. A user-drawn control consists of a UserControl object with no constituent controls. You are responsible for updating the control's visible area with the appropriate code, which must appear in the control's `OnPaint` method. (This method is invoked automatically every time the control's surface must be redrawn.)

To demonstrate the design of user-drawn controls, we'll develop the Label3D control, which is an enhanced Label control and is shown in Figure 12.5. It provides all the members of the Label control plus the capability to render its caption in three-dimensional type. The new custom control is called Label3D, and its project is the FlexLabel project. It contains the Label3D project (which is a Windows Control Library project) and the usual test project (which is a Windows Application project).

At this point, you're probably thinking about the code that aligns the text and renders it as carved or raised. A good idea is to start with a Windows project, which displays a string on a form and aligns it in all possible ways. A control is an application packaged in a way that allows it to be displayed on a form instead of on the Desktop. As far as the functionality is concerned, in most cases it can be implemented on a regular form. Conversely, if you can display 3D text on a form, you can do so with a custom control.

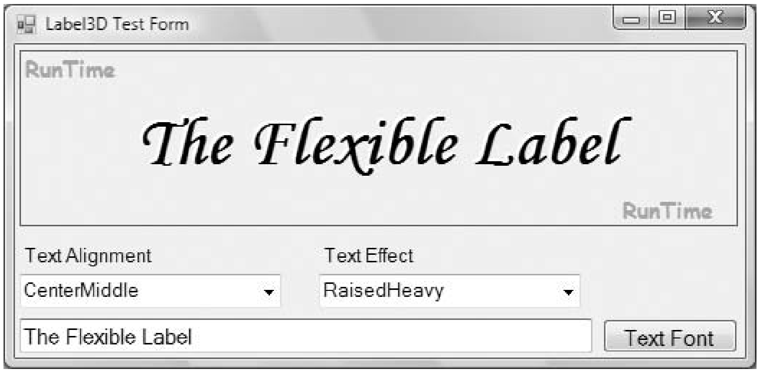


FIGURE 12.5
The Label3D control is an enhanced Label control.

Designing a Windows form with the same functionality is fairly straightforward. You haven't seen the drawing methods yet, but this control doesn't involve any advanced drawing techniques. All we need is a method to render strings on the control. To achieve the 3D effect, you must display the same string twice, first in white and then in black on top of the white. The two strings must be displaced slightly, and the direction of the displacement determines the effect (whether the text will appear as raised or carved). The amount of displacement determines the depth of the effect. Use a displacement of 1 pixel for a light effect, and a displacement of 2 pixels for a heavy one.

VB 2008 at Work: The Label3D Control

The first step of designing a user-drawn custom control is to design the control's interface: what it will look like when placed on a form (its visible interface) and how developers can access this functionality through its members (the programmatic interface). Sure, you've heard the same advice over and over, and many of you still start coding an application without spending much time designing it. In the real world, especially if you are not a member of a programming team, people design as they code (or the other way around).

The situation is quite different with Windows controls. Your custom control must provide properties, which will be displayed automatically in the Properties window. The developer should be able to adjust every aspect of the control's appearance by manipulating the settings of these properties. In addition, developers expect to see the standard properties shared by most controls (such as the background color, the text font, and so on) in the Properties window. You must carefully design the methods so that they expose all the functionality of the control that should be accessed from within the application's code, and the methods shouldn't overlap. Finally, you must provide the events necessary for the control to react to external events. Don't start coding a custom control unless you have formulated a clear idea of what the control will do and how developers will use it at design time.

LABEL3D CONTROL SPECIFICATIONS

The Label3D control displays a caption like the standard Label control, so it must provide a `Font` property, which lets the developer determine the label's font. The UserControl object exposes its own `Font` property, so we need not implement it in our code. In addition, the Label3D control can align its caption both vertically and horizontally. This functionality will be exposed by the `Alignment` property, whose possible settings are the members of the `Align` enumeration: *`TopLeft`*, *`TopMiddle`*, *`TopRight`*, *`CenterLeft`*, *`CenterMiddle`*, *`CenterRight`*, *`BottomLeft`*, *`BottomMiddle`*, and *`BottomRight`*. The (self-explanatory) values are the names that will appear in the drop-down list of the `Alignment` property in the Properties window.

Similarly, the text effect is manipulated through the `Effect` property, whose possible settings are the members of the `Effect3D` custom enumeration: *`None`*, *`Carved`*, *`CarvedHeavy`*, *`Raised`*, and *`RaisedHeavy`*. There are basically two types of effects (raised and carved text) and two variations on each effect (normal and heavy).

In addition to the custom properties, the Label3D control should also expose the standard properties of a Label control, such as `Tag`, `BackColor`, and so on. Developers expect to see standard properties in the Properties window, and you should implement them. The Label3D control doesn't have any custom methods, but it should provide the standard methods of the Label control, such as the `Move` method. Similarly, although the control doesn't raise any special events, it must support the standard events of the Label control, such as the mouse and keyboard events.

Most of the custom control's functionality exists already, and there should be a simple technique to borrow this functionality from other controls instead of implementing it from scratch. This is indeed the case: The UserControl object, from which all user-drawn controls inherit, exposes a large number of members.

Designing the Custom Control

Start a new project of the Windows Control Library type, name it **FlexLabel**, and then rename the UserControl1 object to **Label3D**. Open the UserControl object's code window and change the name of the class from UserControl1 to **Label3D**.

Every time you place a Windows control on a form, it's named according to the UserControl object's name and a sequence digit. The first instance of the custom control you place on a form will be named *`Label3D1`*, the next one will be named *`Label3D2`*, and so on. Obviously, it's important to choose a meaningful name for your UserControl object.

As you will soon see, the UserControl is the ''form'' on which the custom control will be designed. It looks, feels, and behaves like a regular VB form, but it's called a UserControl. UserControl objects have additional unique properties that don't apply to a regular form, but to start designing new controls, think of them as regular forms.

You've set the scene for a new user-drawn Windows control. Start by declaring the `Align` and `Effect3D` enumerations, as shown in Listing 12.6.

Listing 12.6: *Align* and *Effect3D* Enumerations

```
Public Enum Align
   TopLeft
   TopMiddle
   TopRight
   CenterLeft
   CenterMiddle
   CenterRight
   BottomLeft
   BottomMiddle
   BottomRight
End Enum

Public Enum Effect3D
   None
   Raised
```

```
      RaisedHeavy
      Carved
      CarvedHeavy
   End Enum
```

The next step is to implement the `Alignment` and `Effect` properties. Each property's type is an enumeration; Listing 12.7 shows the implementation of the two properties.

LISTING 12.7: *Alignment* and *Effect* Properties

```
   Private Shared mAlignment As Align
   Private Shared mEffect As Effect3D
   Public Property Alignment() As Align
      Get
         Alignment = mAlignment
      End Get
      Set(ByVal Value As Align)
         mAlignment = Value
         Invalidate()
       End Set
   End Property

   Public Property Effect() As Effect3D
      Get
         Effect = mEffect
      End Get
      Set(ByVal Value As Effect3D)
         mEffect = Value
         Invalidate()
      End Set
   End Property
```

The current settings of the two properties are stored in the private variables *`mAlignment`* and *`mEffect`*. When either property is set, the Property procedure's code calls the `Invalidate` method of the UserControl object to force a redraw of the string on the control's surface. The call to the `Invalidate` method is required for the control to operate properly in design mode. You can provide a method to redraw the control at runtime (although developers shouldn't have to call a method to refresh the control every time they set a property), but this isn't possible at design time. In general, when a property is changed in the Properties window, the control should be able to update itself and reflect the new property setting, and this is done with a call to the `Invalidate` method. Shortly, you'll see an even better way to automatically redraw the control every time a property is changed.

Finally, you must add one more property, the `Caption` property, which is the string to be rendered on the control. Declare a private variable to store the control's caption (the *`mCaption`* variable) and enter the code from Listing 12.8 to implement the `Caption` property.

LISTING 12.8: *Caption* Property Procedure

```
Private mCaption As String
Property Caption() As String
   Get
      Caption = mCaption
   End Get
   Set(ByVal Value As String)
      mCaption = Value
      Invalidate()
   End Set
End Property
```

The core of the control's code is in the `OnPaint` method, which is called automatically before the control repaints itself. The same event's code is also executed when the `Invalidate` method is called, and this is why we call this method every time one of the control's properties changes value. The `OnPaint` method enables you to take control of the paint process and supply your own code for painting the control's surface. The single characteristic of all user-drawn controls is that they override the default `OnPaint` method. This is where you must insert the code to draw the control's surface — that is, draw the specified string, taking into consideration the `Alignment` and `Effect` properties. The `OnPaint` method's code is shown in Listing 12.9.

LISTING 12.9: UserControl Object's *OnPaint* Method

```
Protected Overrides Sub OnPaint( _
      ByVal e As System.Windows.Forms.PaintEventArgs)
    Dim lblFont As Font = Me.Font
    Dim lblBrush As New SolidBrush(Color.Red)
    Dim X, Y As Integer
    Dim textSize As SizeF = _
          e.Graphics.MeasureString(mCaption, lblFont)
    Select Case Me.mAlignment
        Case Align.BottomLeft
            X = 2
            Y = Convert.ToInt32(Me.Height - textSize.Height)
        Case Align.BottomMiddle
            X = CInt((Me.Width - textSize.Width) / 2)
            Y = Convert.ToInt32(Me.Height - textSize.Height)
        Case Align.BottomRight
            X = Convert.ToInt32(Me.Width - textSize.Width - 2)
            Y = Convert.ToInt32(Me.Height - textSize.Height)
        Case Align.CenterLeft
            X = 2
            Y = Convert.ToInt32((Me.Height - textSize.Height) / 2)
        Case Align.CenterMiddle
            X = Convert.ToInt32((Me.Width - textSize.Width) / 2)
```

```
                Y = Convert.ToInt32((Me.Height - textSize.Height) / 2)
            Case Align.CenterRight
                X = Convert.ToInt32(Me.Width - textSize.Width - 2)
                Y = Convert.ToInt32((Me.Height - textSize.Height) / 2)
            Case Align.TopLeft
                X = 2
                Y = 2
            Case Align.TopMiddle
                X = Convert.ToInt32((Me.Width - textSize.Width) / 2)
                Y = 2
                Case Align.TopRight
                X = Convert.ToInt32(Me.Width - textSize.Width - 2)
                Y = 2
        End Select
        Dim dispX, dispY As Integer
        Select Case mEffect
            Case Effect3D.None : dispX = 0 : dispY = 0
            Case Effect3D.Raised : dispX = 1 : dispY = 1
            Case Effect3D.RaisedHeavy : dispX = 2 : dispY = 2
            Case Effect3D.Carved : dispX = -1 : dispY = -1
            Case Effect3D.CarvedHeavy : dispX = -2 : dispY = -2
        End Select
        lblBrush.Color = Color.White
        e.Graphics.DrawString(mCaption, lblFont, lblBrush, X, Y)
        lblBrush.Color = Me.ForeColor
        e.Graphics.DrawString(mCaption, lblFont, lblBrush, X + dispX, Y + dispY)

        End If
    End Sub
```

This subroutine calls for a few explanations. The `Paint` method passes a `PaintEventArgs` argument (the ubiquitous `e` argument). This argument exposes the `Graphics` property, which represents the control's surface. The `Graphics` object exposes all the methods you can call to create graphics on the control's surface. The `Graphics` object is discussed in detail in Chapter 18, ''Drawing and Painting with Visual Basic 2008,'' but for this chapter all you need to know is that the `MeasureString` method returns the dimensions of a string when rendered in a specific font, and the `DrawString` method draws the string in the specified font. The first `Select Case` statement calculates the coordinates of the string's origin on the control's surface, and these coordinates are calculated differently for each type of alignment. Then another `Select Case` statement sets the displacement between the two strings, so that when superimposed they produce a three-dimensional effect. Finally, the code draws the string of the `Caption` property on the `Graphics` object. It draws the string in white first, then in black. The second string is drawn *`dispX`* pixels to the left and *`dispY`* pixels below the first one to give the 3D effect. The values of these two variables are determined by the setting of the `Effect` property.

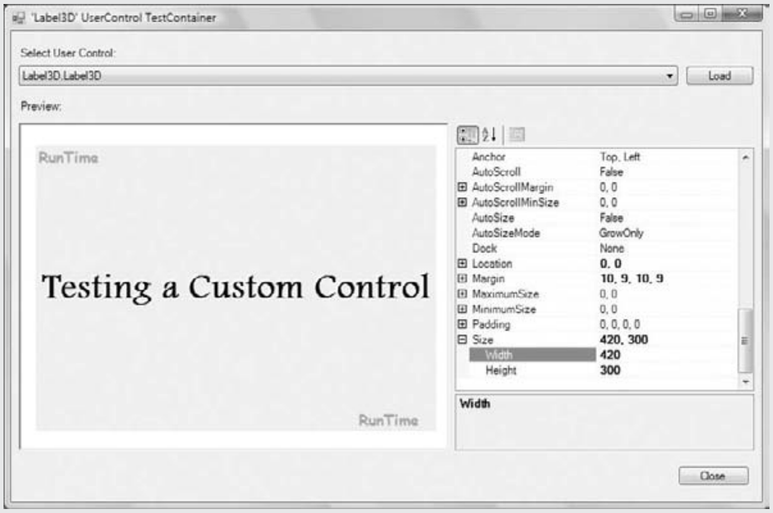
The event handler of the sample project contains a few more statements that are not shown here. These statements print the strings `DesignTime` and `RunTime` in a light color on the control's background, depending on the current status of the control. They indicate whether the control is currently in design (if the `DesignMode` property is True) or runtime (if `DesignMode` is False), and you will remove them after testing the control.

Testing Your New Control

To test your new control, you must first add it to the Toolbox and then place instances of it on the test form. You can add a form to the current project and test the control, but you shouldn't add more components to the control project. It's best to add a new project to the current solution.

A Quick Way to Test Custom Windows Controls

Visual Studio 2008 introduced a new, simple method of testing custom controls. Instead of using a test project, you can press F5 to ''run'' the Windows Control project. Right-click the name of the Label3D project (the Windows Control project in the solution) in Solution Explorer and from the context menu choose Set As Startup Project. Then press F5 to start the project. A dialog box (shown in the following figure) will appear with the control at runtime and its Properties window.



```
Anchor              Top, Left
AutoScroll          False
⊞ AutoScrollMargin  0, 0
⊞ AutoScrollMinSize 0, 0
AutoSize            False
AutoSizeMode        GrowOnly
Dock                None
⊞ Location          0, 0
⊞ Margin            10, 9, 10, 9
⊞ MaximumSize       0, 0
⊞ MinimumSize       0, 0
⊞ Padding           0, 0, 0, 0
⊟ Size              420, 300
    Width           420
    Height          300

Width
```

In this dialog box, you can edit any of the control's properties and see how they affect the control at runtime. If the control reacts to any user actions, you can see how the control's code behaves at runtime.

You can't test the control's methods, or program its events, but you'll get an idea of how the control will behave when placed on a form. Use this dialog box while you're developing the control's interface to see how it will behave when placed on a test form and how it reacts when you change its properties. When you're happy with the control's interface, you should test it with a Windows project, from which you can call its methods and program its events.

Add the TestProject to the current solution and place on its main form a Label3D control, as well as the other controls shown earlier in Figure 12.5. If the Label3D icon doesn't appear in the

Toolbox, build the control's project, and a new item will be added to the FlexLabel Components tab of the ToolBox.

Now double-click the Label3D control on the form to see its events. Your new control has its own events, and you can program them just as you would program the events of any other control. Enter the following code in the control's `Click` event:

```
Private Sub Label3D1_Click(...) Handles Label3D1.Click
    MsgBox(My properties are "& vbCrLf & _
           Caption = "  Label3D1.Caption.ToString & vbCrLf & _
           Alignment = "  Label3D1.Alignment.ToString & vbCrLf & _
           Effect = "  Label3D1.Effect.ToString)
End Sub
```

To run the control, press F5 and then click the control. You will see the control's properties displayed in a message box.

The other controls on the test form allow you to set the appearance of the custom control at runtime. The two ComboBox controls are populated with the members of the appropriate enumeration when the form is loaded. In their `SelectedIndexChanged` event handler, you must set the corresponding property of the FlexLabel control to the selected value, as shown in the following code:

```
Private Sub AlignmentBox_SelectedIndexChanged(...) _
            Handles AlignmentBox.SelectedIndexChanged
   Label3D1.Alignment = AlignmentBox.SelectedItem
End Sub

Private Sub EffectsBox_SelectedIndexChanged(...) _
            Handles EffectsBox.SelectedIndexChanged
   Label3D1.Effect = EffectsBox.SelectedItem
End Sub
```

The TextBox control at the bottom of the form stores the `Caption` property. Every time you change this string, the control is updated because the `Set` procedure of the `Caption` property calls the `Invalidate` method.

Changed Events

The UserControl object exposes many of the events you need to program the control, such as the key and mouse events. In addition, you can raise custom events. The Windows controls raise an event every time a property value is changed. If you examine the list of events exposed by the Label3D control, you'll see the `FontChanged` and `SizeChanged` events. These events are provided by the UserControl object. As a control developer, you should expose similar events for your custom properties, the `OnAlignmentChanged`, `OnEffectChanged`, and `OnCaptionChanged` events. This isn't difficult to do, but you must follow a few steps. Start by declaring an event handler for each of the `Changed` events:

```
Private mOnAlignmentChanged As EventHandler
Private mOnEffectChanged As EventHandler
Private mOnCaptionChanged As EventHandler
```

Then declare the actual events and their handlers:

```
Public Event AlignmentChanged(ByVal sender As Object, _
                              ByVal ev As EventArgs)
Public Event EffectChanged(ByVal sender As Object, _
                           ByVal ev As EventArgs)
Public Event CaptionChanged(ByVal sender As Object,
                            ByVal ev As EventArgs)
```

When a property changes value, you must call the appropriate method. In the `Set` section of the `Alignment` property procedure, insert the following statement:

```
OnAlignmentChanged(EventArgs.Empty)
```

And finally, invoke the event handlers from within the appropriate *`OnEventName`* method:

```
Protected Overridable Sub OnAlignmentChanged(ByVal e As EventArgs)
   Invalidate()
   If Not (mOnAlignmentChanged Is Nothing) Then _
                 mOnAlignmentChanged.Invoke(Me, e)
End Sub

Protected Overridable Sub OnEffectChanged(ByVal e As EventArgs)
   Invalidate()
   If Not (mOnEffectChanged Is Nothing) Then _
                    mOnEffectChanged.Invoke(Me, e)
End Sub

Protected Overridable Sub OnCaptionChanged(ByVal e As EventArgs)
   Invalidate()
   If Not (mOnCaptionChanged Is Nothing) Then _
                    mOnCaptionChanged.Invoke(Me, e)
End Sub
```

As you can see, the `OnPropertyChanged` events call the `Invalidate` method to redraw the control when a property's value is changed. As a result, you can now remove the call to the `Invalidate` method from the Property `Set` procedures. If you switch to the test form, you will see that the custom control exposes the `AlignmentChanged`, `EffectChanged`, and `CaptionChanged` events. The `OnCaptionChanged` method is executed automatically every time the `Caption` property changes value, and it fires the `CaptionChanged` event. The developer using the Label3D control shouldn't have to program this event.

Raising Custom Events

When you select the custom control in the Objects drop-down list of the editor and expand the list of events for this control, you'll see all the events fired by UserControl. Let's add a custom event for our control. To demonstrate how to raise events from within a custom control, we'll return for a moment to the ColorEdit control you developed a little earlier in this chapter.

Let’s say you want to raise an event (the ColorClick event) when the user clicks the Label control displaying the selected color. To raise a custom event, you must declare it in your control and call the RaiseEvent method. Note that the same event may be raised from many different places in the control’s code.

To declare the ColorClick event, enter the following statement in the control’s code. This line can appear anywhere, but placing it after the private variables that store the property values is customary:

```
Public Event ColorClick(ByVal sender As Object, ByVal e As EventArgs)
```

To raise the ColorClick event when the user clicks the Label control, insert the following statement in the Label control’s Click event handler:

```
Private Sub Label1_Click(...) Handles Label1.Click
   RaiseEvent ColorClick(Me, e)
End Sub
```

Raising a custom event from within a control is as simple as raising an event from within a class. It’s actually simpler to raise a custom event than to raise the usual PropertyChanged events, which are fired from within the OnPropertyChanged method of the base control.

The RaiseEvent statement in the Label’s Click event handler maps the Click event of the Label control to the ColorClick event of the custom control. If you switch to the test form and examine the list of events of the ColorEdit control on the form, you’ll see that the new event was added. The ColorClick event doesn’t convey much information. When raising custom events, it’s likely that you’ll want to pass additional information to the developer.

Let’s say you want to pass the Label control’s color to the application through the second argument of the ColorClick event. The EventArgs type doesn’t provide a Color property, so we must build a new type that inherits all the members of the EventArgs type and adds a property: the Color property. You can probably guess that we’ll create a custom class that inherits from the EventArgs class and adds the Color member. Enter the statements of Listing 12.10 at the end of the file (after the existing End Class statement).

LISTING 12.10: Declaring a Custom Event Type

```
Public Class ColorEvent
   Inherits EventArgs
   Public color As Color
End Class
```

Then, declare the following event in the control’s code:

```
Public Event ColorClick(ByVal sender As Object, ByVal e As ColorEvent)
```

And finally, raise the ColorClick event from within the Label’s Click event handler (see Listing 12.11).

LISTING 12.11: Raising a Custom Event

```
Private Sub Label1_Click(...) Handles Label1.Click
   Dim ev As ColorEvent
   ev.color = Label1.BackColor
   RaiseEvent ColorClick(Me, ev)
End Sub
```

Not all events fired by a custom control are based on property value changes. You can fire events based on external conditions or a timer. The AlarmControl sample project, which isn't discussed in this chapter because of space limitations, demonstrates how to design an alarm that can be set to go off at a certain time and trigger a `TimeOut` event. To examine the sample projects code, open the AlarmControl project with Visual Studio. In the project's folder, you will find a Readme file with a detailed discussion of the application.

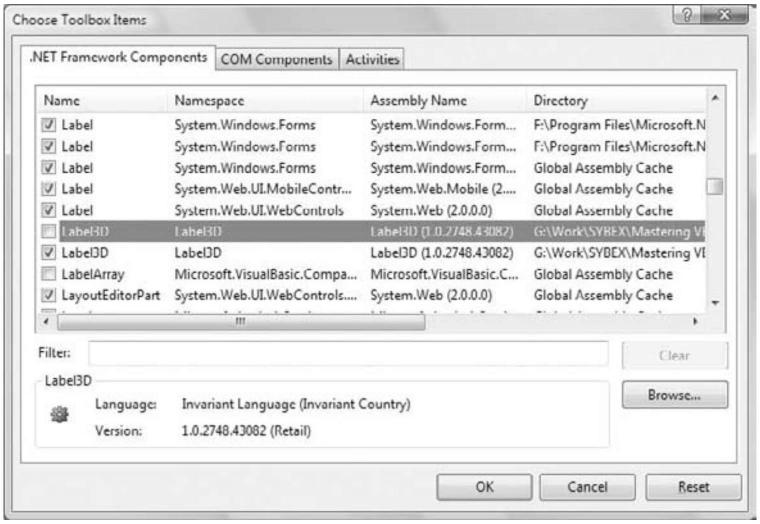
Using the Custom Control in Other Projects

By adding a test project to the Label3D custom control project, we designed and tested the control in the same environment. A great help, indeed, but the custom control can't be used in other projects. If you start another instance of Visual Studio and attempt to add your custom control to the Toolbox, you won't see the Label3D entry there.

To add your custom component in another project, open the Choose Toolbox Items dialog box and then click the .NET Framework Components tab. Be sure to carry out the steps described here while the .NET Framework Components tab is visible. If the COM Components tab is visible instead, you can perform the same steps, but you'll end up with an error message (because the custom component is not a COM component).

Click the Browse button in the dialog box and locate the `FlexLabel`.dll file. It's in the Bin folder under the FlexLabel project's folder. The Label3D control will be added to the list of .NET Framework components, as shown in Figure 12.6. Select the check box in front of the control's name; then click the OK button to close the dialog box and add Label3D to the Toolbox. Now you can use this control in your new project.

FIGURE 12.6
Adding the Label3D control to another project's Toolbox



Name	Namespace	Assembly Name	Directory
✓ Label	System.Windows.Forms	System.Windows.Form...	F:\Program Files\Microsoft.N
✓ Label	System.Windows.Forms	System.Windows.Form...	F:\Program Files\Microsoft.N
✓ Label	System.Windows.Forms	System.Windows.Form...	Global Assembly Cache
✓ Label	System.Web.UI.MobileContr...	System.Web.Mobile (2....	Global Assembly Cache
✓ Label	System.Web.UI.WebControls	System.Web (2.0.0.0)	Global Assembly Cache
☐ Label3D	Label3D	Label3D (1.0.2748.43082)	G:\Work\SYBEX\Mastering VI
✓ Label3D	Label3D	Label3D (1.0.2748.43082)	G:\Work\SYBEX\Mastering VI
☐ LabelArray	Microsoft.VisualBasic.Compa...	Microsoft.VisualBasic.C...	Global Assembly Cache
✓ LayoutEditorPart	System.Web.UI.WebControls....	System.Web (2.0.0.0)	Global Assembly Cache

Label3D
Language: Invariant Language (Invariant Country)
Version: 1.0.2748.43082 (Retail)

Designing Irregularly Shaped Controls

The UserControl object has a rectangular shape by default. However, a custom control need not be rectangular. It's possible to create irregularly shaped forms, too, but unlike irregularly shaped controls, an irregularly shaped form is still quite uncommon. Irregularly shaped controls are used in fancy interfaces, and they usually react to movement of the mouse. (They may change color when the mouse is over them or when they're clicked, for example.)

To change the default shape of a custom control, you must use the `Region` object, which is another graphics-related object that specifies a closed area. You can even use Bezier curves to make highly unusual and smooth shapes for your controls. In this section, we'll do something less ambitious: We'll create controls with the shape of an ellipse, as shown in the upper half of Figure 12.7. To follow the code presented in this section, open the NonRectangularControl project; the custom control is the RoundControl Windows Control Library project, and `Form1` is the test form for the control.

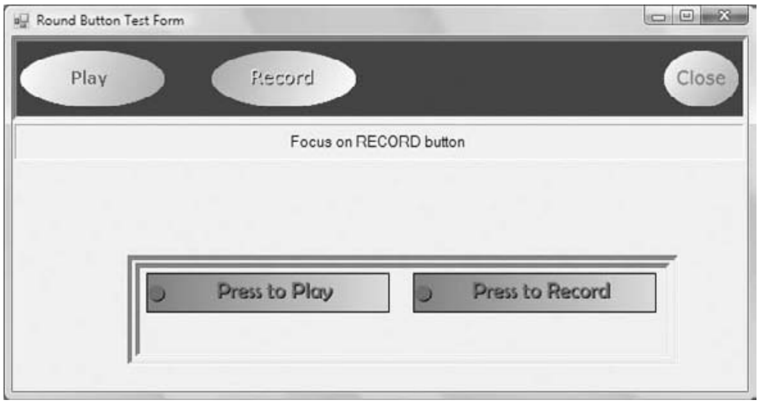


FIGURE 12.7
A few instances of an ellipse-shaped control

You can turn any control to any shape you like by creating the appropriate `Region` object and then applying it to the Region property of the control. This must take place from within the control's `Paint` event. Listing 12.12 shows the statements that change the shape of the control.

LISTING 12.12: Creating a Nonrectangular Control

```
Protected Sub PaintControl(ByVal sender As Object, _
           ByVal pe As PaintEventArgs) Handles Me.Paint
    pe.Graphics.TextRenderingHint = _
              Drawing.Text.TextRenderingHint.AntiAlias
    Dim roundPath As New GraphicsPath()
    Dim R As New Rectangle(0, 0, Me.Width, Me.Height)
    roundPath.AddEllipse(R)
    Me.Region = New Region(roundPath)
End Sub
```

First, we retrieve the Graphics object of the UserControl; then we create a GraphicsPath object, the *`roundPath`* variable, and add an ellipse to it. The ellipse is based on the enclosing rectangle.

The *R* object is used temporarily to specify the ellipse. The new path is then used to create a `Region` object, which is assigned to the Region property of the UserControl object. This gives our control the shape of an ellipse.

Listing 12.12 shows the statements that specify the control's shape. In addition, you must insert a few statements to display the control's caption, which is specified by the control's `Caption` property. The caption is rendered normally in yellow color, unless the mouse is hovering over the control, in which case the same caption is rendered with a 3D effect. You already know how to achieve this effect: by printing the same string twice in different colors with a slight displacement between them.

Listing 12.13 shows the code in the control's `MouseEnter` and `MouseLeave` events. When the mouse enters the control's area (this is detected by the control automatically — you won't have to write a single line of code for it), the *`currentState`* variable is set to *`State.Active`* (`State` is an enumeration in the project's code), and the control's caption appears in raised type. In the control's `MouseLeave` event handler, the *`currentState`* variable is reset to *`State.Inactive`* and the control's caption appears in regular font. In addition, each time the mouse enters and leaves the control, the `MouseInsideControl` and `MouseOutsideControl` custom events are fired.

Listing 12.13: RoundButton Control's *MouseEnter* and *MouseLeave* Events

```
Private Sub RoundButton_MouseEnter(...) _
            Handles MyBase.MouseEnter
    currentState = State.Active
    Me.Refresh()
    RaiseEvent MouseInsideButton(Me)
End Sub

Private Sub RoundButton_MouseLeave(...) _
            Handles MyBase.MouseLeave
    currentState = State.Inactive
    Me.Refresh()
    RaiseEvent MouseOusideButton(Me)
End Sub
```

These two events set up the appropriate variables, and the drawing of the control takes place in the `Paint` event's handler, which is shown in Listing 12.14.

Listing 12.14: RoundButton Control's *Paint* Event Handler

```
Protected Sub PaintControl(ByVal sender As Object, _
              ByVal pe As PaintEventArgs) _
              Handles Me.Paint
    pe.Graphics.TextRenderingHint =   Drawing.Text.TextRenderingHint.AntiAlias
    Dim roundPath As New GraphicsPath()
    Dim R As New Rectangle(0, 0, Me.Width, Me.Height)
    roundPath.AddEllipse(R)
    Me.Region = New Region(roundPath)
```

```
        Dim Path As New GraphicsPath
        Path.AddEllipse(R)
        Dim grBrush As LinearGradientBrush
        If currentState = State.Active Then
            grBrush = New LinearGradientBrush( _
                      New Point(0, 0), _
                      New Point(R.Width, R.Height), _
                      Color.DarkGray, Color.White)
        Else
            grBrush = New LinearGradientBrush( _
                      New Point(R.Width, R.Height), _
                      New Point(0, 0), Color.DarkGray, _
                      Color.White)
        End If
        pe.Graphics.FillPath(grBrush, Path)
        Dim X As Integer = _
                (Me.Width - pe.Graphics.MeasureString( _
                 currentCaption, currentFont).Width) / 2
        Dim Y As Integer = (Me.Height - pe.Graphics.MeasureString( _
                 currentCaption, currentFont).Height) / 2
        If currentState = State.Active Then
            pe.Graphics.DrawString(currentCaption, _
                    currentFont, Brushes.Black, X, Y)
            pe.Graphics.DrawString(currentCaption, _
                    currentFont, _
                    New SolidBrush(currentCaptionColor), X - 1, Y - 1)
        Else
            pe.Graphics.DrawString(currentCaption, _
                    currentFont, _
                    New SolidBrush(currentCaptionColor), X, Y)
        End If
    End Sub
```

The `OnPaint` method uses graphics methods to fill the control with a gradient and center the string on the control. They're the same methods we used in the example of the user-drawn control earlier in this chapter. The drawing methods are discussed in detail in Chapter 18.

The code uses the *`currentState`* variable, which can take on two values: *`Active`* and *`Inactive`*. These two values are members of the `State` enumeration, which is shown next:

```
Public Enum State
   Active
   Inactive
End Enum
```

The test form of the project shows how the RoundButton control behaves on a form. You can use the techniques described in this section to make a series of round controls for a totally different feel and look.

The Play button's `Click` event handler in the test form changes the caption of the button according to the control's current state. It also disables the other RoundButton controls on the test form. Here's the `Click` event handler of the Play button:

```
Private Sub bttnplay_Click(...) Handles bttnPlay.Click
    If bttnPlay.Caption = "Play" Then
        Label1.Text = "Playing..."
        bttnPlay.Caption = "STOP"
        bttnPlay.Color = Color.Red
        bttnRecord.Enabled = False
        bttnClose.Enabled = False
    Else
        Label1.Text = "Stoped Playing"
        bttnPlay.Caption = "Play"
        bttnPlay.Color = Color.Yellow
        bttnRecord.Enabled = True
        bttnClose.Enabled = True
    End If
End Sub
```

In Chapter 18, you'll learn more about shapes and paths, and you may wish to experiment with other oddly shaped controls. How about a progress indicator control that looks like a thermometer? Or a button with an LED that turns on or changes color when you press the button, like the buttons in the lower half of Figure 12.7? The two rectangular buttons are instances of the LEDButton custom control, which is included in the NonRectangularControl project. Open the project in Visual Studio and examine the code that renders the rectangular buttons emulating an LED in the left corner of the control.

Customizing List Controls

In this section, I'll show you how to customize the list controls (such as the ListBox, ComboBox, and TreeView controls). You won't build new custom controls in this section; actually, you'll hook custom code into certain events of a control to take charge of the rendering of its items.

Some of the Windows controls can be customized far more than it is possible through their properties. These are the list controls that allow you to supply your own code for drawing each item. You can use this technique to create a ListBox control that displays its items in different fonts, uses alternating background colors, and so on. You can even put bitmaps on the background of each item, draw the text in any color, and create items of varying heights. This is an interesting technique because without it, as you recall from our discussion of the ListBox control, all items have the same height and you must make the control wide enough to fit the longest item (if this is known at design time). The controls that allow you to take charge of the rendering process of their items are the ListBox, CheckedListBox, ComboBox, and TreeView controls.

To create an owner-drawn control, you must program two events: the `MeasureItem` and `DrawItem` events. In the `MeasureItem` event, you determine the dimensions of the rectangle in which the drawing will take place. In the `DrawItem` event, you insert the code for rendering the items on the control. Every time the control is about to display an item, it fires the `MeasureItem` event first and then the `DrawItem` event. By inserting the appropriate code in the two event handlers, you can take control of the rendering process.

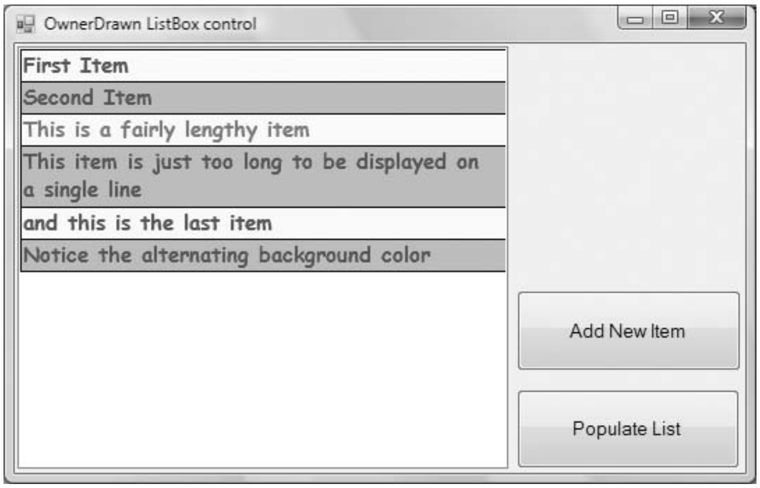
These two events don't take place unless you set the DrawMode property of the control accordingly. Because only controls that expose the DrawMode property can be owner-drawn, you have a quick way of figuring out whether a control's appearance can be customized with the techniques discussed in this section. The DrawMode property can be set to *Normal* (the control draws its own surface), *OwnerDrawnFixed* (you can draw the control, but the height of the drawing area remains fixed), or *OwnerDrawnVariable* (you can draw the control and use a different height for each item). The same property for the TreeView control has three different settings: *None*, *OwnerDrawText* (you provide the text for each item), and *OwnerDrawAll* (you're responsible for drawing each node's rectangle).

Designing Owner-Drawn ListBox Controls

The default look of the ListBox control will work fine with most applications, but you might have to create owner-drawn ListBoxes if you want to use different colors or fonts for different types of items, or to populate the list with items of widely different lengths.

The example you'll build in this section, shown in Figure 12.8, uses an alternating background color, and each item has a different height, depending on the string it holds. Lengthy strings are broken into multiple lines at word boundaries. Because you're responsible for breaking the string into lines, you can use any other technique — for example, you can place an ellipsis to indicate that the string is too long to fit on the control, use a smaller font, and so on. The fancy ListBox of Figure 12.8 was created with the OwnerDrawnList project.

FIGURE 12.8
An unusual, but quite functional, ListBox control


OwnerDrawn ListBox control
First Item
Second Item
This is a fairly lengthy item
This item is just too long to be displayed on a single line
and this is the last item
Notice the alternating background color
Add New Item
Populate List

To custom-draw the items in a ListBox control (or a ComboBox, for that matter), you use the MeasureItem event to calculate the item's dimensions, and the DrawItem event to actually draw the item. Each item is a rectangle that exposes a Graphics object, and you can call any of the Graphics object's drawing methods to draw on the item's area. The drawing techniques we'll use in this example are similar to the ones we used in the previous section, but after you learn more about the drawing methods in Chapter 18, you can create even more elaborate designs than the ones shown here.

Each time an item is about to be drawn, the *MeasureItem* and *DrawItem* events are fired in this order. In the *MeasureItem* event handler, we set the dimensions of the item with the statements shown in Listing 12.15.

Listing 12.15: Setting Up an Item's Rectangle in an Owner-Drawn ListBox Control

```
Private Sub ListBox1_MeasureItem(ByVal sender As Object, _
               ByVal e As System.Windows.Forms.MeasureItemEventArgs) _
               Handles ListBox1.MeasureItem
   If fnt Is Nothing Then Exit Sub
   Dim itmSize As SizeF
   Dim S As New SizeF(ListBox1.Width, 200)
   itmSize = e.Graphics.MeasureString(ListBox1.Items(e.Index).ToString, fnt, S)
   e.ItemHeight = itmSize.Height
   e.ItemWidth = itmSize.Width
End Sub
```

The `MeasureString` method of the Graphics object accepts as arguments a string, the font in which the string will be rendered, and a SizeF object. The SizeF object provides two members: the `Width` and `Height` members, which you use to pass to the method information about the area in which we want to print the string. In our example, we'll print the string in a rectangle that's as wide as the ListBox control and as tall as needed to fit the entire string. I'm using a height of 200 pixels (enough to fit the longest string that users might throw at the control). Upon return, the `MeasureString` method sets the members of the SizeF object to the width and height actually required to print the string.

The two members of the SizeF object are then used to set the dimensions of the current item (properties `e.ItemWidth` and `e.ItemHeight`). The custom rendering of the current item takes place in the `ItemDraw` event handler, which is shown in Listing 12.16. The `Bounds` property of the handler's `e` argument reports the dimensions of the item's cell as you calculated them in the `MeasureItem` event handler.

Listing 12.16: Drawing an Item in an Owner-Drawn ListBox Control

```
Private Sub ListBox1_DrawItem(ByVal sender As Object, _
               ByVal e As System.Windows.Forms.DrawItemEventArgs) _
               Handles ListBox1.DrawItem
   If e.Index = -1 Then Exit Sub
   e.DrawBackground()
   Dim txtBrush As SolidBrush
   Dim bgBrush As SolidBrush
   Dim txtfnt As Font
   If e.Index / 2 = CInt(e.Index / 2) Then
   ' color even numbered items
      txtBrush = New SolidBrush(Color.Blue)
      bgBrush = New SolidBrush(Color.LightYellow)
   Else
   ' color odd numbered items
      txtBrush = New SolidBrush(Color.Blue)
      bgBrush = New SolidBrush(Color.Cyan)
   End If
```

```
      If e.State And DrawItemState.Selected Then
      ' use red color and bold for the selected item
         txtBrush = New SolidBrush(Color.Red)
         txtfnt = New Font(fnt.Name, fnt.Size, FontStyle.Bold)
      Else
         txtfnt = fnt
      End If
      e.Graphics.FillRectangle(bgBrush, e.Bounds)
      e.Graphics.DrawRectangle(Pens.Black, e.Bounds)
      Dim R As New RectangleF(e.Bounds.X, e.Bounds.Y, _
                        e.Bounds.Width, e.Bounds.Height)
      e.Graphics.DrawString(ListBox1.Items(e.Index).ToString, txtfnt, txtBrush, R)
      e.DrawFocusRectangle()
   End Sub
```

To test the custom-drawn ListBox control, place two buttons on the form, as shown in Figure 12.8. The Add New Item button prompts the user for a new item (a string) and adds it to the control's `Items` collection. Listing 12.17 shows the code that adds a new item to the list.

Listing 12.17: Adding an Item to the List at Runtime

```
   Private Sub Button2_Click(...) Handles Button2.Click
      Dim newItem As String
      newItem = InputBox("Enter item to add to the list")
      ListBox1.Items.Add(newItem)
   End Sub
```

The Bottom Line

Extend the functionality of existing Windows Forms controls with inheritance. The simplest type of control you can build is one that inherits an existing control. The inherited control includes all the functionality of the original control plus some extra functionality that's specific to an application and that you implement with custom code.

Master It Describe the process of designing an inherited custom control.

Build compound controls that combine multiple existing controls. A compound control provides a visible interface that combines multiple Windows controls. As a result, this type of control doesn't inherit the functionality of any specific control; you must expose its properties by providing your own code. The UserControl object, on which the compound control is based, already exposes a large number of members, including some fairly advanced ones such as the `Anchoring` and `Docking` properties, and the usual mouse and key events.

Master It How will you map certain members of a constituent control to custom members of the compound control?

Build custom controls from scratch. User-drawn controls are the most flexible custom controls, because you're in charge of the control's functionality and appearance. Of course, you have to implement all the functionality of the control from within your code, so it takes substantial programming effort to create user-drawn custom controls.

Master It Describe the process of developing a user-drawn custom control.

Customize the rendering of items in a ListBox control. To create an owner-drawn list control, you must set the `DrawMode` property to a member of the `DrawMode` enumeration and program two events: `MeasureItem` and `DrawItem`.

Master It Outline the process of creating a ListBox control that wraps the contents of lengthy items.

Chapter 13

Handling Strings, Characters, and Dates

This chapter is a formal discussion of the .NET Framework's string- and date-manipulation capabilities. We have used strings extensively in previous chapters, and you already know many of the properties and methods of the String class. Almost every application manipulates strings, so String and StringBuilder are two classes that you'll use more than any other.

Previous versions of Visual Basic provided numerous functions for manipulating strings. These functions are supported by VB 2008, and not just for compatibility reasons; they're part of the core of Visual Basic. The string-manipulation functions are still a major part of Visual Basic.

Another group of functions deals with dates. The date-manipulation functions are also part of the core of the language and were not moved to a special class. Many of these functions are duplicated in the DateTime class, in the form of properties and methods.

In this chapter, you'll learn how to do the following:

- Use the Char data type to handle characters
- Use the String data type to handle strings
- Use the StringBuilder class to manipulate large or dynamic strings
- Use the DateTime and TimeSpan classes to handle dates and times

Handling Strings and Characters

The .NET Framework provides two basic classes for manipulating text: the String and StringBuilder classes.

The String class exposes a large number of practical methods, and they're all *reference methods*: They don't act on the string directly but return another string instead. After you assign a value to a String object, that's it. You can examine the string, locate words in it, and parse it, but you can't edit it. The String class exposes methods such as the `Replace` and `Remove` methods, which replace a section of the string with another and remove a range of characters from the string, respectively. These methods, however, don't act on the string directly: They replace or remove parts of the original string and then return the result as a new string.

The StringBuilder class is similar to the String class: It stores strings, but it can manipulate them in place. In other words, the methods of the StringBuilder class are instance methods.

The distinction between the two classes is that the String class is better suited for static strings, whereas the StringBuilder class is better suited for dynamic strings. Use the String class for strings that don't change frequently in the course of an application, and use the StringBuilder class for strings that grow and shrink dynamically. The two classes expose similar methods, but the String class's methods return new strings; if you need to manipulate large strings extensively, using the String class might fill the memory quite quickly.

Any code that manipulates strings must also be able to manipulate individual characters. The Framework supports the Char class, which not only stores characters but also exposes numerous methods for handling them. Both the String and StringBuilder classes provide methods for storing strings into arrays of characters, as well as for converting character arrays into strings. After extracting the individual characters from a string, you can process them with the members of the Char class. We'll start our discussion of the text-handling features of the Framework with an overview of the Char data type, and we'll continue with the other two major components, the String and StringBuilder classes.

The Char Class

The Char data type stores characters as individual, double-byte (16-bit), Unicode values; and it exposes methods for classifying the character stored in a Char variable. You can use methods such as `IsDigit` and `IsPunctuation` on a Char variable to determine its type, and other similar methods that can simplify your string validation code.

To use a character variable in your application, you must declare it with a statement such as the following one:

```
Dim ch As Char
ch = Convert.ToChar("A")
```

The expression `"A"` represents a string, even if it contains a single character. Everything you enclose in double quotes is a string. To convert it to a character, you must cast it to the Char type. If the `Strict` option is off (which is the default value), you need not perform the conversion explicitly. If the `Strict` option is on, you must use one of the `CChar()` or the `CType()` functions, or the Convert class, to convert the single-character string in the double quotes to a character value, as shown in the preceding statement. There's also a shorthand notation for converting one-character strings to characters — just append the `c` character to a single-character string:

```
Dim ch As Char = "A"c
```

If you let the compiler decipher the type of the variable from its value, a single-character string will be interpreted as a string, not a Char data type. If you later assign a string value to a Char variable by using a statement such as the following, only the first character of the string will be stored in the `ch` variable:

```
ch = "ABC"    ' the value "A" is assigned to ch!
```

Properties

The Char class provides two trivial properties: `MaxValue` and `MinValue`. They return the largest and smallest character values you can represent with the Char data type.

Methods

The Char data type exposes several useful methods for handling characters. All the methods described here have the same syntax: They accept either a single argument, which is the character they act upon, or a string and the index of a character in the string on which they act.

GetNumericValue

This method returns a positive numeric value if called with an argument that is a digit, and the value −1 otherwise. If you call the `GetNumericValue` with the argument `5`, it will return the numeric value 5. If you call it with the symbol `@`, it will return the value −1.

GetUnicodeCategory

This method returns a numeric value that is a member of the `UnicodeCategory` enumeration and identifies the Unicode group to which the character belongs. The Unicode groups characters into categories such as math symbols, currency symbols, and quotation marks. Look up the `UnicodeCategory` enumeration in the documentation for more information.

IsLetter, IsDigit, IsLetterOrDigit

These methods return a True/False value indicating whether their argument, which is a character, is a letter, decimal digit, or letter/digit, respectively. You can write an event handler by using the `IsDigit` method to accept numeric keystrokes and to reject letters and punctuation symbols.

We commonly use these methods to intercept keystrokes from within a control's `KeyPress` (or `KeyUp` and `KeyDown`) events. The `e.KeyChar` property of the `e` argument returns the character that was pressed by the user and that fired the `KeyPress` event. To reject non-numeric keys as the user enters text in a TextBox control, use the event handler shown in Listing 13.1.

Listing 13.1: Rejecting Non-numeric Keystrokes

```
  Private Sub TextBox1_KeyPress(...) _
           Handles TextBox1.KeyPress
    Dim c As Char
    c = e.KeyChar
    If Not (Char.IsDigit(c) or Char.IsControl(c)) Then
      e.Handled = True
    End If
End Sub
```

This code ignores any keystrokes that don't represent numeric digits and are not control characters. Control characters are not rejected, because we want users to be able to edit the text on the control. The Backspace key, for example, is captured by the `KeyPress` event, and you shouldn't ''kill'' it. For more information on handling keystrokes from within your code, see the section ''Capturing Keystrokes'' in Chapter 6, ''Basic Windows Controls.'' If the TextBox control is allowed to accept fractional values, you should allow the period character as well, by using the following `If` clause:

```
Dim c As Char
c = e.KeyChar
If Not (Char.IsDigit(c) or c = "." or _
        Char.IsControl(c)) Then
    e.Handled = True
End If
```

IsLower, IsUpper

These methods return a True/False value indicating whether the specified character is lowercase or uppercase, respectively.

IsNumber

This method returns a True/False value indicating whether the specified character is a number. The `IsNumber` method takes into consideration hexadecimal digits (the characters 0123456789-ABCDEF) in the same way as the `IsDigit` method does for decimal numbers.

IsPunctuation, IsSymbol, IsControl

These methods return a True/False value indicating whether the specified character is a punctuation mark, symbol, or control character, respectively. The Backspace and Esc keys, for example, are ISO (International Organization for Standardization) control characters.

IsSeparator

This method returns a True/False value indicating whether the character is categorized as a separator (space, new-line character, and so on).

IsWhiteSpace

This method returns a True/False value indicating whether the specified character is white space. Any sequence of spaces, tabs, line feeds, and form feeds is considered white space. Use this method along with the `IsPunctuation` method to remove all characters in a string that are not words.

ToLower, ToUpper

These methods convert their argument to a lowercase or uppercase character, respectively, and return it as another character.

ToString

This method converts a character to a string. It returns a single-character string, which you can use with other string-manipulation methods or functions.

The String Class

The String class implements the String data type, which is one of the richest data types in terms of the members it exposes. We have used strings extensively in earlier chapters, but this is a formal discussion of the String data type and all of the functionality it exposes.

To create a new instance of the String class, you simply declare a variable of the String type. You can also initialize it by assigning to the corresponding variable a text value:

```
Dim title As String = "Mastering VB2008"
```

Everything enclosed in double quotes is a string, even if it's the representation of a number.

String objects are immutable: Once created, they can't be modified. The names of some of the methods of the String class may lead you to think that they change the value of the string, but they don't; instead, they return a new string. The `Replace` method, for example, doesn't replace any characters in the original string, but it creates a new string, replaces some characters, and then returns the new string:

```
Dim title As String = "Mastering VB 2008"
Dim newTitle As String
newTitle = title.Replace("VB", "Visual Basic")
' to replace the original string use the statement:
title = title.Replace("VB", "Visual Basic")
```

The `Replace` method, like all other methods of the String class, doesn't operate directly on the string to which it's applied. Instead, it creates a new string and returns it as a new string. You can also use Visual Basic's string-manipulation functions to work with strings. For example, you can replace the string `VB` with `Visual Basic` by using the following statement:

```
newTitle = Replace(title, "VB", "Visual Basic")
```

Like the methods of the String class, the string-manipulation functions don't act on the original string; they return a new string.

If you plan to create and manipulate long strings in your code often, use the StringBuilder class instead, which is extremely fast compared to the String class and VB's string-manipulation functions. This doesn't mean that the String data type is obsolete, of course. The String class exposes many more methods for handling strings (such as locating a smaller string in a larger one, comparing strings, changing individual characters, and so on). The StringBuilder class, on the other hand, is much more efficient when you build long strings bit by bit, when you need to remove part of a string, and so on. To achieve its speed, however, it consumes considerably more memory than the equivalent String variable. The methods of both classes are presented in the following sections.

PROPERTIES

The String class exposes only two properties, the `Length` and `Chars` properties, which return a string's length and its characters, respectively. Both properties are read-only.

Length

The `Length` property returns the number of characters in the string and is read-only. To find out the number of characters in a string variable, use the following statement:

```
chars = myString.Length
```

You can apply the `Length` property to any expression that evaluates to a string. The following statement formats the current date in long date format (this format includes the day's and month's names) and then retrieves the string's length:

```
StrLen = Format(Now(), "dddd, MMMM dd, yyyy").Length
```

The `Format()` function, which formats numbers and dates, is discussed in Chapter 2, ''The Visual Basic 2008 Language.'' The function returns a string, so we can call this expression's `Length` property.

Chars

The `Chars` property is an array of characters that holds all the characters in the string. Use this property to read individual characters from a string based on their location in the string (the index of the first character in the array is zero). The `Chars` array is read-only, and you can't edit a string by setting individual characters.

The loop detailed in Listing 13.2 rejects strings (presumably passwords) that are fewer than six characters long and don't contain a special symbol.

LISTING 13.2: Validating a Password

```
Private Function ValidatePassword( _
          ByVal password As String) As Boolean
 If password.Length < 6 Then
    MsgBox( _
          "The password must be at least 6 characters long")
    Return False
 End If
 Dim i As Integer
 Dim valid As Boolean = False
 For i = 0 To password.Length - 1
    If Not Char.IsLetterOrDigit(password.Chars(i)) Then
       Return True
    End If
 Next
 MsgBox("The password must contain at least one " & _
        "character that is not a letter or a digit.")
 Return False
End Function
```

The code checks the length of the user-supplied string and makes sure that it's at least six characters long. If not, it issues a warning and returns False. Then it starts a loop that scans all the characters in the string. Each character is accessed by its index in the string. If one of them is not a letter or digit — in which case the `IsLetterOrDigit` method will return False — the function terminates and returns True to indicate a valid password. If the loop is exhausted, the `password` argument contains no special symbols, and the function displays another message and returns False.

METHODS

All the functionality of the String class is available through methods, which are described next. They are all shared methods: They act on a string and return a new string with the modified value.

Compare

This method compares two strings and returns a negative value if the first string is less than the second, a positive value if the second string is less than the first, and zero if the two strings are equal. Of course, the simplest method of comparing two strings is to use the comparison operators, as shown here:

```
If name1 < name 2 Then
    ' name1 is alphabetically smaller than name 2
Else If name 1 > name 2 Then
    ' name2 is alphabetically smaller than name 1
Else
    ' name1 is the same as name2
End If
```

The `Compare` method is overloaded, and the first two arguments are always the two strings to be compared. The method's return value is 0 if the two strings are equal, 1 if the first string is smaller than the second, and −1 if the second is smaller than the first. The simplest form of the method accepts two strings as arguments:

```
String.Compare(str1, str2)
```

The following form of the method accepts a third argument, which is a True/False value and determines whether the search will be case-sensitive (if True) or not:

```
String.Compare(str1, str2, case)
```

Another form of the `Compare` method allows you to compare segments of two strings. Its syntax is as follows:

```
String.Compare(str1, index1, str2, index2, length)
```

`index1` and *`index2`* are the starting locations of the segment to be compared in each string. The two segments must have the same length, which is specified by the last argument.

The following statements return the values highlighted below each:

```
Debug.WriteLine(str.Compare("the quick brown fox", _
                        "THE QUICK BROWN FOX"))
-1
Debug.WriteLine(str.Compare("THE QUICK BROWN FOX", _
                        "the quick brown fox"))
1
Debug.WriteLine(str.Compare("THE QUICK BROWN FOX", _
                        "THE QUICK BROWN FOX"))
0
```

If you want to specify a case-sensitive search, append yet another argument and set it to True. The forms of the `Compare` method that perform case-sensitive searches can accept yet another argument, which determines the CultureInfo object to be used in the comparison. (This argument applies to some combination of characters in foreign languages, such as Turkish.)

CompareOrdinal

The `CompareOrdinal` method compares two strings similar to the `Compare` method, but it doesn't take into consideration the current locale. This method returns zero if the two strings are the same, and a positive or negative value if they're different. These values, however, are not 1 and −1; they represent the numeric difference between the Unicode values of the first two characters that are different in the two strings.

Concat

This method concatenates two or more strings (places them one after the other) and forms a new string. The simpler form of the `Concat` method has the following syntax and it is equivalent to the `&` operator:

```
newString = String.Concat(string1, string2)
```

This statement is equivalent to the following:

```
newString = string1 & string2
```

A more-useful form of the same method concatenates a large number of strings stored in an array:

```
newString = String.Concat(strings())
```

To use this form of the method, store all the strings you want to concatenate into a string array and then call the `Concat` method. If you want to separate the individual strings with special delimiters, append them to each individual string before concatenating them. Or you can use the `Join` method discussed later in this section. The `Concat` method simply appends each string to the end of the previous one. If you want to concatenate very long strings or a large number of strings, you should use the StringBuilder class.

Copy

The `Copy` method copies the value of one string variable to another. Notice that the value to be copied must be passed to the method as an argument. The `Copy` method doesn't apply to the current instance of the String class. Most programmers will use the assignment operator and will never bother with the `Copy` method.

EndsWith, StartsWith

These two methods return True if their argument ends or starts with a user-supplied substring. The syntax of these methods is as follows:

```
found = str.EndsWith(string)
found = str.StartsWith(string)
```

These two methods are equivalent to the `Left()` and `Right()` functions, which extract a given number of characters from the left or right end of the string, respectively. The two statements following the declaration of the `name` variable are equivalent:

```
Dim name As String = "Visual Basic.NET"
If Left(name, 3) = "Vis" Then ...
If String.StartsWith("Vis") Then ...
```

Notice that the comparison performed by the `StartsWith` method is case-sensitive. If you don't care about the case, you can convert both the string and the substring to uppercase, as in the following example:

```
If name.ToUpper.StartsWith("VIS") Then ...
```

IndexOf, LastIndexOf

These two methods locate a substring in a larger string. The `IndexOf` method starts searching from the beginning of the string, and the `LastIndexOf` method starts searching from the end of the string. Both methods return an integer, which is the order of the substring's first character in the larger string (the order of the first character is zero).

To locate a string within a larger one, use the following forms of the `IndexOf` method:

```
pos = str.IndexOf(searchString)
pos = str.IndexOf(SearchString, startIndex)
pos = str.IndexOf(SearchString, startIndex, endIndex)
```

The `startIndex` and the `endIndex` arguments delimit the section of the string where the search will take place, and *`pos`* is an integer variable.

The last three overloaded forms of the `IndexOf` method search for an array of characters in the string:

```
str.IndexOf(Char())
str.IndexOf(Char(), startIndex)
str.IndexOf(Char(), startIndex, endIndex)
```

The following statement will return the position of the string `Visual` in the text of the *`TextBox1`* control or will return −1 if the string isn't contained in the text:

```
Dim pos As Integer
pos = TextBox1.IndexOf("Visual")
```

Both methods perform a case-sensitive search, taking into consideration the current locale. To make case-insensitive searches, use uppercase for both the string and the substring. The following statement returns the location of the string `visual` (or `VISUAL`, `Visual`, and even `vISUAL`) within the text of *`TextBox1`*:

```
Dim pos As Integer
pos = TextBox1.Text.ToUpper.IndexOf("VISUAL")
```

The expression `TextBox1.Text` is the text on the control and its type is String. First, we apply the method `ToUpper` to convert the text to uppercase. Then we apply the `IndexOf` method to this string to locate the first instance of the word `VISUAL`.

IndexOfAny

This is an interesting method that accepts as an argument an array of arguments and returns the first occurrence of any of the array's characters in the string. The syntax of the `IndexOfAny` method is

```
Dim pos As Integer = str.IndexOfAny(chars)
```

where *chars* is an array of characters. This method attempts to locate the first instance of any member of the *chars* array in the string. If the character is found, its index is returned. If not, the process is repeated with the second character, and so on until an instance is found or the array has been exhausted. If you want to locate the first delimiter in a string, call the IndexOfAny method with an array such as the following:

```
Dim chars() As Char = {"."c, ","c, ";"c, " "c}
Dim mystring As String = "This is a short sentence"
Debug.WriteLine(mystring.IndexOfAny(chars))
```

When the last statement is executed, the value 4 will be printed in the Output window. This is the location of the first space in the string. Notice that the space delimiter is the last one in the *chars* array.

To locate the first number in a string, pass the nums array to the IndexOfAny method, as shown in the following example:

```
Dim nums() As Char = {"1"c, "2"c, "3"c, "4"c, "5"c, _
                      "6"c, "7"c, "8"c, "9"c, "0"c}
```

Insert

The Insert method inserts one or more characters at a specified location in a string and returns the new string. The syntax of the Insert method is as follows:

```
newString = str.Insert(startIndex, subString)
```

startIndex is the position in the str variable, where the string specified by the second argument will be inserted. The following statement will insert a dash between the second and third characters of the string CA93010.

```
Dim Zip As String = "CA93010"
Dim StateZip As String
StateZip = Zip.Insert(2, "-")
```

The StateZip string variable will become CA-93010 after the execution of these statements.

Join

This method joins two or more strings and returns a single string with a separator between the original strings. Its syntax is the following, where *separator* is the string that will be used as the separator, and *strings* is an array with the strings to be joined:

```
newString = String.Join(separator, strings)
```

If you have an array of many strings and you want to join a few of them, you can specify the index of the first string in the array and the number of strings to be joined by using the following form of the Join method:

```
newString = String.Join( _
            separator, strings, startIndex, count)
```

The following statement will create a full path by joining folder names:

```
Dim path As String
Dim folders(3) As String = {"My Documents", "Business", "Expenses"}
path = String.Join("/", folders)
```

The value of the `path` variable after the execution of these statements will be as follows:

```
My Documents/Business/Expenses
```

Split

Just as you can join strings, you can split a long string into smaller ones by using the `Split` method, whose syntax is the following, where ***`delimiters`*** is an array of characters and *`str`* is the string to be split:

```
strings() = String.Split(delimiters, str)
```

The string is split into sections that are separated by any one of the delimiters specified with the first argument. These strings are returned as an array of strings.

Splitting Strings with Multiple Separators

The `delimiters` array allows you to specify multiple delimiters, which makes it a great tool for isolating words in a text. You can specify all the characters that separate words in text (spaces, tabs, periods, exclamation marks, and so on) as delimiters and pass them along with the text to be parsed to the `Split` method.

The statements in Listing 13.3 isolate the parts of a path, which are delimited by a backslash character.

Listing 13.3: Extracting a Path's Components

```
   Dim path As String = "c:\My Documents\Business\Expenses"
Dim delimiters() As Char = {"\"c}
Dim parts() As String
parts = path.Split(delimiters)
Dim iPart As IEnumerator
iPart = parts.GetEnumerator
While iPart.MoveNext
   Debug.WriteLine(iPart.Current.tostring)
End While
```

If the path ends with a slash, the `Split` method will return an extra empty string. If you want to skip the empty strings, pass an additional argument to the function, which is a member of the `StringSplitOptions` enumeration: *`None`* or *`RemoveEmptyEntries`*.

Notice that the *parts* array is declared without a size. It's a one-dimensional array that will be dimensioned automatically by the `Split` method, according to the number of substrings separated by the specified delimiter(s). The second half of the code iterates through the parts of the path and displays them in the Output window.

If you execute the statements of Listing 13.3 (place them in a button's `Click` event handler and run the program), the following strings will be printed in the Output window:

```
c:
My Documents
Business
Expenses
```

If you add the colon character to the list of delimiters, the first string will be `C` instead of `C:`.

Remove

The `Remove` method removes a given number of characters from a string, starting at a specific location, and returns the result as a new string. Its syntax is the following, where *startIndex* is the index of the first character to be removed in the *str* string variable and *count* is the number of characters to be removed:

```
newSrting = str.Remove(startIndex, count)
```

Replace

This method replaces all instances of a specified character (or substring) in a string with a new one. It creates a new instance of the string, replaces the characters as specified by its arguments, and returns this string. The syntax of this method is

```
newString = str.Replace(oldChar, newChar)
```

where *oldChar* is the character in the *str* variable to be replaced, and *newChar* is the character to replace the occurrences of *oldChar*. You can also specify strings instead of characters as arguments to the `Replace` method. The string after the replacement is returned as the result of the method. The following statements replace all instances of the tab character with a single space. You can change the last statement to replace tabs with a specific number of spaces — usually three, four, or five spaces.

```
Dim txt, newTxt As String
Dim vbTab As String = vbCrLf
txt = "some text        with two tabs"
newTxt = txt.Replace(vbTab, "    ")
```

Use the following statements to replace all instances of `VB 2005` in a string with the substring `VB 2008`:

```
Dim txt, newTxt As String
txt = "Welcome to VB 2005"
newTxt = txt.Replace("VB 2005", "VB 2008")
```

PadLeft, PadRight

These two methods align the string left or right in a specified field and return a fixed-length string with spaces to the right (for right-padded strings) or to the left (for left-padded strings). After the execution of these statements

```
Dim LPString, RPString As String
RPString = "[" & "Mastering VB".PadRight(20) & "]"
LPString = "[" & "Mastering VB".PadLeft(20) & "]"
```

the values of the `LPString` and `RPString` variables are as follows:

```
[Mastering VB        ]
[        Mastering VB]
```

There are eight spaces to the left of the left-padded string and eight spaces to the right of the right-padded string.

Another form of these methods allows you to specify the character to be used in padding the strings with an additional argument.

You can use the padding methods for visual alignment only if you're using a monospaced font such as Courier. These two methods can be used to create text files with rows made up of fields that have a fixed length (a common task in transferring data to legacy applications on mainframes).

The StringBuilder Class

The StringBuilder class stores dynamic strings and exposes methods to manipulate them much faster than the String class. As you will see, the StringBuilder class is extremely fast, but it uses considerably more memory than the string it holds. To use the StringBuilder class in an application, you must import the System.Text namespace (unless you want to fully qualify each instance of the StringBuilder class in your code). Assuming that you have imported the System.Text class in your code module, you can create a new instance of the class via the following statement:

```
Dim txt As New StringBuilder
```

Because the StringBuilder class handles dynamic strings in place, it's good to declare in advance the size of the string you intend to store in the current instance of the class. The default capacity is 16 characters, and it's doubled automatically every time you exceed it. To set the initial capacity of the StringBuilder class, use the `Capacity` property.

To create a new instance of the StringBuilder class, you can call its constructor without any arguments, or pass the initial string as an argument:

```
Dim txt As New StringBuilder("some string")
```

If you can estimate the length of the string you'll store in the variable, you can specify this value by using the following form of the constructor, so that the variable need not be resized continuously as you add characters to it:

```
Dim txt As New StringBuilder(initialCapacity)
```

The size you specify is not a hard limit; the variable might grow longer at runtime, and the StringBuilder will adjust its capacity.

If you want to specify a maximum capacity for your StringBuilder variable, use the following constructor:

```
Dim txt As New StringBuilder ( _
                  intialcapacity, maxCapacity)
```

Finally, you can initialize a new instance of the StringBuilder class by using both an initial and a maximum capacity, as well as its initial value, by using the following form of the constructor:

```
Dim txt As New StringBuilder( _
                  string, intialcapacity, maxCapacity)
```

Properties

You have already seen the two basic properties of the StringBuilder class: the `Capacity` and `MaxCapacity` properties. In addition, the StringBuilder class provides the `Length` and `Chars` properties, which are the same as the corresponding properties of the String class. The `Length` property returns the number of characters in the current instance of the StringBuilder class, and the `Chars` property is an array of characters. Unlike the `Chars` property of the String class, this one is read/write. You can not only read individual characters, but also set them from within your code. The index of the first character is zero.

Methods

Many of the methods of the StringBuilder class are equivalent to the methods of the String class, but they act directly on the string to which they're applied, and they don't return a new string.

Append

The `Append` method appends a base type to the current instance of the StringBuilder class, and its syntax is the following, where the *`value`* argument can be a single character, a string, a date, or any numeric value:

```
SB.Append(value)
```

When you append numeric values to a StringBuilder, they're converted to strings; the value appended is the string returned by the type's `ToString` method. You can also append an object to the StringBuilder — the actual string that will be appended is the value of the object's `ToString` property. Another form of the `Append` method allows you to append an array of characters, and it has the following syntax:

```
SB.Append(chars, startIndex, count)
```

Or, you can append a segment of a string by specifying the starting location of the segment in the string and the number of characters to be copied:

```
SB.Append(string, startIndex, count)
```

AppendFormat

The AppendFormat method is similar to the Append method. Before appending the string, however, AppendFormat formats it. The string to be appended contains format specifications and the appropriate values. The syntax of the AppendFormat method is as follows:

```
SB.AppendFormat(string, values)
```

The first argument is a string with embedded format specifications, and *values* is an array with values (objects, in general) — one for each format specification in the *string* argument. If you have a small number of values to format, up to four, you can supply them as separate arguments separated by commas:

```
SB.AppendFormat(string, value1, value2, value3, value4)
```

The following statement appends the string Your balance as of Thursday, August 2, 2007 is $19,950.40 to a StringBuilder variable:

```
Dim statement As New StringBuilder
statement.AppendFormat( _
       "Your balance as of {0:D} is ${1: #,###.00}", _
       #8/2/2007#, 19950.40)
```

Each format specification is enclosed in a pair of curly brackets, and they're numbered sequentially (from zero). Then there's a colon followed by the actual specification. The D format specification tells the AppendFormat method to format the specified string in long date format. The second format specification, #,###.00, uses the thousands separator and two decimal digits for the amount.

The following statements append the same string, but they pass the values through an array:

```
Dim accountStatement As New StringBuilder
Dim values() As Object = {#8/2/2007#, 19950.4}
accoutnStatement.AppendFormat( _
        "Your balance as of {0:D} is ${1:#,###.00} ", values)
```

In both cases, the accountStatement variable will hold a string like this one:

```
Your balance as of Wednesday,
August 2, 2007 is $19,950.40
```

For more information on date and time formatting options, see the description of the ToString method of the Date type, later in this chapter.

Insert

This method inserts a string into the current instance of the StringBuilder class, and its syntax is as follows:

```
SB.Insert(index, value)
```

The *index* argument is the location where the new string will be inserted in the current instance of the StringBuilder, and *value* is the string to be inserted. As with the Append method, the *value* argument can be any object. The Insert method will insert the string returned by the object's ToString method. This means that you can use the Insert method to insert numeric values and dates directly into a StringBuilder variable.

A variation of the syntax shown here inserts multiple copies of the specified string into the StringBuilder:

```
SB.Insert(index, string, count)
```

Yet another form of the Insert method inserts an array of characters at the location specified by the *index* argument in the current instance of the StringBuilder (*chars* is a properly declared and initialized array of characters):

```
SB.Insert(index, chars)
```

Remove

This method removes a number of characters from the current StringBuilder, starting at a specified location; its syntax is the following, where *startIndex* is the position of the first character to be removed from the string, and *count* is the number of characters to be removed:

```
SB.Remove(startIndex, count)
```

Replace

This method replaces all instances of a string in the current StringBuilder object with another string. The syntax of the Replace method is the following, where the two arguments can be either strings or characters:

```
SB.Replace(oldValue, newValue)
```

Unlike the String class, the replacement takes place in the current instance of the StringBuilder class and the method doesn't return another string. Another form of the Replace method limits the replacements to a specified segment of the StringBuilder instance:

```
SB.Replace(oldValue, newValue, startIndex, count)
```

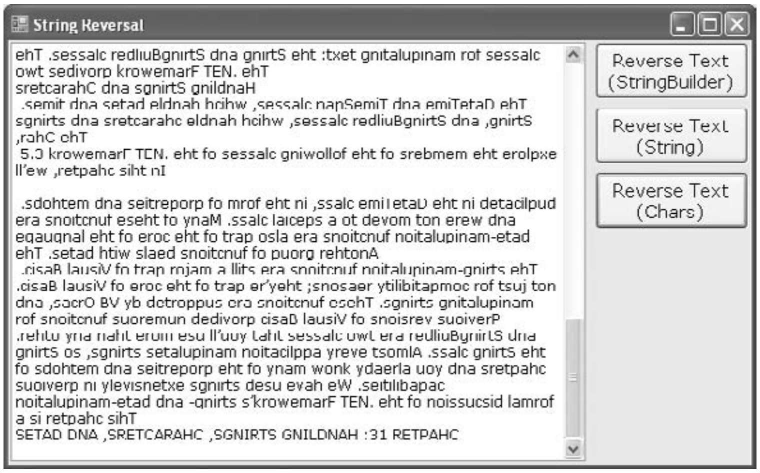
This method will replace all instances of *oldValue* with *newValue* in the section starting at location *startIndex* and extending *count* characters from the starting location.

ToString

Use this method to convert the StringBuilder instance to a string and assign it to a String variable. The ToString method returns the string represented by the StringBuilder variable to which it's applied.

VB 2008 at Work: The StringReversal Project

To get an idea of how efficiently the StringBuilder manipulates strings, Figure 13.1 shows an application that reverses a string. The program reverses two strings — one declared as String, and another one declared as StringBuilder. Note that neither the String nor the StringBuilder class exposes a method for reversing the order of the characters in a string. Actually, the String class exposes a `Reverse` method, which returns a collection of characters — the least useful implementation of a string reversal method, in my humble opinion. However, you can use the `StrReverse()` function in this project to reverse a string with a single function call. In this example, we'll reverse the strings by swapping individual characters to time the two classes.



```
ehT .sessalc redliuBgnirtS dna gnirtS eht :txet gnitalupinam rof sessalc
owt sedivorp krowemarF TEN. ehT
sretcarahC dna sgnirtS gnildnaH
 .semit dna setad eldnah hcihw ,sessalc napSemiT dna emiTetaD ehT
sgnirts dna sretcarahc eldnah hcihw ,sessalc redliuBgnirtS dna ,gnirtS
,rahC ehT
 5.3 krowemarF TEN. eht fo sessalc gniwollof eht fo srebmem eht erolpxe
ll'ew ,retpahc siht nI

 .sdohtem dna seitreporp fo mrof eht ni ,ssalc emiTetaD eht ni detacilpud
era snoitcnuf eseht fo ynaM .ssalc laiceps a ot devom ton erew dna
egaugnal eht fo eroc eht fo trap osla era snoitcnuf noitalupinam-etad
ehT .setad htiw slaed snoitcnuf fo puorg rehtonA
 .cisaB lausiV fo trap rojam a llits era snoitcnuf noitalupinam-gnirts ehT
.cisaB lausiV fo eroc eht fo trap er'yeht ;snosaer ytilibitapmoc rof tsuj ton
dna ,sacrO BV yb detroppus era snoitcnuf esehT .sgnirts gnitalupinam
rof snoitcnuf suoremun dedivorp cisaB lausiV fo snoisrev suoiverP
.rehto yna naht erom esu ll'uoy taht sessalc owt era redliuBgnirtS dna
gnirtS os ,sgnirts setalupinam noitacilppa yreve tsomlA .ssalc gnirtS eht
fo sdohtem dna seitreporp eht fo ynam wonk ydaerla uoy dna sretpahc
suoiverp ni ylevisnetxe sgnirts desu evah eW .seitilibapac
noitalupinam-etad dna -gnirts s'krowemarF TEN. eht fo noissucsid lamrof
a si retpahc sihT
SETAD DNA ,SRETCARAHC ,SGNIRTS GNILDNAH :31 RETPAHC
```

FIGURE 13.1
The StringReversal project's main form

On my computer, it took less than a hundredth of a second to reverse a string of approximately 90,000 characters (the text of this chapter) with the StringBuilder class and nearly 15 seconds to do the same with the String class. Obviously, the StringBuilder class is optimized for manipulating strings dynamically. If you have VB 6 applications that manipulate strings extensively, port them to VB 2008, replace the string variables with instances of the StringBuilder class, and watch them run circles around the old applications written with the String class.

The StringReversal project reads the text on the TextBox control and appends it to the *STR* StringBuilder variable. Then it goes through the first half of the string, one character at a time, and swaps it with the matching character in the second half of the array. The first character in the string, `STR.Chars(0)`, is swapped with the last character, `STR.Chars(STR.Length-1)`. The second character, `STR.Chars(1)`, is swapped with the second-to-last character, `STR.Chars (STR.Length – 2)`, and so on. Notice that we subtract one from the indices because the indexing of the characters in both String and StringBuilder variables starts at zero, and the location of the last character is the length of the string minus one. The code stores the length of the StringBuilder to the *`txtLen`* variable to avoid calling the `Length` property at each iteration.

Listing 13.4 shows the code that reverses a string by using a StringBuilder variable; this is the code behind the Reverse Text (StringBuilder) button. The actual implementation contains a few statements that time the operation, which are not shown in the listing.

LISTING 13.4: Reversing a *StringBuilder* Variable

```
    Dim STR As New StringBuilder()
  Dim txtLen As Integer = TextBox1.Text.Length
  STR.Capacity = txtLen
  STR.Append(TextBox1.Text)
```

```
Dim ichar As Integer
Dim chr As Char
For ichar = 0 To Convert.ToInt32(txtLen / 2 - 1)
    chr = STR.Chars(ichar)
    STR.Chars(ichar) = STR.Chars(txtLen - ichar - 1)
    STR.Chars(txtLen - ichar - 1) = chr
Next
Dim revCrLf As String
revCrLf = vbCrLf.Chars(1) & vbCrLf.Chars(0)
STR.Replace(revCrLf, vbCrLf)
TextBox1.Text = STR.ToString
```

To reverse a string variable, we use the string-manipulation functions of VB, because the String class doesn't provide any methods that act directly on the string (you cannot even manipulate a string through its `Chars` property, because this property is read-only). Unlike the methods of the StringBuilder class, the equivalent VB functions use the index 1 for the first character of the string. The `Mid()` function extracts a character from a string, and the `Mid` statement replaces one of the existing characters with another one. Listing 13.5 is the code of the `Click` event handler of the Reverse Text (String) button. The project contains a few more statements to time the operations, and they're discussed in the following section.

Listing 13.5: Reversing a String Variable

```
  Dim txt As String = TextBox1.Text
Dim txtLen As Integer = txt.Length
Dim aChar As String
For iChar As Integer = 1 To _
          Convert.ToInt32(txtLen / 2))
    aChar = Mid(txt, iChar, 1)
    Mid(txt, iChar, 1) = _
          Mid(txt, txtLen - iChar + 1, 1)
    Mid(txt, txtLen - iChar + 1) = aChar
Next
Dim revCrLf As String = vbCrLf.Chars(1) & _
                        vbCrLf.Chars(0)
txt = txt.Replace(revCrLf, vbCrLf)
TextBox1.Text = txt
```

Notice the statement that replaces carriage returns and line feeds. On the TextBox control, each line is terminated with the sequence `Chr(10)` & `Chr(13)` (the `vbCrLf` constant). When the order of these two characters is reversed, they will no longer change lines on the TextBox control. This statement restores the line-feed/carriage-return combination back to its original state.

To reverse the characters by using the String data type, I've used the straightforward approach that most VB developers would. The string-manipulation functions are so slow compared to the methods of the StringBuilder class, there just had to be a better way. Indeed, you can store the characters that make up the text in an array and then reverse the elements of this array.

The operation is much faster than using string-manipulation functions; it takes almost half a second to reverse a 90,000 character string. However, it takes the StringBuilder class a few milliseconds to do the same. If you're interested in seeing an example of manipulating a string as an array of characters, see the code behind the Reverse Text (Chars) button on the application's form. The core of the string reversal code is the following:

```
Dim chars() As Char = txt.ToCharArray
Dim reversedChars() As Char = chars.ToArray
For iChar As Integer = 0 To _
           Convert.ToInt32(Math.Floor(txtLen / 2)) - 1
    reversedChars.SetValue( _
                    chars(iChar), (txtLen - iChar - 1))
    reversedChars.SetValue( _
                    chars(txtLen - iChar - 1), iChar)
Next
TextBox1.Text = reversedChars
```

VB 2008 at Work: The CountWords Project

The StringBuilder class doesn't provide as many methods as the String class. It's used primarily to build long strings and manipulate them dynamically. If you want to locate words or other patterns in the text, align strings in fixed-length fields, and perform other similar operations, use the String class. We frequently combine both classes in an application: the StringBuilder class for its speed and the String class for its manipulation methods. To extract the text from a StringBuilder, use its `ToString` method. To assign a string to the StringBuilder variable, use its `Append` method:

```
Dim strB As New StringBuilder
Dim str1, str2 As String
str1 = "some text"
strB.Append(str1)
' statements to process the strB variable
str2 = strB.ToString
```

The `ToString` method of the StringBuilder class returns a string, which can be processed with the methods of the String class. For instance, the StringBuilder class lacks the `IndexOf` and `LastIndexOf` methods. To locate an instance of a word in a StringBuilder variable, use the following statement, where *`SB`* is a properly declared and initialized StringBuilder variable and *`pos`* is the index of the first instance of the word *visual* in the StringBuilder's text:

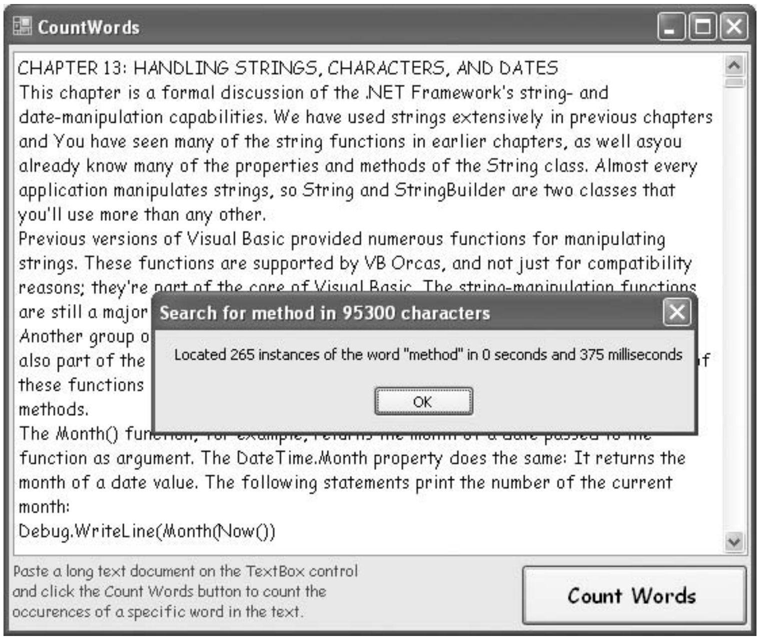
```
pos = SB.ToString.IndexOf("visual")
```

The CountWords application, shown in Figure 13.2, counts all instances of a user-supplied word in a StringBuilder variable. You can do the same with the String class, but if you want to further process the text, you'll have to use the StringBuilder class anyway.

The program prompts the user for a string and attempts to locate it in the text by using the following statement:

```
startIndex = SB.ToString.ToUpper.IndexOf(searchWord.ToUpper)
```

FIGURE 13.2
The CountWords project counts the instances of a user-supplied word in a text.



CHAPTER 13: HANDLING STRINGS, CHARACTERS, AND DATES
This chapter is a formal discussion of the .NET Framework's string- and date-manipulation capabilities. We have used strings extensively in previous chapters and You have seen many of the string functions in earlier chapters, as well asyou already know many of the properties and methods of the String class. Almost every application manipulates strings, so String and StringBuilder are two classes that you'll use more than any other.
Previous versions of Visual Basic provided numerous functions for manipulating strings. These functions are supported by VB Orcas, and not just for compatibility reasons; they're part of the core of Visual Basic. The string-manipulation functions
are still a major
Another group o
also part of the
these functions
methods.
The Month() fu
function as argument. The DateTime.Month property does the same: It returns the month of a date value. The following statements print the number of the current month:
Debug.WriteLine(Month(Now())

Search for method in 95300 characters

Located 265 instances of the word "method" in 0 seconds and 375 milliseconds

Paste a long text document on the TextBox control and click the Count Words button to count the occurences of a specific word in the text.

The preceding statement performs the search operation by using the uppercase of the string and the search argument to avoid mismatches due to the casing of the strings. The sample project sets up a loop that locates one instance of the user-supplied word at a time. The following statement searches for the word in the text, starting at the location *`startIndex`* + *`searchWord.Length`* + *1*. This expression is the location of the first character following the most recently located instance of the search argument in the text. At each iteration of the loop, the `IndexOf` method starts searching for the word in the text following the previous instance of the word. Here's the statement that locates the next instance of the word in the text:

```
startIndex = _
      SB.ToString.ToUpper.IndexOf(searchWord.ToUpper, _
                  startIndex + searchWord.length + 1)
```

This statement appears in a loop that's repeated for as long as the *`startIndex`* variable is positive. When all instances of the word in the text have been located, the `IndexOf` method returns the value −1 and the loop terminates. The complete code of the Count Words button is shown in Listing 13.6.

LISTING 13.6: The CountWords Project's Code

```
   Dim SB As New System.[Text].StringBuilder()
Dim searchWord As String
searchWord = InputBox( _
            "Please enter the word to search for", _
            "StringBuilder Search Example", "BASIC")
```

```
    Dim startIndex As Integer
    SB.Append(Textbox1.Text)
    startIndex = _
          SB.ToString.ToUpper.IndexOf(searchWord.ToUpper)
    Dim count As Integer
    If startIndex = -1 Then
       MsgBox("No instances of the string found")
    End If
    While startIndex >= 0 And startIndex + _
                              searchWord.Length < SB.Length
        count = count + 1
        startIndex = SB.ToString.ToUpper.IndexOf( _
                       searchWord.ToUpper, _
                       startIndex + searchWord.Length + 1)
    End While
    Dim msg As String
    msg = "Located " & count.ToString & _
          " instances of the word " & searchWord & _
          """ in the text"
        MsgBox(msg)
    End Sub
```

When executed, this code will pop up a message box with a statement like the following:

```
Located 22 instances of the word in 270 milliseconds
```

The last few statements calculate the time it took the program to locate all the instances of the word with the methods of the TimeSpan object, which is discussed in the following section.

Handling Dates and Times

Another common task in coding business applications is the manipulation of dates and times. To aid the coding of these tasks, the Framework provides the DateTime and TimeSpan classes. The DateTime class handles date and time *values*, whereas the TimeSpan class handles date and time *differences*. Date is a data type, and there's no equivalent class in the Framework. All date variables are implemented by the DateTime class in the Framework. In effect, the Date data type is an alias for the DateTime data type, and both types are implemented by the System.DateTime class of the Framework.

The DateTime Class

The DateTime class is used for storing date and time values, and it's one of the Framework's base data types. Date and time values are stored internally as Double numbers. The integer part of the value corresponds to the date, and the fractional part corresponds to the time. To convert a DateTime variable to a Double value, use the method `ToOADateTime`, which returns a value that is an OLE (Object Linking and Embedding) Automation-compatible date. The value 0 corresponds to midnight of December 30, 1899. The earliest date you can represent as an OLE

Automation-compatible date corresponds to the Double value is −657,434, and it's the first day of the year 100.

To initialize a DateTime variable, supply a date value enclosed in a pair of pound symbols. If the value contains time information, separate it from the date part by using a space:

```
Dim date1 As Date = #4/15/2007#
Dim date2 As Date = #4/15/2007 2:01:59#
```

If you have a string that represents a date, and you want to assign it to a DateTime variable for further processing, use the DateTime class's `Parse` and `ParseExact` methods. The `Parse` method parses a string and returns a date value if the string can be interpreted as a date value. Let's say your code prompts the user for a date and then it uses it in date calculations. The user-supplied date is read as a string, and you must convert it to a date value:

```
Dim sDate1 As String
Dim dDate1 As DateTime
sDate1 = InputBox("Please enter a date after 1/1/2002")
Try
    dDate1 = DateTime.Parse(sDate1)
        ' use dDate1 in your calculations
    Catch exc As Exception
        MsgBox("You've entered an invalid date")
End Try
```

The `Parse` method will convert a string that represents a date to a DateTime value, regardless of the format of the date. You can enter dates such as *1/17/2007*, *Jan. 17, 2007*, or *January 17, 2007* (with or without the comma). The `ParseExact` method allows you to specify more options, such as the possible formats of the date value.

Real World Scenario

DIFFERENT CULTURES, DIFFERENT DATES

Different cultures use different date formats, and Windows supports them all. However, you must make sure that the proper format is selected in the Regional And Language Options. By default, dates are interpreted as specified by the current date format in the target computer's regional settings. The `Parse` method allows you to specify the culture to be used in the conversion. The following statements prompt the user for a date value and then interpret it in a specific culture (I'm using the English date format for the example):

```
Dim sDate1 As String
Dim dDate1 As DateTime
sDate1 = InputBox("Please enter a date")
Try
    Dim culture As CultureInfo = _
                New CultureInfo("en-GB", True)
    dDate1 = DateTime.Parse(sDate1, culture)
    Debug.WriteLine(dDate1.ToLongDateString)
Catch exc As Exception
    MsgBox("You've entered an invalid date")
End Try
```

To use the CultureInfo class in your code, you must import the System.Gobalization namespace in your project. These statements will convert any English date regardless of the regional settings. If you enter the string 16/3/2007 in the input box, the preceding statements will produce the following output:

```
Friday, March 16, 2007
```

Let's see how the same date will be parsed in two different cultures. Insert the following code segment in a button's Click event handler:

```
Dim sDate1 As String
Dim dDate1 As DateTime
sDate1 = InputBox("Please enter a date")
Try
    Dim culture As CultureInfo = _
                   New CultureInfo("en-GB", True)
    dDate1 = DateTime.Parse(sDate1, culture)
    Debug.WriteLine(dDate1.ToLongDateString)
    culture = New CultureInfo("en-US", True)
    dDate1 = DateTime.Parse(sDate1, culture)
    Debug.WriteLine(dDate1.ToLongDateString)
Catch exc As Exception
    MsgBox("You've entered an invalid date")
End Try
```

The method ToLongDateString returns the verbose description of the date, so that we can read the name of the month instead of guessing it. Run the code and enter a date that can be interpreted differently in the two cultures, such as 4/9/2007. The following output will be produced:

```
Tuesday, September 04, 2007
Monday, April 09, 2007
```

If the month part of the date exceeds 12, the exception handler will be activated. Dates are always a tricky issue in programming, and you should include the appropriate culture in the Parse method so that user-supplied dates will be converted correctly, even if the user's culture hasn't been set correctly in the regional settings.

Properties

The DateTime class exposes the following properties, which are straightforward.

Date, TimeOfDay

The `Date` property returns the date from a date/time value and sets the time to midnight. The `TimeOfDay` property returns the time part of the date. The following statements

```
Dim date1 As DateTime
date1 = Now()
Debug.WriteLine(date1)
Debug.WriteLine(date1.Date)
Debug.WriteLine(date1.TimeOfDay)
```

will print something like the following values in the Output window:

```
8/5/2007 9:41:55 AM
8/5/2007 12:00:00 AM
09:41:55.5296000
```

DayOfWeek, DayOfYear

These two properties return the day of the week (a string such as `Monday`) and the number of the day in the year (an integer from 1 to 365, or 366 for leap years), respectively.

Hour, Minute, Second, Millisecond

These properties return the corresponding time part of the date value passed as an argument. If the current time is 9:47:24 p.m., the three properties of the DateTime class will return the integer values 9, 47, and 24 when applied to the current date and time:

```
Debug.WriteLine("The current time is " & Date.Now.ToString)
Debug.WriteLine("The hour is " & Date.Now.Hour)
Debug.WriteLine("The minute is " & Date.Now.Minute)
Debug.WriteLine("The second is " & Date.Now.Second)
```

Day, Month, Year

These three properties return the day of the month, the month, and the year of a DateTime value, respectively. The `Day` and `Month` properties are numeric values, but you can convert them to the appropriate string (the name of the day or month) with the `WeekDayName()` and `MonthName()` functions. Both functions accept as an argument the number of the day (a value from 1 to 7) or month (from 1 to 13), and they return the name. Use the value 13 with a 13-month calendar (applies to non-U.S. and non-European calendars). They also accept a second optional argument that is a True/False value and indicates whether the function should return the abbreviated name (if True) or full name (if False). The `WeekDayName()` function accepts a third optional argument, which determines the first day of the week. Set this argument to one of the members of the `FirstDayOfWeek` enumeration. By default, the first day of the week is Sunday.

Ticks

This property returns the number of ticks from a date/time value. Each tick is 100 nanoseconds (or 0.0001 milliseconds). To convert ticks to milliseconds, multiply them by 10,000 (or use the TimeSpan object's `TicksPerMillisecond` property, discussed later in this chapter). We use this property to time operations precisely: The `Ticks` property is a long value, and you can read its value before and after the operation you want to time. The difference between the two values is the duration of the operation in tenths of nanoseconds. Divide it by 10,000 to get the duration in milliseconds.

Methods

The DateTime class exposes several methods for manipulating dates. The most practical methods add and subtract time intervals to and from an instance of the DateTime class.

Compare

`Compare` is a shared method that compares two date/time values and returns an integer value indicating the relative order of the two values. The syntax of the `Compare` method is the following, where *`date1`* and *`date2`* are the two values to be compared:

```
order = System.DateTime.Compare(date1, date2)
```

The method returns an integer, which is −1 if *`date1`* is less than *`date2`*, 0 if they're equal, and 1 if *`date1`* is greater than *`date2`*. Of course, you can compare dates directly with a statement such as this one:

```
If date1 > date2 Then ...
```

When comparing dates, keep in mind that older dates are smaller than newer dates. The preceding comparison is true if *`date1`* follows *`date2`*. You can also use the `Subtract` method to find out the difference between two dates in days, weeks, months, and years.

DaysInMonth

This shared method returns the number of days in a specific month. Because February contains a variable number of days depending on the year, the `DaysInMonth` method accepts as arguments both the month and the year:

```
monDays = DateTime.DaysInMonth(year, month)
```

FromOADate

This shared method creates a date/time value from an OLE Automation-compatible date.

```
newDate = DateTime.FromOADate(dtvalue)
```

The argument *`dtvalue`* must be a Double value in the range from −657,434 (first day of year 100) to 2,958,465 (last day of year 9999).

IsLeapYear

This shared method returns a True/False value that indicates whether the specified year is a leap year:

```
Dim leapYear As Boolean = DateTime.IsLeapYear(year)
```

Add

This method adds a TimeSpan object to the current instance of the DateTime class. The TimeSpan object represents a time interval, and there are many methods to create a TimeSpan object, which are all discussed in the section ''The TimeSpan Class'' later in this chapter. The following statements create a new TimeSpan object that represents 3 days, 6 hours, 2 minutes, and 50 seconds and add this TimeSpan object to the current date and time. Depending on when these statements are executed, the two date/time values will differ, but the difference between them will always be 3 days, 6 hours, 2 minutes, and 50 seconds:

```
Dim TS As New TimeSpan()
Dim thisMoment As Date = Now()
TS = New TimeSpan(3, 6, 2, 50)
Debug.WriteLine(thisMoment)
Debug.WriteLine(thisMoment.Add(TS))
```

The values printed in the Output window when I tested this code segment were as follows:

```
9/1/2007 10:10:49 AM
9/4/2007 4:13:39 PM
```

Subtract

This method is the counterpart of the `Add` method; it subtracts a TimeSpan object from the current instance of the DateTime class and returns another Date value.

Adding Intervals to Dates

Various methods add specific intervals to a date/time value. Each method accepts the number of intervals to add (days, hours, milliseconds, and so on) to the current instance of the DateTime class. These methods are the following: `AddYears`, `AddMonths`, `AddDays`, `AddHours`, `AddMinutes`, `AddSeconds`, `AddMilliseconds`, and `AddTicks`. As stated earlier, a tick is 100 nanoseconds and is used for really fine timing of operations. None of the `Add` *`xxx`* methods act on the current instance of the DateTime class; instead, they return a new DateTime value with the appropriate value.

To add 3 years and 12 hours to the current date, use the following statements:

```
Dim aDate As Date
aDate = Now()
aDate = aDate.AddYears(3)
aDate = aDate.AddHours(12)
```

If the argument is a negative value, the corresponding intervals are subtracted from the current instance of the class.

ToString

This method converts a date/time value to a string, using a specific format. The DateTime class recognizes numerous format patterns, which are listed in the following two tables. Table 13.1 lists the standard format patterns, and Table 13.2 lists the characters that can format individual parts of the date/time value. You can combine the custom format characters to format dates and times in any way you wish.

The syntax of the `ToString` method is the following, where *`formatSpec`* is a format specification:

```
aDate.ToString(formatSpec)
```

The `D` named date format, for example, formats a date value as a long date; the following statement will return the highlighted string shown below the statement:

```
Debug.Writeline(#9/17/2010#.ToString("D"))
Friday, September 17, 2010
```

Table 13.1 lists the named formats for the standard date and time patterns. The format characters are case-sensitive — for example, `g` and `G` represent slightly different patterns.

TABLE 13.1: The Date and Time Named Formats

NAMED FORMAT	OUTPUT	FORMAT NAME
`d`	MM/dd/yyyy	`ShortDatePattern`
`D`	dddd, MMMM dd, yyyy	`LongDatePattern`
`F`	dddd, MMMM dd, yyyy HH:mm:ss.mmm	`FullDateTimePattern` (long date and long time)
`f`	dddd, MMMM dd, yyyy HH:mm.ss	`FullDateTimePattern` (long date and short time)
`g`	MM/dd/yyyy HH:mm	`general` (short date and short time)
`G`	MM/dd/yyyy HH:mm:ss	`General` (short date and long time)
`M, m`	MMMM dd	`MonthDayPattern` (month and day)
`r, R`	ddd, dd MMM yyyy HH:mm:ss GMT	`RFC1123Pattern`
`s`	yyyy-MM-dd HH:mm:ss	`SortableDateTimePattern`
`t`	HH:mm	`ShortTimePattern` (short time)
`T`	HH:mm:ss	`LongTimePattern` (long time)
`u`	yyyy-MM-dd HH:mm:ss	`UniversalSortableDateTimePattern` (sortable GMT value)
`U`	dddd, MMMM dd, yyyy HH:mm:ss	`UniversalSortableDateTimePattern` (long date, long GMT time)
`Y, y`	MMMM, yyyy	`YearMonthPattern` (month and year)

TABLE 13.2: Date Format Specifier

FORMAT CHARACTER	DESCRIPTION
`d`	The date of the month
`ddd`	The day of the month with a leading zero for single-digit days
`ddd`	The abbreviated name of the day of the week (a member of the `AbbreviatedDayNames` enumeration)
`dddd`	The full name of the day of the week (a member of the `DayNamesFormat` enumeration)
`M`	The number of the month
`MM`	The number of the month with a leading zero for single-digit months
`MMM`	The abbreviated name of the month (a member of the `AbbreviatedMonthNames` enumeration)
`MMMM`	The full name of the month
`y`	The year without the century (the year 2001 will be printed as 1)
`yy`	The year without the century (the year 2001 will be displayed as 01)
`yyyy`	The complete year
`gg`	The period or era (this pattern is ignored if the date to be formatted does not have an associated period, such as A.D. or B.C.)
`h`	The hour in 12-hour format
`hh`	The hour in 12-hour format with a leading zero for single-digit hours
`H`	The hour in 24-hour format
`HH`	The hour in 24-hour format with a leading zero for single-digit hours
`m`	The minute of the hour
`mm`	The minute of the hour with a leading zero for single-digit minutes
`s`	The second of the hour
`ss`	The second of the hour with a leading zero for single-digit seconds
`t`	The first character in the a.m./p.m. designator
`tt`	The a.m./p.m. designator
`z`	The time-zone offset (applies to hours only)
`zz`	The time-zone offset with a leading zero for single-digit hours (applies to hours only)
`zzz`	The full time-zone offset (hour and minutes) with leading zeros for single-digit hours and minutes

The following examples format the current date by using all the format patterns listed in Table 13.1. An example of the output produced by each statement is shown under each statement, indented and highlighted.

```
Debug.WriteLine(now().ToString("d"))
   6/1/2008
Debug.WriteLine(now().ToString("D"))
   Sunday, June 01, 2008
Debug.WriteLine(now().ToString("f"))
   Sunday, June 01, 2008 10:29 AM
Debug.WriteLine(now().ToString("F"))
   Sunday, June 01, 2008 10:29:35 AM
Debug.WriteLine(now().ToString("g"))
   6/1/2008 10:29 AM
Debug.WriteLine(now().ToString("G"))
   6/1/2008 10:29:35 AM
Debug.WriteLine(now().ToString("m"))
   June 01
Debug.WriteLine(now().ToString("r"))
   Sun, 01 Jun 2008 10:29:34 GMT
Debug.WriteLine(now().ToString("s"))
   2008-06-01T10:29:35
Debug.WriteLine(now().ToString("t"))
   10:29 AM
Debug.WriteLine(now().ToString("T"))
   10:29:00 AM
Debug.WriteLine(now().ToString("u"))
   2008-06-01 10:29:35Z
Debug.WriteLine(now().ToString("U"))
   Sunday, June 01, 2008 7:29:35 AM
Debug.WriteLine(now().Format("y"))
   June, 2008
```

Table 13.2 lists the format characters that can be combined to build custom format date and time values. The patterns are case-sensitive. If the custom pattern contains spaces or characters enclosed in single quotation marks, these characters will appear in the formatted string.

The following examples format the current time by using a few of the format patterns listed in Table 13.2. The output produced by each statement is shown under each statement, indented and highlighted.

```
Debug.WriteLine(now().ToString("m/d/yyyy"))
   6/1/2008
Debug.WriteLine(now().ToString("dddd"))
   Sunday
Debug.WriteLine(now().ToString("M/yyyy"))
   6/2008
Debug.WriteLine(now().ToString("MMM"))
```

```
   Jun
Debug.WriteLine(now().ToString("MMMM"))
   June
Debug.WriteLine(now().ToString("yyyy"))
   2008
Debug.WriteLine(now().ToString("HH"))
   10
Debug.WriteLine(now().ToString("h:m:s"))
   10:48:41
Debug.WriteLine(now().ToString("hh:mm:ss"))
   10:48:41
Debug.WriteLine(now().ToString("h:m:s t"))
   10:48:41 A
Debug.WriteLine(now().ToString("zzz"))
   +03:00
```

To display the full month name and the day in the month, for instance, use the following statement:

```
Debug.WriteLine(now().ToString("MMMM d"))
July 27
```

You may have noticed some overlap between the named formats and the format characters. The character `d` signifies the short date pattern when used as a named format, and the number of the day when used as format character. The compiler figures out how it's used based on the context. If the format argument is `d/mm`, it will display the day and month number, whereas the format argument `d, mmm` will display the number of the day followed by the month's name. If you use the character `d` on its own, however, it will be interpreted as the named format for the short date format.

Date Conversion Methods

The DateTime class supports methods for converting a date/time value to many of the other base types, which are presented here briefly.

ToFileTime, FromFileTime

The `ToFileTime` method converts the value of the current Date instance to the format of the local system file time. There's also an equivalent `FromFileTime` method, which converts a file time value to a Date value.

ToLongDateString, ToShortDateString

These two methods convert the date part of the current DateTime instance to a string with the long (or short) date format. The following statement will return a value like the one highlighted, which is the long date format:

```
Debug.WriteLine(Now().ToLongDateString)
Tuesday, July 15, 2008
```

ToLongTimeString, ToShortTimeString

These two methods convert the time part of the current instance of the Date class to a string with the long (or short) time format. The following statement will return a value like the one highlighted:

```
Debug.WriteLine(Now().ToLongTimeString)
6:40:53 PM
```

ToOADate

This method converts the DateTime instance into an OLE Automation-compatible date (a long value).

ToUniversalTime, ToLocalTime

`ToUniversalTime` converts the current instance of the DateTime class into universal coordinated time (UCT). If you convert the local time of a system in New York to UCT, the value returned by this method will be a date/time value that's five hours ahead. The date may be the same or the date of the following day. If the statement is executed after 7 p.m. local time, the date will be that of the following day. The method `ToLocalTime` converts a UCT time value to local time.

Dates as Numeric Values

The Date type encapsulates complicated operations, and it's worth taking a look at the inner workings of the classes that handle dates and times. Let's declare two variables to experiment a little with dates: a `Date` variable, which is initialized to the current date, and a `Double` variable.

```
Dim Date1 As Date = Now()
Dim dbl As Double
```

Insert a couple of statements to convert the date to a Double value and print it:

```
dbl = Date1.ToOADate
Debug.WriteLine(dbl)
```

On the date I tested this code, June 1, 2007, the value was 39234.6418796643. The integer part of this value is the date, and the fractional part is the time. If you add one day to the current date and then convert it to a double again, you'll get a different value:

```
dbl = (Now().AddDays(1)).ToOADate
Debug.WriteLine(dbl)
```

This time, the value 39235.6426065857 was printed; its integer part is tomorrow's value. You can add two days to the current date by adding (48 × 60) minutes. The original integer part of the numeric value will be increased by two:

```
dbl = Now().AddMinutes(48 * 60).ToOADate
Debug.WriteLine(dbl)
```

The value printed this time will be 39236.6435319329.

Let's see how the date-manipulation methods deal with leap years. We'll add 10 years to the current date via the `AddYears` method, and we'll print the new value with a single statement:

```
Debug.WriteLine(Now().AddYears(10).ToLongDateString)
```

The value that will appear in the Immediate window will be Thursday, June 01, 2017. The Double value of this date is 42887.6470126852. If you add 3,650 days, you'll get a different value because the 10-year span contains at least two leap years:

```
Debug.WriteLine(Now().AddDays(3650).ToLongDateString)
Debug.WriteLine(Now().AddDays(3650).ToOADate)
```

The new value that will be printed in the Immediate window will be `Monday, May 29, 2004`, and the corresponding Double value will be `42884.6450151968`.

Can you figure out what time it was when I executed the preceding statements? If you multiply the fractional part (0.6426065857) by 24, you'll get 15.4225580568, which is 15:00 hours and some minutes. If you multiply the fractional part of this number by 60, you'll get 25.353483408, which is 25 minutes and some seconds. Finally, you can multiply the new fractional part by 60 to get the number of seconds: 21.20900448. So, it was 3:25:21 p.m. And the last fractional part corresponds to 209 milliseconds.

The TimeSpan Class

The last class discussed in this chapter is the TimeSpan class, which represents a time interval and can be expressed in many different units — from ticks and milliseconds to days. The TimeSpan is usually the difference between two date/time values, but you can also create a TimeSpan for a specific interval and use it in your calculations.

To use the TimeSpan variable in your code, just declare it with a statement such as the following:

```
Dim TS As New TimeSpan
```

To initialize the TimeSpan object, you can provide the number of days, hours, minutes, seconds, and milliseconds that make up the time interval. The following statement initializes a TimeSpan object to a duration of 9 days, 12 hours, 1 minute, and 59 seconds:

```
Dim TS As TimeSpan = New TimeSpan(9, 12, 1, 59)
```

As you have seen, the difference between two dates calculated by the `Date.Subtract` method returns a TimeSpan value. You can initialize an instance of the TimeSpan object by creating two date/time values and getting their difference, as in the following statements:

```
Dim TS As New TimeSpan
Dim date1 As Date = #4/11/1985#
Dim date2 As Date = Now()
TS = date2.Subtract(date1)
Debug.WriteLine(TS)
```

Depending on the day on which you execute these statements, they will print something like the following in the Output window:

```
8086.15:37:01.6336000
```

The days are separated from the rest of the string with a period, whereas the time parts are separated with colons. Notice that a TimeSpan object might represent an interval of many years, but it doesn't provide members to report months or years. The difference represented by this value is 8,086 days, 15 hours, 37 minutes, 1 second, and 633,600 nanoseconds (or 633.6 milliseconds).

Properties

The TimeSpan type exposes the properties described in the following sections. Most of these properties are shared.

Field Properties

TimeSpan exposes the simple properties shown in Table 13.3, which are known as *fields* and are all shared.

Table 13.3: The Fields of the TimeSpan Object

Property	Returns
`Empty`	An empty TimeSpan object
`MaxValue`	The largest interval you can represent with a TimeSpan object
`MinValue`	The smallest interval you can represent with a TimeSpan object
`TicksPerDay`	The number of ticks in a day
`TicksPerHour`	The number of ticks in an hour
`TicksPerMillisecond`	The number of ticks in a millisecond
`TicksPerMinute`	The number of ticks in one minute
`TicksPerSecond`	The number of ticks in one second
`Zero`	A TimeSpan object of zero duration

Interval Properties

In addition to the fields, the TimeSpan class exposes two more groups of properties that return the various intervals in a TimeSpan value (shown in Tables 13.4 and 13.5). The members of the first group of properties return the number of specific intervals (days, hours, and so on) in a TimeSpan value. The second group of properties returns the entire TimeSpan's duration in one of the intervals recognized by the TimeSpan method.

TABLE 13.4: The Intervals of a TimeSpan Value

PROPERTY	RETURNS
`Days`	The number of whole days in the current TimeSpan.
`Hours`	The number of whole hours in the current TimeSpan.
`Milliseconds`	The number of whole milliseconds in the current TimeSpan. The largest value of this property is 999.
`Minutes`	The number of whole minutes in the current TimeSpan. The largest value of this property is 59.
`Seconds`	The number of whole seconds in the current TimeSpan. The largest value of this property is 59.
`Ticks`	The number of whole ticks in the current TimeSpan.

TABLE 13.5: The Total Intervals of a TimeSpan Value

PROPERTY	RETURNS
`TotalDays`	The number of days in the current TimeSpan
`TotalHours`	The number of hours in the current TimeSpan
`TotalMilliseconds`	The number of whole milliseconds in the current TimeSpan
`TotalMinutes`	The number of whole minutes in the current TimeSpan
`TotalSeconds`	The number of whole seconds in the current TimeSpan

If a TimeSpan value represents 2 minutes and 10 seconds, the `Seconds` property will return the value 10. The `TotalSeconds` property, however, will return the value 130, which is the total duration of the TimeSpan in seconds.

SIMILAR METHOD NAMES, DIFFERENT RESULTS

Be very careful when choosing the property to express the duration of a TimeSpan in a specific interval. The `Seconds` property is totally different from the `TotalSeconds` property. Because both properties will return a value, you may not notice that you're using the wrong property for the task at hand.

Duration

This property returns the duration of the current instance of the TimeSpan class. The duration is expressed as the number of days followed by the number of hours, minutes, seconds, and milliseconds. The following statements create a TimeSpan object of a few seconds (or minutes, if you don't mind waiting) and print its duration in the Output window. The first few statements initialize a new instance of the DateTime type, the *T1* variable, to the current date and time. Then a message box is displayed that prompts to click the OK button to continue. Wait for several seconds before closing the message box. The last group of statements subtracts the *T1* variable from the current time and displays the duration (how long you kept the message box open on your screen):

```
Dim T1, T2 As DateTime
T1 = Now
MsgBox("Click OK to continue")
T2 = Now
Dim TS As TimeSpan
TS = T2.Subtract(T1)
Debug.WriteLine("Total duration = " & TS.Duration.ToString)
Debug.WriteLine("Minutes = " & TS.Minutes.ToString)
Debug.WriteLine("Seconds = " & TS.Seconds.ToString)
Debug.WriteLine("Ticks = " & TS.Ticks.ToString)
Debug.WriteLine("Milliseconds = " & TS.TotalMilliseconds.ToString)
Debug.WriteLine("Total seconds = " & TS.TotalSeconds.ToString)
```

If you place these statements in a button's `Click` event handler and execute them, you'll see a series of values like the following in the Immediate window:

```
Total duration = 00:01:34.2154752
Minutes = 1
Seconds = 34
Ticks = 942154752
Milliseconds = 94215,4752
Total seconds = 94,2154752
```

The duration of the *TS* TimeSpan is 1 minute and 34 seconds. Its total duration in milliseconds is 94,215.4752, or 94.2154752 seconds.

Methods

There are various methods for creating and manipulating instances of the TimeSpan class, and they're described in the following sections.

Interval Methods

The methods in Table 13.6 create a new TimeSpan object of a specific duration. The TimeSpan's duration is specified as a number of intervals, accurate to the nearest millisecond.

All methods accept a single argument, which is a Double value that represents the number of the corresponding intervals (days, hours, and so on).

Parse(string)

This method creates a new TimeSpan object from a string with the TimeSpan format (days; followed by a period; followed by the hours, minutes, and seconds separated by colons). The following statements create a new TimeSpan variable with a duration of 3 days, 12 hours, 20 minutes, 30 seconds, and 500 milliseconds:

```
Dim SP As New TimeSpan()
SP = TimeSpan.Parse("3.12:20:30.500")
Debug.WriteLine(SP)
3.12:20:30.5000000
```

Table 13.6: Interval Methods of the TimeSpan Object

Method	Creates a New TimeSpan of This Length
`FromDays`	Number of days specified by the argument
`FromHours`	Number of hours specified by the argument
`FromMinutes`	Number of minutes specified by the argument
`FromSeconds`	Number of seconds specified by the argument
`FromMilliseconds`	Number of milliseconds specified by the argument
`FromTicks`	Number of ticks specified by the argument

Add

This method adds a TimeSpan object to the current instance of the class; its syntax is the following, where TS, TS1, and `newTS` are all TimeSpan variables:

```
newTS = TS.Add(TS1)
```

The following statements create two TimeSpan objects and then add them:

```
Dim TS1 As New TimeSpan = "1:00:01"
Dim TS2 As New TimeSpan = "2:01:09"
Dim TS As New TimeSpan
TS = TS1.Add(TS2)
```

The duration of the new TimeSpan variable is 3 hours, 1 minute, and 10 seconds. A more-practical example is the following, which constructs a TimeSpan object by using the `From` *xxx* methods described in the preceding section. The following statements create a TimeSpan object with a duration of 3 hours, 2 minutes, 16 seconds, and 500 milliseconds:

```
Dim TS As New system.TimeSpan()
TS = System.TimeSpan.FromHours(3)
TS = TS.Add(System.TimeSpan.FromMinutes(2))
TS = TS.Add(TimeSpan.FromSeconds(16))
TS = TS.Add(TimeSpan.FromMilliseconds(500))
```

Subtract

The `Subtract` method subtracts a TimeSpan object from the current instance of the TimeSpan class. The following statements create two TimeSpan objects with different durations. Then, the two time spans are subtracted and their difference is printed in three different ways:

```
Dim T1, T2 As TimeSpan
T1 = New TimeSpan(3, 9, 10, 12)
T2 = New TimeSpan(0, 1, 0, 59, 3)
Dim TS As TimeSpan = T2.Subtract(T1)
Debug.WriteLine(TS.Duration())
Debug.WriteLine(TS.Days)
Debug.WriteLine(TS.TotalDays)
```

The last three statements printed the following values in the Output window:

```
3.08:09:12.9970000
-3
-3.33973376157407
```

The duration of the span is 3 days, 8 hours, 9 minutes, 12 seconds, and 997 milliseconds. The `Days` method returns the number of days in the TimeSpan as a whole number. The `TotalDays` method returns the TimeSpan's duration as a number of days with a fractional part. You have already seen how to manipulate fractional parts of a time interval in the section ''Dates as Numeric Values,'' earlier in this chapter.

Negate

This method negates the current TimeSpan instance. A positive TimeSpan (which will yield a future date when added to the current date) becomes negative (which will yield a past date when added to the current date).

The Bottom Line

Use the Char data type to handle characters. The Char data type, which is implemented with the Char class, exposes methods for handling individual characters (`IsLetter`, `IsDigit`, `IsSymbol`, and so on). We use the methods of the Char class to manipulate users' keystrokes as they happen in certain controls (mostly the TextBox control) and to provide immediate feedback.

Master It You want to develop an interface that contains several TextBox controls that accept numeric data. How will you intercept the user's keystrokes and reject any characters that are not numeric?

Use the String data type to handle strings. The String data type represents strings and exposes members for manipulating them. Most of the String class's methods are equivalent to the string-manipulation methods of Visual Basic. The members of the String class are shared: they do not modify the string to which they're applied. Instead, they return a new string.

Master It How would you extract the individual words from a large text document?

Use the StringBuilder class to manipulate large or dynamic strings. The StringBuilder class is very efficient at manipulating long strings, but it doesn't provide as many methods for handling strings. The StringBuilder class provides a few methods to insert, delete, and replace characters within a string. Unlike the equivalent methods of the String class, these methods act directly on the string stored in the current instance of the String-Builder class.

Master It Assuming that you have populated a ListView control with thousands of lines of data from a database, how will you implement a function that copies all the data to the Clipboard?

Use the DateTime and TimeSpan classes to handle dates and times. The Date class represents dates and time, and it exposes many useful shared methods (such as the `IsLeap` method, which returns True if the year passed to the method as an argument is leap; the `DaysInMonth` method; and so on). It also exposes many instance methods (such as `AddYears`, `AddDays`, `AddHours`, and so on) for adding time intervals to the current instance of the Date class, as well as many options for formatting date and time values.

The TimeSpan class represents time intervals — from milliseconds to days — with the `FromDays`, `FromHours`, and even `FromMilliseconds` methods. The difference between two date variables is a TimeSpan value, and you can convert this value to various time units by using methods such as `TotalDays`, `TotalHours`, `TotalMilliseconds`, and so on. You can also add a TimeSpan object to a date variable to obtain another date variable.

Master It How will you use the TimeSpan class to accurately time an operation?

Chapter 14

Storing Data in Collections

One of the most common operations in programming is the storage of large sets of data. There are databases, of course, which can store any type of data and preserve their structure as well, but not all applications use databases. If your application needs to store custom objects, such as the ones you designed in Chapter 10, ''Building Custom Classes,'' or a few names and contact information, you shouldn't have to set up a database. A simple collection like the ones described in this chapter will suffice.

Traditionally, programmers used arrays to store related data. Because arrays can store custom data types, they seem to be the answer to many data-storage and data-manipulation issues. Arrays, however, don't expose all the functionality you might need in your application. To address the issues of data storage outside databases, the Framework provides, in addition to arrays, of course, certain classes known as collections.

In this chapter, you'll learn how to do the following:

- Make the most of arrays
- Store data in specialized collections such as ArrayLists and HashTables
- Sort and search collections

Advanced Array Topics

Arrays are indexed sets of data, and this is how we've used them so far in this book. In this chapter, you will learn about additional members that make arrays extremely flexible. The System.Array class provides methods for sorting arrays, searching for an element, and more. In the past, programmers spent endless hours writing code to perform the same operations on arrays, but the Framework frees them from similar counterproductive tasks.

This chapter starts with a discussion of the advanced features of the Array class. After you know how to make the most of arrays, I'll discuss the limitations of arrays and then move on to other collections that overcome these limitations.

Sorting Arrays

To sort an array, call its `Sort` method. This method is heavily overloaded and, as you will see, it is possible to sort an array based on the values of another array, or even supply your own custom sorting routines. If the array is sorted, you can call the `BinarySearch` method to locate an element very efficiently. If not, you can still locate an element in the array by using the `IndexOf` and `LastIndexOf` methods. The `Sort` method is a reference method: It requires that you supply the name of the array to be sorted as an argument. The simplest form of the Sort method accepts a single argument, which is the name of the array to be sorted:

```
Array.Sort(arrayName)
```

This method sorts the elements of the array according to the type of its elements, as long as the array is strictly typed and was declared as a simple data type (String, Decimal, Date, and so on). If the array contains data that are not of the same type, or they're objects, the Sort method will fail. The Array class just doesn't know how to compare integers to strings or dates, so don't attempt to sort arrays whose elements are not of the same type. If you can't be sure that all elements are of the same type, use a Try...Catch statement.

You can also sort a section of the array by using the following form of the Sort method, where *startIndex* and *endIndex* are the indices that delimit the section of the array to be sorted:

```
System.Array.Sort(arrayName, startIndex, endIndex)
```

An interesting variation of the Sort method sorts the elements of an array according to the values of the elements in another array. Let's say you have one array of names and another of matching Social Security numbers. It is possible to sort the array of names according to their Social Security numbers. This form of the Sort method has the following syntax:

```
System.Array.Sort(array1, array2)
```

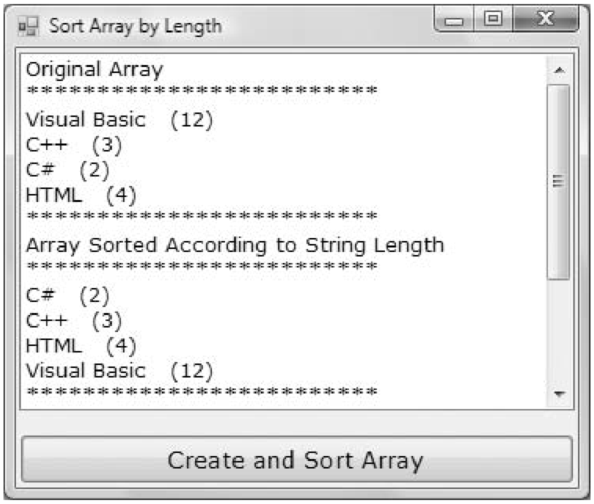
array1 is the array with the keys (the Social Security numbers), and *array2* is the array with the actual elements to be sorted. This is a very handy form of the Sort method. Let's say you have a list of words stored in one array and their frequencies in another. Using the first form of the Sort method, you can sort the words alphabetically. With the third form of the Sort method, you can sort them according to their frequencies (starting with the most common words and ending with the least common ones). The two arrays must be one-dimensional and have the same number of elements. If you want to sort a section of the array, just supply the *startIndex* and *endIndex* arguments to the Sort method, after the names of the two arrays.

The SortArrayByLength application, shown in Figure 14.1, demonstrates how to sort an array based on the length of its elements (short elements appear at the top of the array, whereas longer elements appear near the bottom of the array). First, it populates the array MyStrings with a few strings and then it assigns the lengths of these strings to the matching elements of the array MyStringsLen. The MyStrings(0) element's value is Visual Basic, and the MyStringsLen(0) element's value is 12. After the two arrays have been populated, the code sorts the elements of the MyStrings array according to the values of the MyStringsLen array. The statement that sorts the array is the following:

```
System.Array.Sort(MyStringsLen, MyStrings)
```

The code, which also displays the arrays before and after sorting, is shown in Listing 14.1.

FIGURE 14.1
The SortArrayByLength application



```
Original Array
*************************
Visual Basic   (12)
C++   (3)
C#   (2)
HTML   (4)
*************************
Array Sorted According to String Length
*************************
C#   (2)
C++   (3)
HTML   (4)
Visual Basic   (12)
*************************
```

LISTING 14.1: Sorting an Array According to the Length of Its Elements

```
Protected Sub Button1_Click(...) Handles Button1.Click
  Dim MyStrings(3) As String
  ' populate MyStrings array
  Dim MyStringsLen(3) As Integer
  MyStrings(0) = "Visual Basic"
  MyStrings(1) = "C++"
  MyStrings(2) = "C#"
  MyStrings(3) = "HTML"
  ' populate MyStringsLen array
  Dim i As Integer
  For i = 0 To UBound(MyStrings)
     MyStringsLen(i) = len(MyStrings(i))
  Next
  ListBox1.Items.Clear()
  ListBox1.Items.Add("Original Array")
  ListBox1.Items.Add("*************************")
  Dim str As Integer
  For str = 0 To UBound(MyStrings)
     ListBox1.Items.Add(MyStrings(str) & "    " & MyStringsLen(str).ToString)
  Next
  ListBox1.Items.Add("*************************")
  ListBox1.Items.Add("Array Sorted According to String Length ")
  ListBox1.Items.Add("*************************")
```

```
    ' sort MyStrings array based on MyStringsLen array
    System.Array.Sort(MyStringsLen, MyStrings)
    For str = 0 To UBound(MyStrings)
       ListBox1.Items.Add(MyStrings(str) & "    " & _
                   MyStringsLen(str).ToString)
    Next
  End Sub
```

The output produced by the SortArrayByLength application in the ListBox control is shown here:

```
Original Array
*************************
Visual Basic   12
C++   3
C#   2
HTML   4
*************************
Array Sorted According to String Length
*************************
C#   2
C++   3
HTML   4
Visual Basic   12
```

Notice that the `Sort` method sorts both the auxiliary array (the one with the lengths of the strings) and the main array, so that the two arrays are always in synch. After the call to the `Sort` method, the first element in the `MyStrings` array is C#, and the first element in the `MyStringsLen` array is 2.

The array with the keys that determine the order of the elements can be anything. If the array to be sorted holds some rectangles, you can create an auxiliary array that contains the area of the rectangles and sort the original array according to the area of its rectangles. Likewise, an array of colors can be sorted according to the hue or the luminance of each color component, and so on.

Another form of the `Sort` method uses a user-supplied function to sort arrays of custom objects. As you recall, arrays can store all types of objects. But the Framework doesn't know how to sort your custom objects. To sort an array of objects, you must provide your own class that implements the IComparer interface (basically, a function that can compare two instances of a custom class). This form of the `Sort` method is described in detail in the section titled ''Custom Sorting,'' later in this chapter. By the way, you can create a function to sort an array based on the length of its elements or any other property of its elements, similar to the `Sort` method that uses the items of an auxiliary array as keys.

Searching Arrays

Arrays can be searched in two ways: with the `BinarySearch` method, which works on sorted arrays and is extremely fast, and with the `IndexOf` (and `LastIndexOf`) methods, which work regardless of the order of the elements. All three methods search for an instance of an item and return its index, and they're all reference methods. The `IndexOf` and `LastIndexOf` methods are similar to the methods by the same name of the String class. They return the index of the first (or last) instance of an object in the array, or the value −1 if the object isn't found in the array. Both methods are overloaded, and the simplest form of the `IndexOf` method is the following, where *`arrayName`* is the name of the array to be searched and *`object`* is the item you're searching for:

```
itemIndex = System.Array.IndexOf(arrayName, object)
```

The `LastIndexOf` method's syntax is identical, but the `LastIndexOf` method starts searching from the end of the array. If the item you're searching for is unique in the array, both methods will return the same index.

Another form of the `IndexOf` and `LastIndexOf` methods allows you to begin the search at a specific index:

```
itemIndex = System.Array.IndexOf(arrayName, object, startIndex)
```

This form of the method starts searching in the segment of the array from *`startIndex`* to the end of the array. Finally, you can specify a range of indices in which the search will take place by using the following form of the method:

```
itemIndex = System.Array.IndexOf( _
            arrayName, object, startIndex, endIndex)
```

You can search large arrays more efficiently with the `BinarySearch` method if the array is sorted. The simplest form of the `BinarySearch` method is the following:

```
System.Array.BinarySearch(arrayName, object)
```

The `BinarySearch` method returns an integer value, which is the index of the object you're searching for in the array. If the *`object`* argument is not found, the method returns a negative value, which is the negative of the index of the next larger item minus one. This transformation, the negative of a number minus one, is called the *one's complement*, and other languages provide an operator for it: the tilde (~). The one's complement of 10 is −11, and the one's complement of −3 is 2.

Why all this complexity? Zero is a valid index, so only a negative value could indicate a failure in the search operation. A value of −1 would indicate that the operation failed, but the `BinarySearch` method does something better. If it can't locate the item, it returns the index of the item immediately after the desired item (the first item in the array that exceeds the item you're searching for). This is a near match, and the `BinarySearch` method returns a negative value to indicate near matches. A near match is usually the same string with different character casing, or a slightly different spelling. It may also be a string that's totally irrelevant to the one you're searching for. Notice that there will always be a near match unless you're searching for a value larger than the last value in the array. In this case, the `BinarySearch` method will return the one's complement of the array's upper bound (−100 for an array of 100 elements, if you consider that the index of the last element is 99).

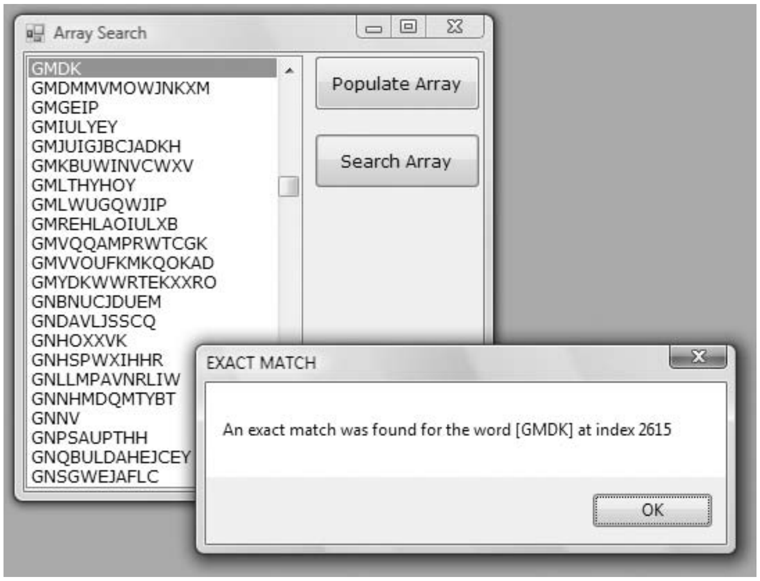
Arrays Perform Case-Sensitive Searches

The `BinarySearch`, `IndexOf`, and `LastIndexOf` methods perform case-sensitive searches. However, because the `BinarySearch` method reports near matches, it appears as if it performs case-insensitive searches. If the array contains the element *Charles* and you search for *charles*, the `IndexOf` method will not find the string and will report a no-match, whereas the `BinarySearch` method will find the element *Charles* and report it as a near match. My recommendation is to standardize the case of the data and the search argument when you plan to perform searches (such as uppercase for titles, camel case for names, and so on). To perform case-insensitive searches, you must implement your own custom comparer, a process that's described later in this chapter. Also the `Option Compare` statement has no effect on the comparisons performed by either the `BinarySearch` or the `IndexOf`/`LastIndexOf` methods.

VB2008 At Work: The ArraySearch Application

The ArraySearch application, shown in Figure 14.2, demonstrates how to handle exact and near matches reported by the `BinarySearch` method. The Populate Array button populates an array with 10,000 random strings. The same strings are also displayed in a sorted ListBox control, so you can view them. The elements have the same order in both the array and the ListBox, so we can use the index reported by the `BinarySearch` method to locate and select instantly the same item in the ListBox.

Each of the 10,000 random strings has a random length of 3 to 15 characters. When you run the application, message boxes will pop up, displaying the time it took for each operation: how long it took to populate the array, how long it took to sort it, and how long it took to populate the ListBox. You might want to experiment with large arrays (100,000 elements or more) to get an idea of how efficiently VB 2008 handles arrays.



GMDK
GMDMMVMOWJNKXM
GMGEIP
GMIULYEY
GMJUIGJBCJADKH
GMKBUWINVCWXV
GMLTHYHOY
GMLWUGQWJIP
GMREHLAOIULXB
GMVQQAMPRWTCGK
GMVVOUFKMKQOKAD
GMYDKWWRTEKXXRO
GNBNUCJDUEM
GNDAVLJSSCQ
GNHOXXVK
GNHSPWXIHHR
GNLLMPAVNRLIW
GNNHMDQMTYBT
GNNV
GNPSAUPTHH
GNQBULDAHEJCEY
GNSGWEJAFLC

An exact match was found for the word [GMDK] at index 2615

Figure 14.2
Searching an array and locating the same element in the ListBox control

The Search Array button prompts the user for a string via the `InputBox()` function and then locates the string in the array by calling the `BinarySearch` method in the array. The result is either an exact or a near match, and it's displayed in a message box. At the same time, the item reported by the `BinarySearch` method is also selected in the ListBox control.

Run the application, populate the ListBox control, and then click the Search Array button. Enter an existing string (you can use lowercase or uppercase characters; it doesn't make a difference) and verify that the application reports an exact match and locates the item in the ListBox. The program appears to perform case-insensitive searches because all the strings stored in the array are in uppercase, and the search argument is also converted to uppercase before the `BinarySearch` method is called. Then enter a string that doesn't exist in the list (or the beginning of an existing string) and see how the `BinarySearch` handles near matches.

The code behind the Search Array button calls the `BinarySearch` method and stores the integer returned by the method to the *`wordIndex`* variable. Then it examines the value of this variable. If *`wordIndex`* is positive, there was an exact match, and it's reported. If *`wordIndex`* is negative, the program calculates the one's complement of this value, which is the index of the nearest match. The element at this index is reported as a near match. Finally, regardless of the type of the match, the code selects the same item in the ListBox and makes it visible. Listing 14.2 is the code behind the Search Array button.

LISTING 14.2: Locating Exact and Near Matches with *BinarySearch*

```
Private Sub bttnSearch_Click(...) Handles bttnSearch.Click
   Dim srchWord As String    ' the word to search for
   Dim wordIndex As Integer  ' the index of the word
   srchWord = InputBox( _
              "Enter word to search for").ToUpper
   wordIndex = System.Array.BinarySearch(words, srchWord)
       If wordIndex >= 0 Then  ' exact match!
       ListBox1.TopIndex = wordIndex
       ListBox1.SelectedIndex = wordIndex
       MsgBox("An exact match was found for " & _
          " the word [" & words(wordIndex) & _
          "] at index " & wordIndex.ToString,, _
          "EXACT MATCH")
   Else                     ' Near match
       ListBox1.TopIndex = -wordIndex - 1
       ListBox1.SelectedIndex = -wordIndex - 1
       MsgBox("The nearest match is the word [" & _
           words(-wordIndex - 1) & "] at index " & _
           (-wordIndex - 1).ToString, , "NEAR MATCH")
   End If
End Sub
```

Notice that all methods for sorting and searching arrays work with the base data types only. If the array contains custom data types, you must supply your own functions for comparing elements of this type, a process described in detail in the section ''Custom Sorting,'' later in this chapter.

The Binary Search Algorithm

The `BinarySearch` method uses a powerful search algorithm, the *binary search algorithm*, but it requires that the array be sorted. You need not care about the technical details of the implementation of a method, but in the case of the binary search algorithm, a basic understanding of how it works will help you understand how it performs near matches.

To locate an item in a sorted array, the `BinarySearch` method compares the search string to the array's middle element. If the search string is smaller, we know that the element is in the first half of the array and we can safely ignore the second half. The same process is repeated with the remaining half of the elements. The search string is compared with the middle element of the reduced array, and after the comparison, we can ignore one-half of the reduced array. At each step, the binary search algorithm rejects one-half of the items left until it reduces the list to a single item. This is the item we're searching for. If not, the item is not in the list. To search a list of 1,024 items, the binary search algorithm makes 10 comparisons. At the first step, it rejects 512 elements, then 256, then 128, and so on, until it reaches a single element. For an array of 1,024 $\times$ 1,024 (that's a little more than a million) items, the algorithm makes 20 comparisons to locate the desired item.

If you apply the `BinarySearch` method to an array that hasn't been sorted, the method will carry out all the steps and report that the item wasn't found, even if the item belongs to the array. The algorithm doesn't check the order of the elements; it just assumes that they're sorted. The binary search algorithm always halves the number of elements in which it attempts to locate the search argument. That's why you should never apply the `BinarySearch` method to an array that hasn't been sorted yet.

To see what happens when you apply the `BinarySearch` method to an array that hasn't been sorted, remove the statement that calls the `Sort` method in the ArraySearch sample application. The application will keep reporting near matches, even if the string you're searching is present in the array. Of course, the near match reported by the `BinarySearch` method in an unsorted array isn't close to the element you're searching for — it's just an element that happens to be there when the algorithm finishes.

Performing Other Array Operations

The Array class exposes additional methods, which are described briefly in this section. The `Reverse` method reverses the order of the elements in an array. The syntax of the `Reverse` method is the following:

```
reversedArray = System.Array.Reverse(arrayName)
```

The `Reverse` method can't be applied to an array and reverse its elements. Instead, it returns a new array with the elements of the array passed as an argument, only in reverse order.

The `Copy` and `CopyTo` methods copy the elements of an array (or segment of an array) to another array. The syntax of the `Copy` method is the following:

```
System.Array.Copy(sourceArray, destinationArray, count)
```

`sourceArray` and *`destinationArray`* are the names of the two arrays, and *`count`* is the number of elements to be copied. The copying process starts with the first element of the source array and ends after the first *`count`* elements have been copied. If *`count`* exceeds the length of either array, an exception will be thrown.

Another form of the Copy method allows you to specify the range of elements in the source array to be copied and a range in the destination array in which these elements will be copied. The syntax of this form of the method is as follows:

```
System.Array.Copy(sourceArray, sourceStart,_
         destinationArray, destinationStart, count)
```

This method copies *count* elements from the source array, starting at location *sourceStart*, and places them in the destination array, starting at location *destinationStart*. All indices must be valid, and there should be *count* elements after the *sourceStart* index in the source array, as well as *count* elements after the *destinationStart* in the destination array. If not, an exception will be thrown.

The CopyTo method is similar, but it doesn't require the name of the source array. It copies the elements of the array to which it's applied into the destination array, where *sourceArray* is a properly dimensioned and initialized array:

```
sourceArray.CopyTo(destinationArray, sourceStart)
```

Finally, you can filter array elements by using the Filter() function, which is not a method of the Array class; it's a VB function that acts on arrays. The Filter() function performs an element-by-element comparison and rejects the elements that don't meet the user-specified criteria. The filtered elements are returned as a new array, while the original array remains intact. The syntax of the Filter() function is as follows:

```
filteredArray = Filter(source, match, include, compare)
```

source is the array to be searched, and it must be a one-dimensional array of strings or objects. The *match* argument is the string to search for. Every element that includes the specified string is considered a match. The remaining arguments are optional: include is a True/False value indicating whether the method will return the elements that include (if True) or exclude (if False) the matching elements. The compare argument is a member of the CompareMethod enumeration: It can be Binary (for binary or case-sensitive comparisons) or Text (for textual or case-insensitive comparisons). If no match is found, the method will return an empty array.

The following code segment filters out the strings that don't contain the word *visual* from the words array:

```
Dim words() As String = {"Visual Basic", "Java", "Visual Studio"}
Dim selectedWords() As String
selectedWords = Filter(words, "visual", True, CompareMethod.Text)
Dim selword As String
Dim msg As String = ""
For Each selword In selectedWords
    msg &= selword & vbCrLf
Next
MsgBox(msg)
```

If you execute the preceding statements, the message box will display the following:

```
Visual Basic
Visual Studio
```

Change the last argument of the `Filter` function to `CompareMethod.Binary`, and no elements will be displayed. The method will return an empty array, because no element contains the search argument with the exact spelling as specified by the `match` argument.

Among the new array features introduced with version 3.5 of the Framework, I've singled out the `FindAll` method, which selects multiple elements and returns them as a new array. The `FindAll` method accepts two arguments: the array to be searched and a predicate, which is a pointer to a function that selects the desired elements. Here's the syntax of the `FindAll` method:

```
System.Array.FindAll(array, match)
```

The `array` argument must be strongly typed, and the `match` predicate must be a function that accepts as an argument a value of the same type as the array. The formal syntax of the `FindAll` method in the documentation is the following:

```
Array.FindAll(Of T)(array() As T, _
                match As System.Predicate(Of T)) As T()
```

It's not as bad as it looks: `T` stands for Type. The `FindAll` method returns an array with elements of the `T` type. The array it accepts as an argument must also be of the same type. Finally, the predicate is a function that accepts as an argument an object of the `T` type.

Let's say you have an array of integers and you want to select the ones that are positive and less than 100. Start by writing a function that accepts as an argument an integer value and returns it if it meets the specified criteria:

```
Private Function smallValue(ByVal v As Integer) As Integer
    If v > 0 and v < 100 Then Return v
End Function
```

Then call the `FindAll` method, passing a pointer to the `smallValue()` function with the `AddressOf` operator:

```
Dim values(999) As Integer
' statements to populate array
Dim selected() As Integer
Selected = System.Array.FindAll(values, AddressOf smallValue)
```

The `FindAll` method is a shortcut to a loop that iterates through all the elements of the array and calls the same function to select or reject each element. The predicate is a regular function that can get as complicated as dictated by the needs of your application.

A similar method of the Array class, the `TrueForAll` method, applies a function to all the elements of the array and returns True if the function returns True for every element in the array. You can use this method to make sure that an array's elements are of the same type, or that they all meet a specified requirement. The function you must provide should examine each element of the array and return True if it meets the criteria, as shown in the following sample function, which assumes that the elements are Rectangle objects and returns True if their area exceeds 1:

```
Private Function largeRectangle(ByVal R As Rectangle) As Boolean
    If R.Width * R.Height > 1 Then
        Return True
    Else
        Return False
    End If
End Function
```

The following statements show how you would use this function with the `TrueForAll` method. If True, then all the elements of the `Rects` array are large (their area exceeds 1):

```
Dim Rects(10000) As Rectangle
' statements to populate array
MsgBox(System.Array.TrueForAll(Rects, AddressOf largeRectangle))
```

Array Limitations

As implemented in version 3.5 of the Framework, arrays are more flexible than ever. They're very efficient, and the most demanding tasks programmers had to perform with arrays are now implemented as methods of the Array class. However, arrays aren't perfect for all types of data storage. The most important shortcoming of arrays is that they're not dynamic. Inserting or removing elements entails that all the following elements be moved up or down. The ArrayList collection is similar to the array, but it's a dynamic structure. You can insert and remove elements to an ArrayList without having to worry about reordering the other elements in the collection. Arrays are the most efficient collection in the Framework, but when you need a dynamic structure for adding and removing elements in the course of an application, you should use an ArrayList object.

The ArrayList Collection

The ArrayList collection allows you to maintain multiple elements, similar to an array; however, the ArrayList collection allows the insertion of elements anywhere in the collection, as well as the removal of any element. In other words, it's a dynamic structure that can also grow automatically as you add/remove elements. Like an array, the ArrayList's elements can be sorted and searched. In effect, the ArrayList is a more ''convenient'' array, a dynamic array. You can also remove elements by value, not only by index. If you have an ArrayList populated with names, you remove the item `Charles` by passing the string itself as an argument. Notice that `Charles` is not an index value; it's the element you want to remove.

Creating an ArrayList

To use an ArrayList in your code, you must first create an instance of the ArrayList class by using the `New` keyword, as in the following statement:

```
Dim aList As New ArrayList
```

The `aList` variable represents an ArrayList that can hold only 16 elements (the default size). You can set the initial capacity of the ArrayList by setting its `Capacity` property, which is the number of elements the ArrayList can hold. The ArrayList's capacity can be increased or reduced at any time, just by setting the `Capacity` property. You can also specify the collection's initial capacity in the ArrayList's constructor:

```
Dim aList As New ArrayList(1000)
```

Notice that you don't have to prepare the collection to accept a specific number of items. Every time you exceed the collection's capacity, it's doubled automatically. However, it's not decreased automatically when you remove items.

The exact number of items currently in the ArrayList is given by the `Count` property, which is always less than (or, at most, equal to) the `Capacity` property. (Both properties are expressed in terms of items.) If you decide that you will no longer add more items to the collection, you can call the `TrimToSize` method, which will set the collection's capacity to the number of items in the list.

Adding and Removing ArrayList Items

To add a new item to an ArrayList, use the `Add` method, whose syntax is as follows:

```
index = aList.Add(obj)
```

`aList` is a properly declared ArrayList, and *`obj`* is the item you want to add to the ArrayList collection (it could be a number, a string, or an object). The `Add` method appends the specified item to the collection and returns the index of the new item. If you're using an ArrayList named `Capitals` to store the names of the state capitals, you can add an item by using the following statement:

```
Capitals.Add("Sacramento")
```

If the `Persons` ArrayList holds variables of a custom type, prepare a variable of that type and then add it to the collection. Let's say you created a structure called `Person` by using the following declaration:

```
Structure Person
   Dim LastName As String
   Dim FirstName As String
   Dim Phone As String
   Dim EMail As String
End Structure
```

To store a collection of `Person` items in an ArrayList, create a variable of the `Person` type, set its fields, and then add it to the ArrayList, as shown in Listing 14.3.

LISTING 14.3: Adding a Structure to an ArrayList

```
Dim Persons As New ArrayList
Dim p As New Person
p.LastName = "Last Name"
p.FirstName = "First Name"
p.Phone = "Phone"
p.EMail = "name@server.com"
Persons.Add(p)
p = New Person
p.LastName = "another name"
{ statements to set the other fields}
Persons.Add(p)
```

If you execute these statements, the ArrayList will hold two items, both of the `Person` type. Notice that you can add multiple instances of the same object to the ArrayList collection. To find out whether an item belongs to the collection already, use the `Contains` method, which accepts as an argument an object and returns a True or False value, depending on whether the object belongs to the list:

```
If Persons.Contains(p) Then
   MsgBox("Duplicate element rejected")
Else
   Persons.Add(p)
   MsgBox("Element appended successfully")
End If
```

By default, items are appended to the ArrayList. To insert an item at a specific location, use the `Insert` method, which accepts as an argument the location at which the new item will be inserted and, of course, an object to insert in the ArrayList, as shown next:

```
aList.Insert(index, object)
```

Unlike the `Add` method, the `Insert` method doesn't return a value — the location of the new item is already known.

You can also add multiple items via a single call to the `AddRange` method. This method appends a collection of items to the ArrayList. These items could come from an array or from another ArrayList. The following statement appends the elements of an array to the `aList` collection:

```
Dim colors() As Color = {Color.Red, Color.Blue, Color.Green}
aList.AddRange(colors)
```

The AddRange method in this example appends three items of the same type to the aList collection. The array could have been declared as Object, too; it doesn't have to be strongly typed because the ArrayList collection is not strongly typed.

To insert a range of items anywhere in the ArrayList, use the InsertRange method. Its syntax is the following, where *index* is the index of the ArrayList where the new elements will be inserted, and *objects* is a collection of the elements to be inserted:

```
aList.InsertRange(index, objects)
```

Finally, you can overwrite a range of elements in the ArrayList with a new range by using the SetRange method. To overwrite the items in locations 5 through 9 in an ArrayList, use a few statements like the following:

```
Dim words() As String = _
               {"Just", "a", "few", "more", "words"}
aList.SetRange(5, words)
```

This code segment assumes that the aList collection contains at least 10 items, and it replaces half of them.

To remove an item, use the Remove method, whose syntax is the following:

```
aList.Remove(object)
```

The *object* argument is the value to be removed, not an index value. If the collection contains multiple instances of the same item, only the first instance of the object will be removed.

Notice that the Remove method compares values, not references. If the ArrayList contains a Rectangle object, you can search for this item by creating a new Rectangle variable and setting its properties to the properties of the Rectangle object you want to remove:

```
Dim R1 As New Rectangle(10, 10, 100, 100)
Dim R2 As Rectangle = R1
aList.Add(R1)
aList.Add(R2)
Dim R3 As Rectangle
R3 = New Rectangle(10, 10, 100, 100)
aList.Remove(R3)
```

If you execute these statements, they will add two identical rectangles to the aList ArrayList. The last statement will remove the first of the two rectangles.

If you attempt to remove an item that doesn't exist, no exception is thrown — simply, no item is removed from the list. You can also remove items by specifying their index in the list via the RemoveAt method. This method accepts as an argument the index of the item to remove, which must be less than the number of items currently in the list.

To remove more than one consecutive item, use the RemoveRange method, whose syntax is the following:

```
aList.RemoveRange(startIndex, count)
```

The *startIndex* argument is the index of the first item to be removed, and *count* is the number of items to be removed.

The following statements are examples of the methods that remove items from an ArrayList collection. The first two statements remove an item by value. The first statement removes an object, and the second removes a string item. The following statement removes the third item, and the last one removes the third through the fifth items.

```
aList.Remove(Color.Red)
aList.Remove("Richard")
aList.RemoveAt(2)
aList.RemoveRange(2, 3)
```

Extracting Items from an ArrayList

To access the items in the ArrayList, use an index value, similar to the array. The first item's index is 0 and the last item's index is `AL.Count-1`, where *`AL`* is a properly initialized ArrayList collection. You can also extract a range of items from the list by using the `GetRange` method. This method extracts a number of consecutive elements from the ArrayList and stores them to a new ArrayList, where *`index`* is the index of the first item to copy, and *`count`* is the number of items to be copied:

```
newList = ArrayList.GetRange(index, count)
```

The `GetRange` method returns another ArrayList with the proper number of items. The following statement copies three items from the `aList` ArrayList and inserts them at the beginning of the `bList` ArrayList. The three elements copied are the fourth through sixth elements in the original collection:

```
bList.InsertRange(0, aList.GetRange(3, 3))
```

The `Repeat` method, which fills an ArrayList with multiple instances of the same item, has the following syntax:

```
newList = ArrayList.Repeat(item, count)
```

This method returns a new ArrayList with *`count`* elements, all of them being identical to the *`item`* argument.

Another method of the ArrayList class is the `Reverse` method, which reverses the order of the elements in an ArrayList collection, or a portion of it.

Sorting ArrayLists

To sort the ArrayList, use the `Sort` method, which has three overloaded forms:

```
aList.Sort()
aList.Sort(comparer)
aList.Sort(startIndex, endIndex, comparer)
```

The ArrayList's `Sort` method doesn't require you to pass the name of the ArrayList to be sorted as an argument; unlike the `Sort` method of the Array class, this is an instance method and sorts the

ArrayList object to which it's applied. `aList` is a properly declared and initialized ArrayList object. The first form of the `Sort` method sorts the ArrayList alphabetically or numerically, depending on the data type of the objects stored in it. If the items are not all of the same type, an exception will be thrown. You'll see how you can handle this exception shortly.

If the items stored in the ArrayList are of a data type other than the base data types, you must supply your own mechanism to compare the objects. The other two forms of the `Sort` method use a custom function for comparing items; you will see how they're used in the section ''Custom Sorting,'' later in this chapter. Notice that there is no overloaded form of the `Sort` method that sorts a section of the ArrayList.

If the list contains items of widely different types, the `Sort` method will fail. To prevent a runtime exception (the `InvalidOperationException`), you must make sure that all items are of the same type. If you can't ensure that all the items are of the same type, catch the possible errors and handle them from within a structured exception handler, as demonstrated in Listing 14.4.

LISTING 14.4: Foolproof Sorting

```
Dim sorted As Boolean = True
Try
  aList.Sort()
Catch SortException As InvalidOperationException
     MsgBox("You can't sort an ArrayList whose items " & _
            "aren't of the same type")
     sorted = False
Catch GeneralException As Exception
     MsgBox("The following exception occurred:" & _
            vbCrLf & GeneralException.Message)   sorted = False
End Try
If sorted Then
  { process sorted ArrayList}
Else
  { process unsorted list}
End If
```

The *`sorted`* Boolean variable is initially set to True because the `Sort` method will most likely succeed. If not, an exception will be thrown, in which case the code resets the *`sorted`* variable to False and uses it later to distinguish between sorted and unsorted collections. Notice the two clauses of the `Catch` statement that distinguish between the most common exception that the `Sort` method can throw (the invalid operation exception) and any other type of exception.

The `Sort` method can't even sort a collection of various numeric data types. If some of its elements are doubles and some are integers or decimals, the `Sort` method will fail. You must either make sure that all the items in the ArrayList are of the same type, or provide your own function for comparing the ArrayList's items. The best practice is to make sure that your collection contains items of the same type. If a collection contains items of different types, how likely is it that you'll have to sort such a collection?

Searching ArrayLists

Like arrays, the ArrayList class exposes the `IndexOf` and `LastIndexOf` methods to search in an unsorted list and the `BinarySearch` method for sorted lists. The `IndexOf` and `LastIndexOf` methods accept as an argument the object to be located and return an index:

```
Dim index As Integer = aList.IndexOf(object)
```

Here, ***`object`*** is the item you're searching. The `LastIndexOf` method has the same syntax, but it starts scanning the array from its end and moves backward toward the beginning. The `IndexOf` and `LastIndexOf` methods are overloaded. The other two forms of the `IndexOf` method are these:

```
aList.IndexOf(object, startIndex)
aList.IndexOf(object, startIndex, length)
```

The two additional arguments determine where the search starts and ends. Both methods return the index of the item if it belongs to the collection. If not, they return the value −1. The `IndexOf` and `LastIndexOf` methods of the ArrayList class perform case-sensitive searches, and they report exact matches only.

If the ArrayList is sorted, use the `BinarySearch` method, which accepts as an argument the object to be located and returns its index in the collection, where ***`object`*** is the item you're looking for:

```
Dim index As Integer = aList.BinarySearch(object)
```

There are two more forms of this method. To search for an item in an ArrayList with custom objects, use the following form of the `BinarySearch` method:

```
Dim index As Integer = aList.BinarySearch(object, comparer)
```

The first argument is the object you're searching for, and the second is the name of an IComparer object. Another form of the `BinarySearch` method allows you to search for an item in a section of the collection; its syntax is as follows:

```
Dim index As Integer = _
         aList.BinarySearch(startIndex, length, object, comparer)
```

The first argument is the index at which the search will begin, and the second argument is the length of the subrange. ***`object`*** and ***`comparer`*** are the same as with the second form of the method.

Iterating an ArrayList

To iterate through the elements of an ArrayList collection, you can set up a `For...Next` loop like the following one:

```
For i = 0 To aList.Count - 1
   { process item aList(i)}
Next
```

This is a trivial operation, but the processing itself can get as complicated as required by the type of objects stored in the collection. The current item at each iteration is the `aList(i)`. It's recommended that you cast the object to the appropriate type and then process it.

You can also use the `For Each...Next` loop with an `Object` variable, as shown next:

```
Dim itm As Object
For Each itm In aList
   { process item itm}
Next
```

If all the items in the ArrayList are of the same type, you can use a variable of this type to iterate through the collection, instead of a generic object variable. If all the elements are decimals, for example, you can declare the `itm` variable as Decimal.

An even better method is to create an enumerator for the collection and use it to iterate through its items. This technique applies to all collections and is discussed in the ''Enumerating Collections'' section later in this chapter.

The ArrayList class addresses most of the problems associated with the Array class, but one last problem remains — that of accessing the items in the collection through a meaningful key. This is the problem addressed by the HashTable collection.

The HashTable Collection

As you saw, the ArrayList addresses most of the problems of the Array class, while it supports all the convenient array features. Yet, the ArrayList, like the Array, has a major drawback: You must access its items by an index value. Another collection, the HashTable collection, is similar to the ArrayList, but it allows you to access the items by a key.

Each item in a HashTable has a value and a key. The *value* is the same value you'd store in an array, but the *key* is a meaningful entity for accessing the items in the collection, and each element's key must be unique. Both the values stored in a HashTable and their keys can be objects. Typically, the keys are short strings or integers.

The HashTable collection exposes most of the properties and methods of the ArrayList, with a few notable exceptions. The `Count` property returns the number of items in the collection as usual, but the HashTable collection doesn't expose a `Capacity` property. The HashTable collection uses fairly complicated logic to maintain the list of items, and it adjusts its capacity automatically. Fortunately, you need not know how the items are stored in the collection.

To create a HashTable in your code, declare it with the `New` keyword:

```
Dim hTable As New HashTable
```

To add an item to the HashTable, use the `Add` method with the following syntax:

```
hTable.Add(key, value)
```

`value` is the item you want to add (it can be any object), and *`key`* is a value you supply, which represents the item. This is the value you'll use later to retrieve the item. If you're setting up a structure for storing temperatures in various cities, use the city names as keys:

```
Dim Temperatures As New HashTable
Temperatures.Add("Houston", 81)
Temperatures.Add("Los Angeles", 78)
```

To find out the temperature in Houston, use the following statement:

```
MsgBox(Temperatures("Houston").ToString)
```

Notice that you can have duplicate values, but the keys must be unique. If you attempt to use an existing key, an `InvalidArgumentException` exception will be thrown. To find out whether a specific key or value is already in the collection, use the `ContainsKey` and `ContainsValue` methods. The syntax of the two methods is quite similar, and they return True if the specified value is already in use in the collection:

```
hTable.ContainsKey(object)
hTable.ContainsValue(object)
```

The HashTable collection exposes the `Contains` method, too, which is identical to the `ContainsKey` method.

To find out whether a specific key is in use already, use the `ContainsKey` method, as shown in the following statements, which add a new item to the HashTable only if its key doesn't exist already:

```
Dim value As New Rectangle(100, 100, 50, 50)
Dim key As String = "Rect1"
If Not hTable.ContainsKey(key) Then
   hTable.Add(key, value)
End If
```

The `Values` and `Keys` properties allow you to retrieve all the values and the keys in the HashTable, respectively. Both properties are collections and expose the usual members of a collection. To iterate through the values stored in the HashTable `hTable`, use the following loop:

```
Dim itm As Object
For Each itm In hTable.Values
   Debug.WriteLine(itm)
Next
```

There is only one method to remove items from a HashTable — the `Remove` method, which accepts as an argument the key of the item to be removed:

```
hTable.Remove(key)
```

To extract items from a HashTable, use the `CopyTo` method. This method copies the items to a one-dimensional array, and its syntax is as follows:

```
newArray = HTable.CopyTo(arrayName)
```

You must set up the array that will accept the items beforehand, but you need not supply its dimensions in the declaration. The `CopyTo` method can throw several different exceptions for various error conditions. The array that accepts the values must be one-dimensional, and there should be enough space in the array for the HashTable's values. Moreover, the array's type must be `Object` because a HashTable stores objects.

Listing 14.5 demonstrates how to scan the keys of a HashTable through the `Keys` property and then use these keys to access the matching items through the `Item` property.

LISTING 14.5: Iterating a HashTable

```
Private Function ShowHashTableContents( _
             ByVal table As Hashtable) As String
  Dim msg As String
  Dim element, key As Object
  msg = "The HashTable contains " & _
         table.Count.tostring & " elements: " & vbCrLf
  For Each key In table.keys
     element = table.Item(key)
     msg = msg & vbCrLf
     msg = msg & "    Element Type = " & element.GetType.ToString
     msg = msg & vbCrLf & "    Element Key= " & Key.ToString
     msg = msg & "    Element Value= " & element.ToString & vbCrLf
  Next
  Return(msg)
End Sub
```

To print the contents of a HashTable variable in the Output window, call the `ShowHashTable-Contents()` function, passing the name of the HashTable as an argument, and then print the string returned by the function:

```
Dim HT As New HashTable
{ statements to populate HashTable}
Debug.WriteLine(ShowHashTableContents(HT))
```

VB 2008 at Work: The WordFrequencies Project

In this section, you'll develop an application that counts word frequencies in a text. The Word-Frequencies application scans text files and counts the occurrences of each word in the text. As you will see, the HashTable is the natural choice for storing this information because you want to access a word's frequency by using the actual word as the key. To retrieve (or update) the frequency of the word *elaborate*, for example, you will use this expression:

```
Words("ELABORATE").Value
```

where `Words` is a properly initialized HashTable object.

When the code runs into another instance of the word *elaborate*, it simply increases the matching item of the `Words` HashTable by one:

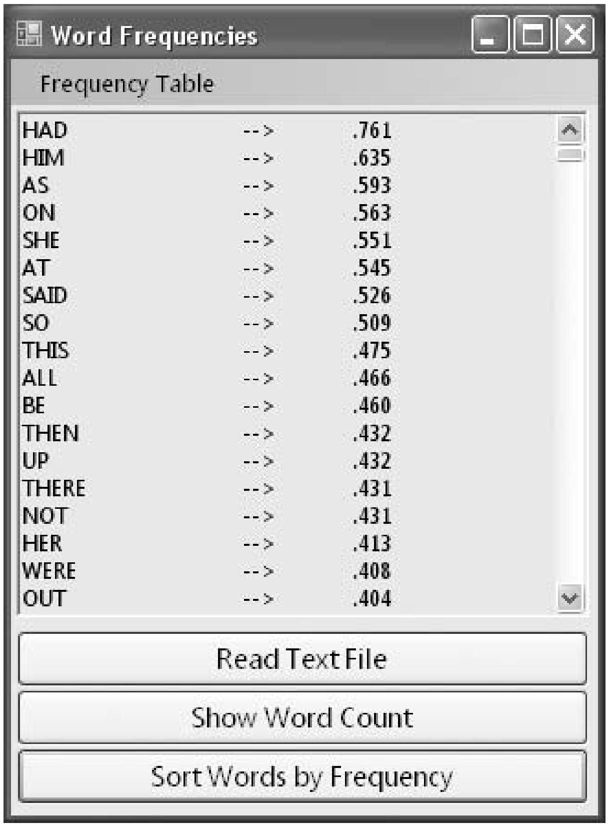
```
Words("ELABORATE").Value += 1
```

Arrays and ArrayLists are out of the question because they can't be accessed by a key. You could also use the `SortedList` collection (described later in this chapter), but this collection maintains its items sorted at all times. If you need this functionality as well, you can modify the application accordingly. The items in a `SortedList` are also accessed by keys, so you won't have to introduce substantial changes in the code.

Let me start with a few remarks. First, all words we locate in the various text files will be converted to uppercase. Because the keys of the HashTable are case-sensitive, converting them to uppercase eliminates the usual problem of case-sensitivity (*hello* being a different word than *Hello* and *HELLO*) by eliminating multiple possible spellings for the same word.

The frequencies of the words can't be calculated instantly because we need to know the total number of words in the text. Instead, each value in the HashTable is the number of occurrences of a specific word. To calculate the actual frequency of the same word, we must divide this value by the number of occurrences of all words, but this can happen only after we have scanned the entire text file and counted the occurrences of each word.

The application's interface is shown in Figure 14.3. To scan a text file and process its words, click the Read Text File button. The Open dialog box will prompt you to select the text file to be processed, and the application will display in a message box the number of unique words read from the file. Then you can click the Show Word Count button to count the number of occurrences of each word in the text. The last button on the form calculates the frequency of each word and sorts the words according to their frequencies.



```
HAD       -->   .761
HIM       -->   .635
AS        -->   .593
ON        -->   .563
SHE       -->   .551
AT        -->   .545
SAID      -->   .526
SO        -->   .509
THIS      -->   .475
ALL       -->   .466
BE        -->   .460
THEN      -->   .432
UP        -->   .432
THERE     -->   .431
NOT       -->   .431
HER       -->   .413
WERE      -->   .408
OUT       -->   .404
```

FIGURE 14.3
The WordFrequencies project demonstrates how to use the HashTable collection.

The application maintains a single HashTable collection, the `Words` collection, and it updates this collection rather than counting word occurrences from scratch for each file you open. The Frequency Table menu contains the commands to save the words and their counts to a disk file and read the same data from the file. The commands in this menu can store the data either to a text file (Save XML/Load XML commands) or to a binary file (Save Binary/Load Binary commands). Use these commands to store the data generated in a single session, load the data in a later session, and process more files.

The WordFrequencies application uses techniques and classes we haven't discussed yet. The topic of serialization is discussed in detail in Chapter 16, ''XML and Object Serialization,'' whereas the topic of reading from (or writing to) files is discussed in Chapter 15, ''Accessing Folders and Files.'' You don't really have to understand the code that opens a text file and reads its lines; just focus on the segments that manipulate the items of the HashTable.

To test the project, I used some large text files I downloaded from the Project Gutenberg website (`http://promo.net/pg/`). This site contains entire books in electronic format (plain text files), and you can borrow some files to test any program that manipulates text. (Choose some titles you will also enjoy reading.)

The code reads the text into a string variable and then it calls the `Split` method of the String class to split the text into individual words. The `Split` method uses the space, comma, period, quotation mark, exclamation mark, colon, semicolon, and new-line characters as delimiters. The individual words are stored in the `Words` array; after this array has been populated, the program goes through each word in the array and determines whether it's a valid word by calling the `IsValidWord()` function. This function returns False if one of the characters in the word is not a letter; strings such as `B2B` or `U2` are not considered proper words. `IsValidWord()` is a custom function, and you can edit it as you wish.

Any valid word becomes a key to the *`WordFrequencies`* HashTable. The corresponding value is the number of occurrences of the specific word in the text. If a key (a new word) is added to the table, its value is set to 1. If the key exists already, its value is increased by 1 via the following `If` statement:

```
If Not WordFrequencies.ContainsKey(word) Then
   WordFrequencies.Add(word, 1)
Else
   WordFrequencies(word) = CType(WordFrequencies(word), Integer) + 1
End If
```

The code that reads the text file and splits it into individual words is shown in Listing 14.6. The code reads the entire text into a string variable, the `txtLine` variable, and the individual words are isolated with the `Split` method of the String class. The *`Delimiters`* array stores the characters that the `Split` method will use as delimiters, and you can add more delimiters depending on the type of text you're processing. If you're counting keywords in program listings, for example, you'll have to add the math symbols and parentheses as delimiters.

Listing 14.6: Splitting a Text File into Words

```
Private Sub bttnRead_Click(...) Handles bttnRead.Click
   OpenFileDialog1.DefaultExt = "TXT"
   OpenFileDialog1.Filter = "Text|*.TXT|All Files|*.*"
   If OpenFileDialog1.ShowDialog() <> _
           Windows.Forms.DialogResult.OK Then Exit Sub
```

```
       Dim str As StreamReader = File.OpenText(OpenFileDialog1.FileName)
       Dim txtLine As String
       Dim Words() As String
       Dim Delimiters() As Char = _
             {CType(" ", Char), CType(".", Char), CType(",", Char), _
              CType("?", Char), CType("!", Char), CType(";", Char), _
              CType(":", Char), Chr(10), Chr(13), vbTab}
       txtLine = str.ReadToEnd
       Words = txtLine.Split(Delimiters)
       Dim uniqueWords As Integer
       Dim iword As Integer, word As String
       For iword = 0 To Words.GetUpperBound(0)
           word = Words(iword).ToUpper
           If IsValidWord(word) Then
               If Not WordFrequencies.ContainsKey(word) Then
                   WordFrequencies.Add(word, 1)
                   uniqueWords += 1
               Else
                   WordFrequencies(word) = _
                          CType(WordFrequencies(word), Integer) + 1
               End If
           End If
       Next
       MsgBox("Read " & Words.Length & " words and found " & _
               uniqueWords & " unique words")
       RichTextBox1.Clear()
    End Sub
```

This event handler keeps track of the number of unique words and displays them in a RichTextBox control. In a document with 90,000 words, it took less than a second to split the text and perform all the calculations. The process of displaying the list of unique words in the RichTextBox control was very fast, too, thanks to the StringBuilder class. The code behind the Show Word Count button (see Listing 14.7) displays the list of words along with the number of occurrences of each word in the text.

LISTING 14.7: Displaying the Count of Each Word in the Text

```
    Private Sub bttnCount_Click(...) Handles bttnCount.Click
       Dim wEnum As IDictionaryEnumerator
       Dim allWords As New System.Text.StringBuilder
       wEnum = WordFrequencies.GetEnumerator
       While wEnum.MoveNext
           allWords.Append(wEnum.Key.ToString & _
                   vbTab & "-->" & vbTab & _
                   wEnum.Value.ToString & vbCrLf)
       End While
       RichTextBox1.Text = allWords.ToString
    End Sub
```

The last button on the form calculates the frequency of each word in the HashTable, sorts the words according to their frequencies, and displays the list. Its code is detailed in Listing 14.8.

LISTING 14.8: Sorting the Words According to Frequency

```
Private Sub bttnShow_Click(...) Handles bttnSort.Click
   Dim wEnum As IDictionaryEnumerator
   Dim Words(WordFrequencies.Count) As String
   Dim Frequencies(WordFrequencies.Count) As Double
   Dim allWords As New System.Text.StringBuilder
   Dim i, totCount As Integer
   wEnum = WordFrequencies.GetEnumerator
   While wEnum.MoveNext
       Words(i) = CType(wEnum.Key, String)
       Frequencies(i) = CType(wEnum.Value, Integer)
       totCount = totCount + Convert.ToInt32(Frequencies(i))
       i = i + 1
   End While
   For i = 0 To Words.GetUpperBound(0)
       Frequencies(i) = Frequencies(i) / totCount
   Next
   Array.Sort(Frequencies, Words)
   RichTextBox1.Clear()
   For i = Words.GetUpperBound(0) To 0 Step -1
       allWords.Append(Words(i) & vbTab & "-->" & _
                           vbTab & Format(100 * Frequencies(i), _
                           "#.000") & vbCrLf)
   Next
   RichTextBox1.Text = allWords.ToString
End Sub
```

Real World Scenario

HANDLING LARGE SETS OF DATA

Incidentally, my first attempt was to display the list of unique words in a ListBox control. The process was incredibly slow. The first 10,000 words were added in a couple of seconds, but as the number of items increased, the time it took to add them to the control increased exponentially (or so it seemed). Adding thousands of items to a ListBox control is a very slow process. You can call the `BeginUpdate`/`EndUpdate` methods, but they won't help a lot. It's likely that sometimes a seemingly simple task will turn out to be detrimental to your application's performance.

You should try different approaches but also consider a total overhaul of your user interface. Ask yourself this: Who needs to see a list with 10,000 words? You can use the application to do the calculations and then retrieve the count of selected words, display the 100 most common ones, or even display 100 words at a time. I'm displaying the list of words because this is a demonstration, but a real application shouldn't display such a long list. The core of the application counts unique words in a text file, and it does it very efficiently.

Even if you decide to display an extremely long list of items on your interface, you should perform some worst-case scenarios (that is, attempt to load the control with too many items), and if this causes serious performance problems, consider different controls. I've decided to append all the items to a `StringBuilder` variable and then display this variable in a RichTextBox control. I could have used a plain TextBox control — after all, I'm not formatting the list of words and their frequencies — but the RichTextBox allowed me to specify the absolute tab positions. The tab positions of the TextBox control are fixed and weren't wide enough for all words.

The SortedList Collection

The SortedList collection is a combination of the Array and HashTable classes. It maintains a list of items that can be accessed either with an index or with a key. When you access items by their indices, the SortedList behaves just like an ArrayList; when you access items by their keys, the SortedList behaves like a HashTable. What's unique about the SortedList is that this collection is always sorted according to the keys. The items of a SortedList are always ordered according to the values of their keys, and there's no method for sorting the collection according to the values stored in it.

To create a new SortedList collection, use a statement such as the following:

```
Dim sList As New SortedList
```

As you might have guessed, this collection can store keys that are of the base data types. If you want to use custom objects as keys, you must specify an argument of the IComparer type, which tells VB how to compare the custom items. This information is crucial; without it, the SortedList won't be able to maintain its items sorted. You can still store items in the SortedList, but they will appear in the order in which they were added. This form of the SortedList constructor has the following syntax, where ***`comparer`*** is the name of a custom class that implements the IComparer interface (which is discussed in detail later in this chapter):

```
Dim sList As New SortedList(New comparer)
```

There are also two more forms of the constructor, which allow you to specify the initial capacity of the SortedList collection, as well as a Dictionary object, whose data (keys and values) will be automatically added to the SortedList.

Like the other two collections examined in this chapter, the SortedList collection supports the `Capacity` and `Count` properties. To add an item to a SortedList collection, use the `Add` method, whose syntax is the following, where ***`key`*** is the key of the new item and *`item`* is the item to be added:

```
sList.Add(key, item)
```

Both arguments are objects. But remember, if the keys are objects, the collection won't be automatically sorted; you must provide your own comparer, as discussed later in this chapter. The `Add` method is the only way to add items to a SortedList collection, and all keys must be unique; attempting to add a duplicate key will throw an exception.

The SortedList class also exposes the ContainsKey and ContainsValue methods, which allow you to find out whether a key or item already exists in the list. To add a new item, use the following statement to make sure that the key isn't in use:

```
If Not sList.ContainsKey(myKey) Then
   sList.Add(myKey, myItem)
End If
```

It's okay to store duplicate values in the same SortedList collection, but you can still detect the presence of an item in the list via a similar If statement.

To replace an existing item, use the SetByIndex method, which replaces the value at a specific index. The syntax of the method is the following, where the first argument is the index at which the value will be inserted, and *item* is the new item to be inserted in the collection:

```
sList.SetByIndex(index, item)
```

This object will replace the value that corresponds to the specified index. The key, however, remains the same. There's no equivalent method for replacing a key; you must first remove the item and then insert it again with its new key.

To remove items from the collection, use the Remove and RemoveAt methods. The Remove method accepts a key as an argument and removes the item that corresponds to that key. The RemoveAt method accepts an index as an argument and removes the item at the specified index. To remove all the items from a SortedList collection, call its Clear method. After clearing the collection, you should also call its TrimToSize method to restore its capacity to the default size (16).

VB 2008 at Work: The SortedList Project

Let's build a SortedList and print out its elements (this section's sample project is the SortedList project). Listing 14.9 declares the sList SortedList and then adds 10 items to the collection. The keys are integers, and the values are strings. The items are added in no specific order. However, as soon as they're added, they're inserted at the proper location in the collection, so that their keys are in ascending order.

Create a new project, place a button on its form, the Show Keys and Values button, and enter the statements of Listing 14.9 in its Click event handler.

LISTING 14.9: Populating a Simple SortedList

```
Private Sub bttnShow_Click(...) Handles bttnShow.Click
    Dim sList As New System.Collections.SortedList
    sList.Add(116, "item 3"): sList.Add(110, "item 9")
    sList.Add(115, "item 4"): sList.Add(217, "item 2")
    sList.Add(211, "item 8"): sList.Add(214, "item 5")
    sList.Add(318, "item 1"): sList.Add(312, "item 7")
    sList.Add(319, "item 0"): sList.Add(313, "item 6")

    Dim SLEnum As IDictionaryEnumerator
    SLEnum = sList.GetEnumerator()
```

```
        ListBox1.Items.Clear()
        ListBox1.Items.Add("The HashTable's Keys and Values")
        While SLEnum.MoveNext
            ListBox1.Items.Add("Key = " & _
                 SLEnum.Key.ToString & ", Value= " & _
                 SLEnum.Value.ToString)
        End While
        ListBox1.Items.Add("The HashTable's Values by Index")
        Dim idx As Integer
        For idx = 0 To sList.Count - 1
            ListBox1.Items.Add("Item " & _
                 sList.GetByIndex(idx).ToString & _
                 " is at location " & _
                 sList.IndexOfValue( sList.GetByIndex(idx)).ToString)
        Next
        ListBox1.Items.Add("The HashTable's Keys by Index")
        For idx = 0 To sList.Count - 1
            ListBox1.Items.Add("The key at location " & _
                 idx.ToString & " is " & _
                 sList.GetKey(idx).ToString)
        Next
    End Sub
```

The first segment of the code populates the collection, and the second segment of the code prints all the key-value pairs in the order in which the enumerator retrieves them. The *enumerator* is the built-in mechanism for scanning a collection's items (it will be discussed in detail later in this chapter).

If you execute these statements, they will produce the following output:

```
Key = 110, Value= item 9
Key = 115, Value= item 4
Key = 116, Value= item 3
Key = 211, Value= item 8
Key = 214, Value= item 5
Key = 217, Value= item 2
Key = 312, Value= item 7
Key = 313, Value= item 6
Key = 318, Value= item 1
Key = 319, Value= item 0
```

The items are sorted according to their keys, regardless of the order in which they were inserted into the collection.

Working with Keys and Values

Let's look now at a few methods for extracting keys and values. To find out the index of a value in the SortedList, use the `IndexOfValue` method, which accepts an object as an argument. If the object exists in the collection, it returns its index; if not, it returns the value −1. If the same value appears more than once in the collection, the `IndexOfValue` property will return the first instance of the value. Notice that the IndexOfValue property performs a case-sensitive search when applied to strings. The following statement will return the index 6 (the item you're looking for is in the seventh place in the original SortedList):

```
Debug.WriteLine(sList.IndexOfValue("item 7"))
```

You can also find out the index of a specific key, with the IndexOfKey method, whose syntax is similar. Instead of a value, it locates a key. The following statement will return the index 3 (the key you're looking for is in the fourth place in the SortedList):

```
Debug.WriteLine(sList.IndexOfKey(211))
```

If either the key or the value you're searching for can't be found, the IndexOfKey and IndexOfValue methods will return −1.

The GetKey and GetValue methods allow you to retrieve the index that corresponds to a specific key or value in the SortedList. Both methods accept an object as an argument and return an index.

Finally, you can combine the two methods to retrieve the key that corresponds to a value with a statement like the following one:

```
Debug.WriteLine( _
          sList.GetKey(sList.IndexOfValue("item 7")))
```

This statement will print the value 12, based on the contents of the sList collection in Listing 14.9.

You can retrieve the keys in a SortedList collection and create another list by using the GetKeyList method. Likewise, the GetValueList method returns all the values in the SortedList. The following code extracts the keys from the sList SortedList and stores them in the keys list. Then, it scans the list with the help of the key variable and prints all the keys:

```
Dim keys As IList
keys = slist.GetKeyList()
Dim key As Integer
For Each key In Keys
   Debug.WriteLine(key)
Next
```

You can also extract both the keys and the values from a SortedList and store them in an ArrayList, as shown here:

```
Dim AllKeys As New ArrayList()
AllKeys.InsertRange(0, sList.GetValueList)
```

Each item is stored at a specific location in the SortedList, and you can find the location of each item via a loop like the following:

```
Dim idx As Integer
For idx = 0 To sList.Count - 1
```

```
   Debug.WriteLine("ITEM: " & sList.GetByIndex(idx).ToString & _
                   " is at location " & idx.Tostring)
Next
```

The partial output produced by this code segment is this:

```
ITEM: item 9 is at location 0
ITEM: item 4 is at location 1
ITEM: item 3 is at location 2
ITEM: item 8 is at location 3
```

You can also find the location of each key by using a loop like the following:

```
For idx = 0 To sList.Count - 1
   Debug.WriteLine("The key at location " & idx.ToString & " is " & _
                    sList.GetKey(idx).ToString)
   Next
```

The partial output produced by the preceding code segment is as follows:

```
The key at location 0 is 110
The key at location 1 is 115
The key at location 2 is 116
The key at location 3 is 211
```

Notice that the keys are rearranged as they're added to the list, and they're always physically sorted; you can't assume that an element's position remains the same in the course of the application.

Remember the WordFrequencies project we built earlier to demonstrate the use of the HashTable class? Change the declaration of the `WordFrequencies` variable from HashTable to SortedList, and the project will work as before. The only difference is that the words will appear in the RichTextBox control sorted alphabetically when you click the Show Word Count button.

Other Collections

The System.Collections class exposes a few more collections, including the Queue and the Stack collections. The main characteristic of these two collections is how you add and remove items to them. When you add items to a Queue collection, the items are appended to the collection. When you remove items, they're removed from the top of the collection. Queues are known as *last in, first out (LIFO)* structures because you can extract only the oldest item in the queue. You'd use this collection to simulate the customer line in a bank or a production line.

The Stack collection inserts new items at the top, and you can remove only the top item. The Stack collection is a *first in, first out (FIFO)* structure. You'd use this collection to emulate the stack maintained by the CPU, one of the most crucial structures for the operating system and applications alike. Stack and Queue collections are used heavily in computer science but hardly ever in business applications, so I won't discuss them further in this book.

The IEnumerator and IComparer Interfaces

IEnumerator and IComparer are two classes that unlock some of the most powerful features of collections. The proper term for IEnumerator and IComparer is *interface*, a term I will describe shortly. Every class that implements the IEnumerator interface is capable of retrieving a list of pointers for all the items in a collection, and you can use this list to iterate through the items in a collection. Every collection has a built-in enumerator, and you can retrieve it by calling its `GetEnumerator` method. Every class that implements the IComparer interface exposes the `Compare` method, which tells the compiler how to compare two objects of the same type. After the compiler knows how to compare the objects, it can sort a collection of objects with the same type.

The IComparer interface consists of a function that compares two items and returns a value indicating their order (which one is the smaller item or whether they're equal). The Framework can't compare objects of all types; it knows only how to compare the base types. It doesn't even know how to compare built-in objects such as two rectangles or two color objects. If you have a collection of colors, you might want to sort them according to their luminance, saturation, brightness, and so on. Rectangles can be sorted according to their area or perimeter. The Framework can't make any assumptions as to how you might wish to sort your collection and, of course, it doesn't expose members to sort a collection in all possible ways. Instead, it gives you the option to specify a function that compares two colors (or two objects of any other type, for that matter) and uses this function to sort the collection. The same function is used by the `BinarySearch` method to locate an item in a sorted collection. In effect, the IComparer interface consists of a single function that knows how to compare two specific custom objects.

So, what is an interface? An *interface* is another term in object-oriented programming that describes a very simple technique. When we write the code for a class, we might not know how to implement a few operations, but we do know that they'll have to be implemented later. We insert a placeholder for these operations (one or more function declarations) and expect that the application that uses the class will provide the actual implementation of these functions. All collections expose a `Sort` method, which sorts the items in the collection by comparing them to one another. To do so, the `Sort` method calls a function that compares two items and returns a value indicating their relative order. Custom objects must provide their own comparison function — or more than a single function, if you want to sort them in multiple ways. Because you can't edit the collection's `Sort` method code, you must supply your comparison function through a mechanism that the class can understand. This is what the IComparer interface is all about.

Enumerating Collections

All collections expose the `GetEnumerator` method. This method returns an object of the IEnumerator type, which allows you to iterate through the collection without having to know anything about its items, not even the count of the items. To retrieve the enumerator for a collection, call its `GetEnumerator` method by using a statement like the following:

```
Dim ALEnum As IEnumerator
ALEnum = aList.GetEnumerator
```

The IEnumerator class exposes two methods: the `MoveNext` and `Reset` methods. The `MoveNext` method moves to the next item in the collection and makes it the current item (property `Current`). When you initialize the IEnumerator object, it's positioned in front of the very first item, so you must call the `MoveNext` method to move to the first item. The `Reset` method does exactly the same thing: It repositions the IEnumerator in front of the first element.

The `MoveNext` method doesn't return an item, as you might expect. It returns a True/False value that indicates whether it has successfully moved to the next item. After you have reached the end of the collection, the `MoveNext` method will return False. Here's how you can enumerate through an ArrayList collection by using an enumerator:

```
Dim aItems As IEnumerator
aItems = aList.GetEnumerator
While aItems.MoveNext
   { process item aItems.Current}
End While
```

At each iteration, the current item is given by the `Current` property of the enumerator. After you reach the last item, the `MoveNext` method will return False, and the loop will terminate. To rescan the items, you must reset the enumerator by calling its `Reset` method.

To process the current item, you can call its methods through the `aItems.Current` object. Because the `Current` property is an object, you must first cast it to the appropriate type. If the collection holds Rectangles, for example, you can access their sizes by using these expressions:

```
CType(aItems.Current, Rectangle).Width
CType(aItems.Current, Rectangle).Height
```

The Strict option necessitates the explicit conversion of the `Current` property to a Rectangle object. In other words, you can't use an expression such as `aItems.Current.Width` with the Strict option on (which is the recommended setting for this option).

The event handler in Listing 14.10 populates an ArrayList with Rectangle objects and then iterates through the collection and prints the area of each Rectangle.

LISTING 14.10: Iterating an ArrayList with an Enumerator

```
Dim aList As New ArrayList()
Dim R1 As New Rectangle(1, 1, 10, 10)
aList.Add(R1)
R1 = New Rectangle(2, 2, 20, 20)
aList.Add(R1)
aList.add(New Rectangle(3, 3, 2, 2))
Dim REnum As IEnumerator
REnum = aList.GetEnumerator
Dim R As Rectangle()
While REnum.MoveNext
   R = CType(REnum.Current, Rectangle)
   Debug.WriteLine((R.Width * R.Height).ToString)
End While
```

The *REnum* variable is set up and used to iterate through the items of the collection. At each iteration, the code saves the current Rectangle to the `R` variable, and it uses this variable to access the properties of the Rectangle object (its width and height).

Of course, you can iterate a collection without the enumerator, but with a `For Each...Next` loop. To iterate through a HashTable, you can use either the `Keys` or the `Values` collection. The code shown in Listing 14.11 populates a HashTable with Rectangle objects. Then it scans the items and prints their keys, which are strings, and the area of each rectangle.

Listing 14.11: Iterating a HashTable with Its Keys

```
Dim hTable As New HashTable()
Dim r1 As New Rectangle(1, 1, 10, 10)
hTable.Add("R1", r1)
r1 = New Rectangle(2, 2, 20, 20)
hTable.Add("R2", r1)
hTable.add("R3", New Rectangle(3, 3, 2, 2))
Dim key As Object
Dim R As Rectangle
For Each key In hTable.keys
   R = CType(hTable(key), Rectangle)
   Debug.WriteLine(String.Format( _
           "The area of Rectangle {0} is {1}", _
            key.ToString, R.Width * R.Height))
Next
```

The code adds three Rectangle objects to the HashTable and then iterates through the collection using the `Keys` properties. Each item's key is a string (`R1`, `R2`, and `R3`). The `Keys` property is itself a collection and can be scanned with a `For Each...Next` loop. At each iteration, we access a different item through its key with the expression `hTable(key)`. The output produced by this code is shown here:

```
The area of Rectangle R1 is 100
The area of Rectangle R3 is 4
The area of Rectangle R2 is 400
```

Alternatively, you can iterate a HashTable with an enumerator, but be aware that the `GetEnumerator` method of the HashTable collection returns an object of the IDictionary-Enumerator type, not an IEnumerator object. The IDictionaryEnumerator class is quite similar to the IEnumerator class, but it exposes additional properties. They are the `Key` and `Value` properties, and they return the current item's key and value. The IDictionaryEnumerator class also exposes the `Entry` property, which contains both the key and the value. You can access the current item's key and value either as `DEnum.Key` and `DEnum.Value`, or as `DEnum.Entry.Key` and `DEnum.Entry.Value.` The *`DEnum`* variable is a properly declared enumerator for the HashTable:

```
Dim DEnum As IDictionaryEnumerator
```

Assuming that you have populated the `hTable` collection with the same three Rectangle objects, you can use the statements in Listing 14.12 to iterate through the collection's items.

LISTING 14.12: Iterating a HashTable with an Enumerator

```
Dim hEnum As IDictionaryEnumerator
hEnum = hTable.GetEnumerator
While hEnum.MoveNext
   Debug.WriteLine( _
            String.Format("The area of rectangle " & _
            "{0} is {1}", hEnum.Key, _
            CType(hEnum.Value, Rectangle).Width * _
            CType(hEnum.Value, Rectangle).Height))
End While
```

If you execute these statements after populating the HashTable collection with three Rectangles, they will produce the same output as Listing 14.11.

The Enumerations project shows how to iterate through an ArrayList and a HashTable with and without an enumerator. The code should be quite familiar to you by now, so I do not list it here. You can open the project and examine its code and routines.

Custom Sorting

The `Sort` method allows you to sort collections, as long as the items are of the same base data type. If the items are objects, however, the collection doesn't know how to sort them. If you want to sort objects, you must help the collection a little by telling it how to compare the objects. A sorting operation is nothing more than a series of comparisons. Sorting algorithms compare items and swap them if necessary.

All the information needed by a sorting algorithm to operate on an item of any type is a function that compares two objects. Let's say you have a list of persons, and each person is a structure that contains names, addresses, e-addresses, and so on. The System.Collections class can't make any assumptions as to how you want your list sorted. This collection can be sorted by any field in the structure (names, e-addresses, postal codes, and so on).

The comparer is implemented as a separate class, outside all other classes in the project, and is specific to a custom data type. Let's say you have created a custom structure for storing contact information. The Person object is declared as a structure with the following fields:

```
Structure Person
   Dim Name As String
   Dim BDate As Date
   Dim EMail As String
End Structure
```

You'll probably build a class to represent persons, but I'm using a Structure to simplify the code. To add an instance of the Person object to an ArrayList or HashTable, create a variable of Person type, initialize its fields, and then add it to the `aList` ArrayList via the `Add` method. This collection can't be sorted with the simple forms of the `Sort` method because the compiler doesn't know how to compare two Person objects. You must provide your own function for comparing two variables of the Person type. After this function is written, the compiler can compare items and therefore sort the collection. This custom function, however, can't be passed to the `Sort` and `BinarySearch` methods by name. You must create a new class that implements the IComparer interface and pass an IComparer object to the two methods.

Implementing the IComparer Interface

Here's the outline of a class that implements the IComparer interface:

```
Class customComparer : Implements IComparer
   Public Function Compare( _
              ByVal o1 As Object, ByVal o2 As Object) _
              As Integer Implements IComparer.Compare
      { function's code }
   End Function
End Class
```

The name of the class can be anything. It should be a name that indicates the type of comparison it performs or the type of objects it compares. After the class declaration, you must specify the interface implemented by the class. As soon as you type the first line of the preceding code segment, the editor will insert automatically the stub of the Compare function. The name of the custom function must be `Compare` and it must implement the IComparer.Compare interface. The interface declares a placeholder for a function, whose code must be provided by the developer.

Let's get back to our example. To use the custom function, you must create an object of the `customComparer` type (or whatever you have named the class) and then pass it to the `Sort` and `BinarySearch` methods as an argument:

```
Dim CompareThem As New customComparer
aList.Sort(CompareThem)
```

You can combine the two statements in one by initializing the `customComparer` variable in the line that calls the `Sort` method:

```
aList.Sort(New customComparer)
```

You can also use the equivalent syntax of the `BinarySearch` method to locate a custom object that implements its own IComparer interface:

```
aList.BinarySearch(object, New customComparer)
```

This is how you can use a custom function to compare two objects. Everything is the same, except for the name of the comparer, which is different every time.

The last step is to implement the function that compares the two objects and returns an integer value, indicating the order of the elements. This value should be −1 if the first object is smaller than the second object, 0 if the two objects are equal, and 1 if the first object is larger than the second object. *Smaller* here means that the element appears before the larger one when sorted in ascending order. Listing 14.13 is the function that sorts Person objects according to the `BDate` field. The sample code for this and the following section comes from the CustomComparer project.

The main form contains a single button, which populates the collection and then prints the original collection, the collection sorted by name, and the collection sorted by birth date.

Listing 14.13: A Custom Comparer

```
Class PersonAgeComparer : Implements IComparer
  Public Function Compare( _
      ByVal o1 As Object, ByVal o2 As Object) _
      As Integer Implements IComparer.Compare
     Dim person1, person2 As Person
     Try
        person1 = CType(o1, Person)
        person2 = CType(o2, Person)
     Catch compareException As system.Exception
        Throw (compareException)
        Exit Function
     End Try
     If person1.BDate < person2.BDate Then
        Return -1
     Else
        If person1.BDate > person2.BDate Then
           Return 1
        Else
           Return 0
        End If
     End If
  End Function
End Class
```

The code could have been considerably simpler, but I'll explain momentarily why the `Try` statement is necessary. The comparison takes place in the `If` statement. If the first person's birth date is chronologically earlier than the second person's, the function returns the value −1. If the first person's birth date is chronologically later than the second person's, the function returns 1. Finally, if the two values are equal, the function returns 0.

The code is straightforward, so why the error-trapping code? Before we perform any of the necessary operations, we convert the two objects into Person objects. It's not unthinkable that the collection with the objects you want to sort contains objects of different types. If that's the case, the `CType()` function won't be able to convert the corresponding argument to the Person type, and the comparison will fail. The same exception that would be thrown in the function's code is raised again from within the error handler, and it's passed back to the calling code.

Implementing Multiple Comparers

The Person objects can be sorted in many ways. You might wish to sort them by ID, name, and so on. To accommodate multiple sorts, you must implement several classes, each one with a different `Compare` function. Listing 14.14 shows two classes that implement two different `Compare` functions for the Person class. The PersonNameComparer class compares the names, whereas the

PersonAgeComparer class compares the ages. Both classes, however, implement the IComparer interface.

LISTING 14.14: A Class with Two Custom Comparers

```
Class PersonNameComparer : Implements IComparer
  Public Function Compare( _
            ByVal o1 As Object, ByVal o2 As Object) _
            As Integer Implements IComparer.Compare
     Dim person1, person2 As Person
     Try
        person1 = CType(o1, Person)
        person2 = CType(o2, Person)
     Catch compareException As system.Exception
        Throw (compareException)
        Exit Function
     End Try
     If person1.Name < person2.Name Then
        Return -1
     Else
        If person1.Name > person2.Name Then
           Return 1
        Else
           Return 0
        End If
     End If
  End Function
End Class

Class PersonAgeComparer : Implements IComparer
  Public Function Compare( _
            ByVal o1 As Object, ByVal o2 As Object) _
            As Integer Implements IComparer.Compare
     Dim person1, person2 As Person
     Try
        person1 = CType(o1, Person)
        person2 = CType(o2, Person)
     Catch compareException As system.Exception
        Throw (compareException)
        Exit Function
     End Try
     If person1.BDate > person2.BDate Then
        Return -1
     Else
        If person1.BDate < person2.BDate Then
           Return 1
        Else
```

```
          Return 0
        End If
      End If
    End Function
  End Class
```

To test the custom comparers, create a new application and enter the code of Listing 14.14 (the two classes) in a separate Class module. Don't forget to include the declaration of the Person structure. Then place a button on the form and enter the code of Listing 14.15 in its `Click` event handler. This code adds three persons with different names and birth dates to an ArrayList.

Listing 14.15: Testing the Custom Comparers

```
Private Sub Button1_Click(...) Handles Button1.Click
  Dim aList As New ArrayList()
  Dim p As Person
  ' Populate collection
  p.Name = "C Person"
  p.EMail = "PersonC@sybex.com"
  p.BDate = #1/1/1961#
  If Not aList.Contains(p) Then aList.Add(p)
  p.Name = "A Person"
  p.EMail = "PersonA@sybex.com"
  p.BDate = #3/3/1961#
  If Not aList.Contains(p) Then aList.Add(p)
  p.Name = "B Person"
  p.EMail = "PersonB@sybex.com"
  p.BDate = #2/2/1961#
  If Not aList.Contains(p) Then aList.Add(p)
  ' Print collection as is
  Dim PEnum As IEnumerator
  PEnum = aList.GetEnumerator
  ListBox1.Items.Add("Original Collection")
  While PEnum.MoveNext
     ListBox1.Items.Add( _
          CType(PEnum.Current, Person).Name & _
          vbTab & CType(PEnum.Current, Person).BDate)
  End While
  ' Sort by name, then print collection
  ListBox1.Items.Add(" ")
  ListBox1.Items.Add("Collection Sorted by Name")
  aList.Sort(New PersonNameComparer())
  PEnum = aList.GetEnumerator
  While PEnum.MoveNext
     ListBox1.Items.Add( _
          CType(PEnum.Current, Person).Name & _
```

```
                vbTab & CType(PEnum.Current, Person).BDate)
        End While
        ' Sort by age, then print collection
        ListBox1.Items.Add(" ")
        ListBox1.Items.Add("Collection Sorted by Age")
        aList.Sort(New PersonAgeComparer())
        PEnum = aList.GetEnumerator
        While PEnum.MoveNext
           ListBox1.Items.Add( _
                CType(PEnum.Current, Person).Name & _
                vbTab & CType(PEnum.Current, Person).BDate)
        End While
    End Sub
```

The four sections of the code are delimited by comments in the listing. The first section populates the collection with three variables of the Person type. The second section prints the items in the order in which they were added to the collection:

```
C Person   1/1/1961
A Person   3/3/1961
B Person   2/2/1961
```

The third section of the code calls the `Sort` method, passing the `PersonNameComparer` custom comparer as an argument, and it again prints the contents of the ArrayList. The names are listed now in alphabetical order:

```
A Person   3/3/1961
B Person   2/2/1961
C Person   1/1/1961
```

In the last section, it calls the `Sort` method again — this time to sort the items by age — and prints them:

```
C Person   1/1/1961
B Person   2/2/1961
A Person   3/3/1961
```

It is straightforward to write your own custom comparers and sort your custom object in any way that suits your application. Custom comparisons might include more-complicated calculations, not just comparisons. For example, you can sort Rectangles by their area, color values by their hue or saturation, and customers by the frequency of their orders.

Custom Sorting of a SortedList

The items of a SortedList are sorted according to their keys. Of course, the SortedList cannot maintain the order of the keys unless the keys are of a base type, such as integers or strings. If you need to use objects as keys, you must simply provide a function to implement the IComparer interface, as you know well by now, and pass the name of the class that implements the interface to the constructor of the SortedList class.

Generic Collections

All collections we examined so far were designed to store objects because they should accommodate all data types, built-in or custom. In most practical situations, however, we want to make sure that collections store elements of the same type, just like arrays that were declared with a specific type. Why not be able to declare collections that can store only variables of a specific type? The obvious advantage of such a collection is performance because we wouldn't have to convert variables of the Object type to and from other more-specific types. This can have a substantial performance penalty. You can still process the ArrayList's items without casting them to their corresponding type, but then you give up the benefits of type-safe programming. If your code calls a member that's not supported by the specific object, the compiler won't catch the error and an exception will be thrown when the statement is executed.

Let's say you created the `Rects` ArrayList and you plan to store Rectangle objects in it. There's no mechanism to prevent you from storing a Color object in this ArrayList. If the Color object was stored in the collection by mistake, it's possible (if not likely) that you would attempt to treat it later as a Rectangle object. An attempt to request the `Width` property of a Color object will throw an exception. The compiler can't catch this error at design time because collections are not typed.

If we can turn a collection into a typed collection, we'll be able to use its items without casting them to a different type, and the collection itself would reject items of any other type. This is what a *generic collection* does: It is a typed collection that allows you to specify the type of objects you'll store to the collection when you declare it. The generic collections are implemented in the System.Collection.Generic namespace; they're equivalent to the regular collections but have different names.

Let's consider a collection for storing employees. Each employee is defined with the following class:

```
Public Class Employee
    Public ID As Long
    Public Title As String
    Public LastName As String
    Public FirstName As String
    Public SSN As String
    Public HiredOn As Date
    Public ReportsTo As Integer
End Class
```

One of the generic collections of the Framework is the List collection, which is similar to the ArrayList. The List collection provides the same functionality as the ArrayList collection, but it stores objects of the same type only: the type specified in the collection's declaration. The following statement declares a List collection for storing Employee objects:

```
Dim LEmployees As New List(Of Employee)
```

The `Of` keyword in the constructor of the collection is followed by a data type. The `LEmployees` collection won't accept elements of any other type, other than the Employee type. The List collection is also strongly typed, and you can use expressions such as the following:

```
LEmployees(0).SSN
```

This expression is early bound and as soon as you enter the dot following an element, the IntelliSense box will display the members of the Employee type.

A better example is that of a Dictionary collection. If you store the employees to a Dictionary and use the employee's SSN as a key, you must declare the type of both the keys and the items you want to store to the Dictionary collection. In this case, the `Of` keyword will be followed by two types: one for the keys and another one for the elements (I'm assuming that you have imported the System.Collections.Generic namespace):

```
Dim Employees As New Dictionary(Of String, Employee)
```

This form of the declaration tells the compiler that the keys of the Dictionary will be strings, and the values will be Employee objects. To add the `newEmployee` object to the `Employees` Dictionary, use a few statements like the following:

```
Dim Emp As Employee
Emp.SSN = "334-19-0020"
Employees.Add(Emp.SSN, Emp)
```

After the collection has been populated, you can access the items of the collection by their keys with an expression like this one:

```
Employees("334-19-0020")
```

The employee's name is given by the following expression:

```
Employees("334-19-0020").LastName & "," & _
Employees("334-19-0020").FirstName
```

You can do the same with a regular Dictionary (a nongeneric dictionary), but the editor won't prevent you from requesting a nonexistent member such as FullName with a late-bound expression. To take advantage of type-safe programming, you must cast the items to their proper types. The reason for including generic collections in the Framework is performance, because the compiler generates code optimized for the type of objects you store in the collection. If you store the same collection of Employee objects to an ArrayList and a List collection, the ArrayList is at least five times slower than the List. The most important benefit of generic collections, however, is that they enable type-safe programming.

As far as the members of the generic collections are concerned, they're the same as the members of the corresponding regular collections. The Dictionary generic collection, for example, is the typed version of the HashTable collection and it exposes the `ContainsKey` and `ContainsValue` methods, the `Keys` and `Values` properties to iterate through the items of the collection as described earlier in this chapter, the `Add` and `Remove` methods to manipulate its contents, and so on. The same is true for the List collection, which is the typed version of the ArrayList collection and it allows you to find specific items by using the `FindIndex`, `FindIndexLast`, and `BinarySearch` methods; sort its items via the `Sort` method; extract items via the `CopyTo` and `ToArray` methods; and so on. If the collection stores objects and not value types, you must pass to the `Sort` and `BinarySearch` methods the appropriate comparer.

The generic collections are implemented in the System.Collections.Generic namespace, and there are quite a few generic collections. However, not all collections implemented by the

Framework have a generic counterpart. You can create generic Dictionary (equivalent to the HashTable untyped collection), SortedDictionary (it's a typed Dictionary sorted by its keys), and List (it's equivalent to the untyped ArrayList) collections. Note that the generic collections don't have the same name as their regular counterparts, because if they did, you'd have to fully qualify their names in your code.

My recommendation is to use generic collections, which allow you to write strongly typed code. If you need a collection for storing elements of different types, use one of the regular collections. Eventually, all collections should be replaced by generic collections, and the usual collections will become special cases of generic collections. You can actually declare a generic collection for storing variables of the Object type.

In this chapter, we discussed how to use collections to store data in memory. To make the most of collections, however, you should be able to store them to disk files and read them back at a later session. This topic is covered a little later in the book, in Chapter 16.

The Bottom Line

Make the most of arrays. The simplest method of storing sets of data is to use arrays. They're very efficient and they provide methods to perform advanced operations such as sorting and searching their elements. Use the `Sort` method of the Array class to sort an array's elements. To search for an element in an array, use the `IndexOf` and `LastIndexOf` methods, or the `Binary-Search` method if the array is sorted. The `BinarySearch` method always returns an element's index, which is a positive value for *exact matches* and a negative value for *near matches*.

Master It Explain how you can search an array and find exact and near matches.

Store data in specialized collections such as ArrayLists and HashTables. In addition to arrays, the Framework provides collections, which are dynamic data structures. The most commonly used collections are the ArrayList and the HashTable. ArrayLists are similar to arrays, but they're dynamic structures. ArrayLists store lists of items, whereas HashTables store key-value pairs and allow you to access their elements via a key. You can add elements by using the `Add` method and remove existing elements by using the `Remove` and `RemoveAt` methods.

HashTables provide the `ContainsKey` and `ContainsValue` methods to find out whether the collection already contains a specific key or value, and the `GetKeys` and `GetValues` methods to retrieve all the keys and values from the collection, respectively.

Master It How will you populate a HashTable with a few pairs of keys/values and then iterate though the collection's items?

Sort and search collections. Collections provide the `Sort` method for sorting their items and several methods to locate items: `IndexOf`, `LastIndexOf`, and `BinarySearch`. Both sort and search operations are based on comparisons, and the Framework knows how to compare values types only (Integers, Strings, and the other primitive data types). If a collection contains objects, you must provide a custom function that knows how to compare two objects of the same type.

Master It How do you specify a custom comparer function for a collection that contains Rectangle objects?

Chapter 15

Accessing Folders and Files

Files have always been an important aspect of programming. We use files to store data, and in many cases we have to manipulate files and folders from within applications. I need not give examples: Just about any application that allows user input must store its data to a file (or multiple files) for later retrieval — databases excluded, of course.

Manipulating files and folders is quite common, too. Organizing files into folders and processing files en masse are two typical examples. I recently ran into a few web-related tasks that are worth mentioning here. A program for placing watermarks on pictures was the first. A *watermark* is a graphic that's placed over an image to indicate its origin. The watermark is transparent, so it doesn't obscure the image, but it makes the image unusable on any site other than the original one. You will see how to place a semitransparent graphic on top of an image in Chapter 19, ''Manipulating Images and Bitmaps,'' and with the help of the information in this chapter, you'll be able to scan a folder that has thousands of image files and to automate the process of watermarking the images.

Another example has to do with matching filenames to values stored in a database. Product images are usually named after the product's ID and stored in separate files. There's a need for programs to match product IDs to images, to find out whether there's an image for a specific product in the database, or to simply move the image files around (store the images for different product categories into different folders and so on).

In this chapter, you'll learn how to do the following:

- Handle files with the My object
- Manipulate folders and files
- Save data to a file
- Monitor changes in the file system and react to them

The IO Namespace and the FileSystem Component

To manipulate folders and files, as well as file input/output (I/O) operations, the Framework provides the System.IO namespace. The My object provides My.Computer.FileSystem component, which simplifies the basic file tasks. Obviously, there's an enormous overlap between the two components.

The FileSystem component is a subset of the IO namespace in terms of the functionality it exposes, but it's considerably simpler to use. The My object was designed to simplify some of the most common tasks for the VB developer and, as you may recall from Chapter 1, ''Getting Started with Visual Basic 2008,'' it's a speed-dial into the Framework. You can perform all common file I/O operations with a single line of code. (Okay, sometimes you may need a second line, but you get the idea.) To access the full power of the Framework's I/O capabilities, use the IO namespace. There's nothing you can do with the My object that you can't do with the Framework; the opposite isn't true. The My object was designed to simplify the most common programming tasks, but it's not a substitute for the Framework.

That said, I will start with a brief overview of the My.Computer.FileSystem component and then I'll discuss the IO namespace, which is the whole enchilada. Old VB developers will use the My object to access the file system, because VB is about productivity and the My object is simpler. The Framework, on the other hand, is the core of Windows programming and you shouldn't ignore it.

Using the My.Computer.FileSystem Component

Using the My object, you can write some text to a file via a single statement. The `WriteAllText` method accepts as arguments a path and the string to be written to the file (as well as a third optional argument that determines whether the text will be appended to the file or will replace the current contents), writes some text to the file (the contents of a TextBox control in the following sample), and then closes the file:

```
My.Computer.FileSystem.WriteAllText(fName, TextBox1.Text, True)
```

If the specified file does not exist, the write method creates it. To write binary data to a file, use the `WriteAllBytes` method, whose syntax is almost identical, but the second argument is an array of bytes instead of a string.

By the way, because My is not a class, you can't import it to a file and shorten the statements that access its members; you have to fully qualify the member names. You can still use the `With` statement, as shown here:

```
With My.Computer.FileSystem
    .WriteAllText(fname, TextBox1.Text, True)
End With
```

To read back the data saved with the `WriteAllText` and `WriteAllBytes` methods, use the `ReadAllText` and `ReadAllBytes` methods, respectively. The `ReadAllText` method accepts as an argument the path of a file and returns its contents as a string. `ReadAllBytes` accepts the same argument, but returns the file's contents as an array of bytes. This is all you need to know in order to save data to disk files between sessions with the My object. The following code segment saves the contents of the *`TextBox1`* control to a user-specified file, clears the control, reads the text from the same file, and populates the *`TextBox1`* control:

```
' Set up the SaveFileDialog control
SaveFileDialog1.DefaultExt = "*.txt"
SaveFileDialog1.AddExtension = True
SaveFileDialog1.FileName = ""
SaveFileDialog1.Filter = "Text Files|*.txt|All Files|*.*"
If SaveFileDialog1.ShowDialog =  Windows.Forms.DialogResult.OK Then
' Use the WriteAllText method to save the text
    My.Computer.FileSystem.WriteAllText( _
       SaveFileDialog1.FileName, TextBox1.Text, False)
```

```
End If
' Clear the control
TextBox1.Clear()
' Set up the OpenFileDialog control
OpenFileDialog1.DefaultExt = "txt"
OpenFileDialog1.Filter = "Text Files|*.txt|All Files|*.*"
OpenFileDialog1.FileName = "Test File.txt"
If OpenFileDialog1.ShowDialog = Windows.Forms.DialogResult.OK Then
' Use the ReadAllText method to read back the text
' and display it on the TextBox control
    TextBox1.Text = My.Computer.FileSystem.ReadAllText( _
                       OpenFileDialog1.FileName)
End If
```

As you can see, it takes two statements to send the data to the file and read it back. All other statements set up the Open and Save As dialog boxes.

Here's another example of using the FileSystem object. To delete a folder, call the `DeleteDirectory` method of the My.Computer.FileSystem component, which accepts three arguments: the name of the folder to be deleted, a constant that specifies whether the `DeleteDirectory` method should delete the contents of the specified folder if the folder isn't empty, and another constant that determines whether the folder will be deleted permanently or moved to the Recycle Bin. This constant is a member of the `FileIO.RecycleOption` enumeration: *`DeletePermanently`* (to remove the file permanently from the file system) and *`SendToRecycleBin`* (moves the file to the Recycle Bin). To delete a file, use the `DeleteFile` method, which has the same syntax. (The first argument is the path of a file, not a folder.)

Another interesting member of the FileSystem object is the `SpecialDirectories` property, which allows you to access the special folders on the target computer (folders such as My Documents, the Desktop, the Program Files folder, and so on). Just enter the name of the `SpecialDirectories` property followed by a period to see the names of the special folders in the IntelliSense box. To find out the application's current folder, call the `CurrentDirectory` method.

The `RenameDirectory` and `RenameFile` methods allow you to rename folders and files, respectively. Both methods accept as arguments the original folder name or filename and the new name, and perform the operation. They do not return a value to indicate whether the operation was successful, but they throw an exception if the operation fails.

The `CopyFile` and `CopyDirectory` methods copy a single file and an entire folder, respectively. They accept as arguments the path of the file or folder to be copied, the destination path, and an argument that determines which dialog boxes will be displayed during the copying operation. The value of this argument is a member of the `FileIO.UIOption` enumeration: *`AllDialogs`* (shows the progress dialog box and any error dialog boxes) and *`OnlyErrorDialogs`* (shows only error dialog boxes). The following code segment copies a fairly large folder. It's interesting to see how it displays the usual file copy animation and prompts users every time it can't copy a folder (because the user doesn't have adequate privileges or because a file is locked, and so on).

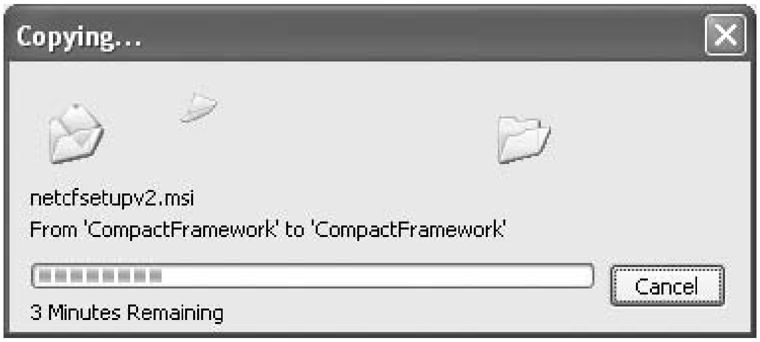
```
Dim dir as String
dir = "C:\Program Files\Microsoft Visual Studio 9.0"
Try
    My.Computer.FileSystem.CopyDirectory( _
              dir, "E:\Copy of " & _
```

```
                  My.Computer.FileSystem.GetName(dir), _
                  Microsoft.VisualBasic.FileIO. _
                  UIOption.AllDialogs, _
                  FileIO.UICancelOption.ThrowException)
    Catch ex As Exception
        MsgBox(ex.Message)
    End Try
```

Please do change the destination drive (`E:` in the preceding sample code segment); you may not have an `E:` drive, or you may overwrite a working installation of Visual Studio 2008.

Notice that I used the `GetName` method of the FileSystem component to extract the last part of the path and then combine it with the new drive name. The last argument of the `CopyDirectory` method, which is a member of the `UICancelOption` enumeration: `DoNothing` or `ThrowException`, determines how the method reacts when the user clicks the Cancel button on the copy animation (see Figure 15.1). I used the `ThrowException` member and embedded the entire statement in an exception handler. If you click the Cancel button while the folder's files are being copied, the following message will appear:

```
The operation was canceled.
```



netcfsetupv2.msi
From 'CompactFramework' to 'CompactFramework'
3 Minutes Remaining

FIGURE 15.1 Copying a large folder by using the `DirectoryCopy` method

Cancelling a copy operation doesn't reset the destination folder. You must insert some additional code to remove the files that have been copied to the destination folder, or notify the user that some files have copied already and they're not automatically removed.

To manipulate folders, use the `CreateDirectory` and `DirectoryExists` methods, which accept as an argument the path of a folder. To find out whether a specific file exists, call the `FileExists` method, passing the file's path as the argument.

To retrieve information about drives, folders, and files, use the `GetDriveInfo`, `GetDirectoryInfo`, and `GetFileInfo` methods, respectively. These methods accept as an argument the name of the drive or the path to a folder/file, respectively, and return the relevant information as an object. Drive properties are described with the IO.DriveInfo class, folder properties are described with the IO.DirectoryInfo class, and file properties with the IO.FileInfo class. These objects are part of the Framework's IO namespace and they provide properties such as a directory's path and attributes, a file's path, size, creation and last modification date, and so on. The three objects are described in detail later in this chapter, in the discussion of the IO namespace. To find out the properties of the `C:` drive on your system, execute a statement such as the following:

```
Dim DI As IO.DriveInfo = _
      My.Computer.FileSystem.GetDriveInfo("C")
    Debug.WriteLine("DRIVE " & DI.Name & vbCrLf & _
     "VOLUME " & DI.VolumeLabel & vbCrLf & _
     "TYPE " & DI.DriveType.ToString & vbCrLf & _
     "TOTAL SIZE " & DI.TotalSize.ToString & vbCrLf & _
     "FREE SPACE " & DI.AvailableFreeSpace.ToString)
```

This statement produced the following output on my system:

```
DRIVE C:\
VOLUME VAIO
TYPE Fixed
TOTAL SIZE 3100019372032
FREE SPACE 50142416896
```

To retrieve information about all drives in your system, call the `Drives` method, which returns a read-only collection of DriveInfo objects. If you want to search a folder for specific files, use the `FindInFiles` method, which is quite flexible. The `FindInFiles` method goes through all files in a specified folder and selects files by a wildcard specification, or by a string in their contents. The method has two overloaded forms; their syntax is the following:

```
FindInFiles(dir, containsText, ignoreCase, FileIO.SearchOption)
```

and

```
FindInFiles(dir, containsText, ignoreCase, _
            FileIO.SearchOption,fileWildCards() String)
```

Both methods return the list of matching files as a read-only collection of strings. The *`dir`* argument is the folder to be searched, and the *`containsText`* argument is the string we want to locate in the files. The *`ignoreCase`* argument is a True/False value that determines whether the search is case-sensitive, and the *`SearchOption`* argument is a member of the `FileIO.SearchOption` enumeration and specifies whether the method will search in the specified folder or will include the subfolders as well: `SearchAllSubdirectories`, `SearchTopLevelOnly`. The second overloaded form of the method accepts an additional argument, which is an array of strings with the patterns to be matched (for example, `*.txt`, `Sales*.doc`, `*.xls`, and so on). The following statements locate all text, `.doc`, and `.xml` files in the Program Files folder that contain the string Visual Basic. The search is case-insensitive and includes the all subfolders under Program Files.

```
Dim patterns() As String = {"*.txt", "*.doc", "*.xml"}
Dim foundFiles As System.Collections.ObjectModel. _
                         ReadOnlyCollection(Of String)
```

```
foundFiles = My.Computer.FileSystem.FindInFiles( _
           "C:\Program Files", "visual basic", True, _
           FileIO.SearchOption.SearchAllSubDirectories, patterns)
Dim file As String
For Each file In foundFiles
    Debug.WriteLine(file)
Next
```

A Simpler Method of Saving Data to Files

The Framework provides an attractive alternative to writing data to files: the serialization mechanism. You can create collections of objects and persist them to a file via a few simple statements. Actually, it's much simpler to create a collection of customer/product/sales data and persist it as a whole, than to write code to write every field to a file (let's not forget the code for reading the data back into the application). Serialization is a major component of .NET, and it's discussed in detail in Chapter 16, ''XML and Object Serialization.''

This concludes the overview of the file-related methods of the FileSystem component. This component doesn't expose many members, and their syntax is quite simple. You can experiment with the methods and properties of the FileSystem component to get a better idea of the type of operations you can perform with it. In the remainder of this chapter, you'll find a detailed discussion of the IO namespace.

Manipulating Folders and Files with the IO Namespace

In this section, you'll learn how to access and manipulate files and folders with the help of the Directory and File classes of the System.IO namespace. The Directory class provides methods for manipulating folders, and the File class provides methods for manipulating files. These two objects allow you to perform just about any of the usual operations on folders and files, respectively, short of storing data into or reading from files. By the way, *directory* is another name for *folder*; the two terms mean the same thing, but *folder* is the more-familiar term in Windows. When it comes to developers and administrators, Microsoft still uses *directory* (the Active Directory, the Directory object, and so on), especially with command-line utilities.

Keep in mind that Directory and File objects don't represent folders or files. Directory and File are shared classes, and you must supply the name of the folder or file they will act upon as an argument to the appropriate method. The two classes that represent folders and files are the DirectoryInfo and FileInfo classes. If you're in doubt about which class you should use in your code, consider that the members of the Directory and File classes are *shared:* You can call them without having to explicitly create an instance of the corresponding object first, and you must supply the name of the folder or file their methods will act upon as an argument. The methods of the DirectoryInfo and FileInfo classes are *instance* methods: Their methods apply to the folder or file represented by the current instance of the class.

Both the Directory and the DirectoryInfo classes allow you to delete a folder, including its subfolders. The `Delete` method of the DirectoryInfo class will act on a directory you specified when you instantiated the class:

```
Dim DI As New System.IO.DirectoryInfo("C:\Work Files\Assignments")
DI.Delete()
```

But you can't call `Delete` on a DirectoryInfo object that you haven't specifically declared. The `DirectoryInfo.Delete` method doesn't accept the name of a folder as an argument. The `Delete` method of the Directory class, on the other hand, deletes the folder passed as an argument to the method:

```
System.IO.Directory.Delete("C:\Work Files\Assignments")
```

The Directory Class

The System.IO.Directory class exposes all the members you need to manipulate folders. Because the Directory class belongs to the System.IO namespace, you must import the IO namespace into any project that might require the Directory object's members with the following statement:

```
Imports System.IO
```

Methods

The Directory object exposes methods for accessing folders and their contents, which are described in the following sections.

CreateDirectory

This method creates a new folder, whose path is passed to the method as a string argument:

```
Directory.CreateDirectory(path)
```

`path` is the path of the folder you want to create and can be either an absolute or a relative path. If it's a relative path, its absolute value is determined by the current drive and path (use the `GetCurrentDirectory` method to find out the absolute current path). The `CreateDirectory` method returns a DirectoryInfo object, which contains information about the newly created folder. The DirectoryInfo object is discussed later in this chapter, along with the FileInfo object.

Notice that the `CreateDirectory` method can create multiple nested folders in a single call. The following statement will create the folder *`folder1`* (if it doesn't exist), *`folder2`* (if it doesn't exist) under *`folder1`*, and finally *`folder3`* under *`folder2`* in the `C:` drive:

```
Directory.CreateDirectory("C:\folder1\folder2\folder3")
```

If *`folder1`* exists already, but it doesn't contain a subfolder named *`folder2`*, then *`folder2`* will be automatically created. An exception will be thrown if the total path is too long or if your application doesn't have permission to create a folder in the specified path. However, no exception will be thrown if the specified path already exists on the disk. The method will simply not create any new folders. It will still return a DirectoryInfo object, which describes the existing folder.

Delete

This method deletes a folder and all the files in it. If the folder contains subfolders, the `Delete` method will optionally remove the entire directory tree under the node you're removing. The simplest form of the `Delete` method accepts as an argument the path of the folder to be deleted:

```
Directory.Delete(path)
```

This method will delete the specified path only. If the specified folder contains subfolders, they will not be deleted and, therefore, the specified folder won't be deleted, either. To delete a folder recursively (that is, also delete any subfolders under it), use the following form of the `Delete` method, which accepts a second argument:

```
Directory.Delete(path, recursive)
```

The *`recursive`* argument is a True/False value. Set it to True to delete recursively the subfolders under the specified folder. This method deletes folders permanently (it doesn't send them to the Recycle Bin).

The statements in Listing 15.1 attempt to delete a single folder. If the folder contains subfolders, the `Delete` method will fail, and the structured exception handler will be activated. The exception handler examines the type of the exception, and if it was caused because the folder isn't empty, the exception handler prompts the user about whether it should delete the contents of the folder. If the user gives permission to delete the folder's contents, the code calls the second form of the `Delete` method, forcing it to delete the folder recursively.

Listing 15.1: Deleting a Directory

```
Private Sub bttnDelete_Click(...) Handles bttnDelete.Click
    Directory.CreateDirectory( "c:/folder1/folder2/folder3")
        Try
            Directory.Delete("c:\folder1", False)
        Catch exc As IOException
            If exc.Message.IndexOf( _
                "The directory is not empty") > -1 Then
                Dim reply As MsgBoxResult
                reply = MsgBox( _
                  "Delete all files and subfolders?", _
                   MsgBoxStyle.YesNo, "Directory Not Empty")
                If reply = MsgBoxResult.Yes Then
                    Try
                        Directory.Delete("c:\folder1", True)
                    Catch ex As Exception
                        MsgBox("Failed to delete folder" & vbCrLf & _
                                ex.Message)
                    End Try
                Else
                    MsgBox(exc.Message)
                End If
            End If
        End Try
    End Sub
```

Notice the nested `Try...Catch` statement that catches unauthorized exceptions (you may not have the rights to delete the specific folder).

Exists

This method accepts a path as an argument and returns a True/False value indicating whether the specified folder exists:

```
Directory.Exists(path)
```

The `Delete` method will throw an exception if you attempt to delete a folder that doesn't exist, so you can use the `Exists` method to make sure the folder exists before attempting to delete it:

```
If Directory.Exists(path) Then Directory.Delete(path)
```

Move

This method moves an entire folder to another location in the file system; its syntax is the following, where *`source`* is the name of the folder to be moved and *`destination`* is the name of the destination folder:

```
Directory.Move(source, destination)
```

The `Move` method doesn't work along different volumes, and the *`destination`* can't be the same as the *`source`* argument, obviously.

Notice the lack of a `Copy` method that would copy an entire folder to a different location. To copy a folder, you must manually create an identical folder structure and then copy the corresponding files to the proper subfolders. The FileSystem component provides a `MoveFile` and a `MoveFolder` method, which move a single file and an entire folder, respectively.

GetCurrentDirectory, SetCurrentDirectory

Use these methods to retrieve and set the path of the current directory. The current directory is a basic concept when working with files. This is the folder in which all files specified by name will be saved and where the application will look for files specified by their name, not their complete path. Also, relative paths are resolved according to their relation to the current directory. By default, the `GetCurrentDirectory` method returns the folder in which the application is running. `SetCurrentDirectory` accepts a string argument, which is a path, and sets the current directory to the specified path. You can change the current folder by specifying an absolute or a relative path, such as the following:

```
Directory.SetCurrentDirectory("..\Resources")
```

The two periods are a shortcut for the parent folder. From the application folder, we move up to the parent folder and then to the Resources folder under the application's folder. This is where any resources (such as images and sounds) used by the application are stored. Notice that the value you pass to the `SetCurrentDirectory` method as an argument must be the name of an existing folder. If not, a `DirectoryNotFoundException` exception will be thrown. You can also switch to a folder on another drive if you specify the full folder's path, including its drive letter.

If you're working on a new project that hasn't been saved yet, the current directory is the application's folder (WindowsApplication1 or something similar) under the Temporary Projects folders.

GetDirectoryRoot

This method returns the root part of the path passed as argument, and its syntax is the following:

```
root = Directory.GetDirectoryRoot(path)
```

The *`path`* argument is a string, and the return value is also a string, such as `C:\` or `D:\`. Notice that the `GetDirectoryRoot` method doesn't require that the *`path`* argument exists. It will return the name of the root folder of the specified path.

GetDirectories

This method retrieves all the subfolders of a specific folder and returns their names as an array of strings:

```
Dim Dirs() As String
Dirs = Directory.GetDirectories(path)
```

The *`path`* argument is the path of the folder whose subfolders you want to retrieve.

Another form of the `GetDirectories` method allows you to specify search criteria for the folders you want to retrieve, and its syntax is the following:

```
Dirs = Directory.GetDirectories(path, pattern)
```

This statement returns an array of strings with the names of the subfolders that match the search criteria. To retrieve all the subfolders of the `C:\Windows` folder with the string `System` in their names, use the following statement:

```
Dirs = Directory.GetDirectories("C:\Windows", "*SYSTEM*")
```

This statement will go through the subfolders of `C:\WINDOWS` and return those that contain the string **`SYSTEM`** (including `System32` and `MySystem`). The only special characters you can use in the criteria specification are the question mark, which stands for any single character, and the asterisk, which stands for any string. Listing 15.2 retrieves the names of the folders that contain the string `System` under the `C:\WINDOWS` folder and prints them in the Output window.

Listing 15.2: Retrieving Selected Subfolders of a Folder

```
Dim Dirs() As String
Dirs = Directory.GetDirectories("C:\WINDOWS", "*SYSTEM*")
Dim dir As String
Debug.WriteLine(Dirs.Length & " folders match the pattern '*SYSTEM*' ")
```

```
For Each dir In Dirs
   Debug.WriteLine(dir)
Next
```

The `GetDirectories` method doesn't work recursively; it returns the subfolders of the specified folder, but not their subfolders.

GetFiles

This method returns the names of the files in the specified folder as an array of strings. The syntax of the `GetFiles` method is the following, where *`path`* is the path of the folder whose files you want to retrieve and *`files`* is an array of strings that's filled with the names of the files:

```
Dim files() As String = Directory.GetFiles(path)
```

Another form of the `GetFiles` method allows you to specify a pattern and retrieve only the names of the files that match the pattern. This form of the method accepts a second argument, which is a string similar to the *`pattern`* argument of the `GetDirectories` method:

```
Dim files() As String = Directory.GetFiles(path, pattern)
```

The statements in Listing 15.3 retrieve all the `.exe` files under the `C:\WINDOWS` folder and print their names in the Output window.

LISTING 15.3: Retrieving Selected Files of a Folder

```
Dim files() As String
files = Directory.GetFiles("C:\WINDOWS", "*.EXE")
MsgBox("Found " & files.Length & " EXE files")
Dim file As String
For Each file In files
   Debug.WriteLine(file)
Next
```

GetFileSystemEntries

This method returns an array of all items (files and folders) in a path. The simplest form of the method is

```
items = Directory.GetFileSystemEntries(path)
```

where *`items`* is an array of strings. As with the `GetFiles` method, you can specify a second argument, which filters the entries you want to retrieve. To iterate through the items of a folder, use a loop such as the following:

```
Dim itm As String
For Each itm In Directory.GetFileSystemEntries("C:\windows")
    Debug.WriteLine(itm)
Next
```

Because the `GetFileSystemEntries` method returns an array of strings, use the `Exists` method of the Directory object to distinguish between folders and files. The File object, which is equivalent to the Directory object and is discussed in the following section, also exposes an `Exists` method. The loop shown in Listing 15.4 goes through the file system items in the `C:\Program Files` folder and displays their names, along with the indication `FOLDER` or `FILE`, depending on the type of each item.

Listing 15.4: Retrieving the File System Items of a Folder

```
Dim items() As String
Dim path As String = "c:\Program Files"
items = Directory.GetFileSystemEntries(path)
Dim itm As String
For Each itm In items
   If Directory.Exists(itm) Then
      Debug.WriteLine("FOLDER " & itm)
    Else
      Debug.WriteLine("FILE " & itm)
    End If
Next
```

If you execute these statements, you will see a list such as the following in the Output window (only considerably longer):

```
FOLDER c:\Program Files\Microsoft.NET
FOLDER c:\Program Files\HTML Help Workshop
FOLDER c:\Program Files\Microsoft Web Controls 0.6
FILE c:\Program Files\folder.htt
FILE c:\Program Files\desktop.ini
```

The My.Computer.FileSystem component doesn't expose a method to retrieve folders and files at once. Instead, you must use the `GetFiles` and `GetDirectories` methods to retrieve either the files or the folders under a specific folder.

GetCreationTime, SetCreationTime

These methods read or set the date that a specific folder was created. The `GetCreationTime` method accepts a path as an argument and returns a Date value:

```
Dim CreatedOn As Date
CreatedOn = Directory.GetCreationTime(path)
```

`SetCreationTime` accepts a path and a date value as arguments and sets the specified folder's creation time to the value specified by the second argument:

```
Directory.SetCreationTime(path, datetime)
```

GetLastAccessTime, SetLastAccessTime

These two methods are equivalent to the `GetCreationTime` and `SetCreationTime` methods, except they return and set the most recent date and time that the file was accessed. The most common reason to change the last access time for a file is so that the specific file will be excluded from a routine that deletes old files or to include it in a list of backup files (with an automated procedure that backs up only the files that have been changed since their last backup).

GetLastWriteTime, SetLastWriteTime

These two methods are equivalent to the `GetCreationTime` and `SetCreationTime` methods, but they return and set the most recent date and time the file was written to.

GetLogicalDrives

This method returns an array of strings, which are the names of the logical drives on the computer. The statements in Listing 15.5 print the names of all logical drives.

LISTING 15.5: Retrieving the Names of All Drives on the Computer

```
Dim drives() As String
drives = Directory.GetLogicalDrives
Dim drive As String
For Each drive In drives
    Debug.WriteLine(drive)
Next
```

When executed, these statements will produce a list such as the following:

```
C:\
D:\
E:\
F:\
```

Notice that the `GetLogicalDrives` method doesn't return any floppy drives, unless there's a disk inserted into the drive.

GetParent

This method returns a DirectoryInfo object that represents the properties of a folder's parent folder. The syntax of the `GetParent` method is as follows:

```
Dim parent As DirectoryInfo = Directory.GetParent(path)
```

The name of the parent folder, for example, is `parent.Name`, and its full name is `parent.FullName`.

The File Class

The System.IO.File class exposes methods for manipulating files (copying them, moving them around, opening them, and closing them), similar to the methods of the Directory class. The names of the methods are self-descriptive, and most of them accept as an argument the path of the file on which they act. Use these methods to implement the common operations that users normally perform through the Windows interface, from within your application.

Methods

Many of the following methods allow you to open existing or create new files. We'll use some of these methods later in the chapter to write data to, and read from, text and binary files.

AppendText

This method appends some text to a file, whose path is passed to the method as an argument, along with the text to be written:

```
File.AppendText(path, text)
```

Copy

This method copies an existing file to a new location; its syntax is the following, where *source* is the path of the file to be copied and *destination* is the path where the file will be copied to:

```
File.Copy(source, destination)
```

If the destination file exists, the Copy method will fail. An exception will be thrown also if either the source or the destination folder does not exist.

To overwrite the destination file, use the following form of the method, which allows you to specify whether the destination file can be overwritten with a True/False value (the *overwrite* argument):

```
File.Copy(source, destination, overwrite)
```

The Copy method works across volumes. The following statement copies the file faces.jpg from the folder C:\My Documents\Screen\ to the folder D:\Fun Images and changes its name to Bouncing Face.jpg:

```
File.Copy("C:\My Documents\Screen\faces.jpg", _
          "D:\Fun Images\Bouncing Face.jpg")
```

The Copy method doesn't accept wildcard characters; you can't copy multiple files via a single call to the Copy method.

Create

This method creates a new file and returns a FileStream object, which you can use to write to or read from the file. (The FileStream object is discussed in detail later in this chapter, along with the methods for writing to or reading from the file.) The simplest form of the Create method accepts a single argument, which is the path of the file you want to create:

```
Dim FStream As FileStream = File.Create(path)
```

You can also create a new file and specify the size of the buffer to be associated with this file by using the following form of the method, where *bufferSize* is an Integer (Int32) value:

```
FStream = File.Create(path, bufferSize)
```

If the specified file exists already, it's replaced. The new file is opened for read-write operations, and it's opened exclusively by your application. Other applications can access it only after your application closes it. After the file has been created, you can use the methods of the FileStream object to write to it. These methods are discussed in the section ''Accessing Files,'' later in this chapter.

The `Create` method can raise several exceptions, which are described in Table 15.1. Pathnames are limited to 248 characters, and filenames are limited to 259 characters.

TABLE 15.1: Exceptions of the *Create* Method

EXCEPTION	DESCRIPTION
`IOException`	The folder you specified doesn't exist.
`ArgumentNullException`	The path you specified doesn't reference a file.
`SecurityException`	The user of your application doesn't have permission to create a new file in the specified folder.
`ArgumentException`	The path you specified is invalid.
`AccessException`	The file can't be opened in read-write mode. Most likely, you've attempted to open a read-only file, but the `File.Create` method opens a file in read-write mode.
`DirectoryNotFoundException`	The folder you specified doesn't exist.

CreateText

This method is similar to the `Create` method, but it creates a text file and returns a StreamWriter object for writing to the file. The StreamWriter object is similar to the FileStream object but is used for text files only, whereas the FileStream object can be used with both text and binary files.

```
Dim SW As StreamWriter = File.CreateText(path)
```

Delete

This method removes the specified file from the file system. The syntax of the `Delete` method is the following, where *path* is the path of the file you want to delete:

```
File.Delete(path)
```

This method will raise an exception if the file is open at the time for reading or writing, or if the file doesn't exist.

Notice that the `Delete` method of the File object deletes files permanently and doesn't send them to the Recycle Bin. Moreover, it doesn't recognize wildcard characters. To delete all the files in a folder, you must call the Directory object's `Delete` method to remove the entire folder.

Exists

This method accepts as an argument the path of a file and returns a True/False value that indicates whether a file exists. The following statements delete a file, after making sure that the file exists:

```
If File.Exists(path) Then
   File.Delete(path)
Else
   MsgBox("The file " & path & " doesn't exist")
End If
```

The `File.Delete` method will not raise an exception if the file doesn't exist, so you don't have to make sure that a file exists before deleting it.

GetAttributes

The `GetAttributes` method accepts a file path as an argument and returns the attributes of the specified file as a FileAttributes object. A file can have more than a single attribute (for instance, it can be hidden and compressed). Table 15.2 lists all possible attributes a file can have.

TABLE 15.2: Attributes of a File

VALUE	DESCRIPTION
`Archive`	The file's archive status. Most of the files in your file system have the Archive attribute.
`Compressed`	The file is compressed.
`Encrypted`	The file is encrypted.
`Hidden`	The file is hidden, and it doesn't appear in an ordinary directory listing.
`Normal`	Normal files have no other attributes, so this setting excludes all other attributes.
`NotContentIndexed`	The file isn't indexed by the operating system's content-indexing service.
`Offline`	The file is offline, and its contents might not be available at all times.
`ReadOnly`	The file is read-only.
`SparseFile`	The file is sparse (a large file whose data are mostly zeros).
`System`	A file that is part of the operating system or is used exclusively by the operating system.
`Temporary`	The file is temporary. Temporary files are created by applications and they're deleted by the same applications that created them when they terminate.

To examine whether a file has an attribute set, you must check the value returned by the `GetAttributes` method with the desired attribute, which is a member of the `FileAttributes` enumeration. To find out whether a file is read-only, use the following `If` statement:

```
If File.GetAttributes(fpath) And FileAttributes.ReadOnly Then
   Debug.WriteLine("The file " & fpath & " is read only")
Else
   Debug.WriteLine("You can write to the file " & fpath)
End If
```

You can also retrieve a file's attributes through the FileInfo object, described later in this chapter.

GetCreationTime, SetCreationTime

The `GetCreationTime` method returns a date value, which is the date and time the file was created. This value is set by the operating system, but you can change it with the `SetCreationTime` method. `SetCreationTime` accepts as an argument the file's path and the new creation time:

```
File.SetCreationTime(path, datetime)
```

GetLastAccessTime, SetLastAccessTime

The `GetLastAccessTime` method returns a date value, which is the date and time the specified file was accessed for the last time. Use the `SetLastAccessTime` method to set this value. (Its syntax is identical to the syntax of the `SetCreationTime` method.) Changing the last access of a file is sometimes called *touching* the file. If you have a utility that manipulates files according to when they were last used (for example, one that moves data files that haven't been accessed in the last three months to tape), you can *touch* a few files to exclude them from the operation.

GetLastWriteTime, SetLastWriteTime

The `GetLastWriteTime` method returns a date value, which is the date and time that the specified file was written to for the last time. To change this attribute, use the `SetLastWriteTime` method.

Move

This method moves the specified file to a new location. You can also use the `Move` method to rename a file by simply moving it to another name in the same folder. Moving a file is equivalent to copying it to another location and then deleting the original file. The `Move` method works across volumes:

```
File.Move(sourceFileName, destFileName)
```

The first argument is the path of the file to be moved, and the second argument is the path of the destination file. The `Move` method will throw an exception if the source file or the destination does not exist, if the application doesn't have write permission on the destination folder, or if one of the arguments is invalid.

Open

This method opens an existing file for read-write operations. The simplest form of the method is the following, which opens the file specified by the *path* argument and returns a FileStream object to this file:

```
FStream = File.Open(path)
```

You can use the *FStream* object's methods to write to or read from the file. The following form of the method allows you to specify the mode in which you want to open the file, where the *fileMode* argument can have one of the values shown in Table 15.3.

```
FStream = File.Open(path, fileMode)
```

Table 15.3: *FileMode* Enumeration

Value	Effect
`Append`	Opens the file in write mode, and all the data you write to the file are appended to its existing contents.
`Create`	Requests the creation of a new file. If a file by the same name exists, this will be overwritten.
`CreateNew`	Requests the creation of a new file. If a file by the same name exists, an exception will be thrown. This mode will create and open a file only if it doesn't already exist and it's the safest mode.
`Open`	Requests that an existing file be opened.
`OpenOrCreate`	Opens the file in read-write mode if the file exists, or creates a new file and opens it in read-write mode if the file doesn't exist.
`Truncate`	Opens an existing file and resets its size to zero bytes. As you can guess, this file must be opened in write mode.

Another form of the `Open` method allows you to specify the access mode in addition to the file mode, where the *accessMode* argument can have one of the values listed in Table 15.4:

```
FStream = File.Open(path, fileMode, accessMode)
```

You can also specify a fourth argument to the `Open` method, which specifies how the file will be shared with other applications. This form of the method requires that the other two arguments (*fileMode* and *accessMode*) be supplied as well:

```
FStream = File.Open(path, fileMode, accessMode, shareMode)
```

The *shareMode* argument determines how the file will be shared among multiple applications and can have one of the values in Table 15.5.

TABLE 15.4: *AccessMode* Enumeration

VALUE	EFFECT
`Read`	The file is opened in read-only mode. You can read from the Stream object that is returned, but an exception will be thrown if you attempt to write to the file.
`ReadWrite`	The file is opened in read-write mode. You can either write to the file or read from it.
`Write`	The file is opened in write mode. You can write to the file, but if you attempt to read from it, an exception will be thrown.

TABLE 15.5: *ShareMode* Enumeration

VALUE	EFFECT
`None`	The file can't be shared for reading or writing. If another application attempts to open the file, it will fail until the current application closes the file.
`Read`	The file can be opened by other applications for reading, but not for writing.
`ReadWrite`	The file can be opened by other applications for reading or writing.
`Write`	The file can be opened by other applications for writing, but not for reading.

OpenRead

This method opens an existing file in read mode and returns a FileStream object associated with this file. You can use this stream to read from the file. The syntax of the `OpenRead` method is the following:

```
Dim FStream As FileStream = File.OpenRead(path)
```

The `OpenRead` method is equivalent to opening an existing file with read-only access via the `Open` method.

OpenText

This method opens an existing text file for reading and returns a StreamReader object associated with this file. Its syntax is the following:

```
Dim SR As StreamReader = File.OpenText(path)
```

Why do we need an `OpenText` method in addition to the `Open`, `OpenRead`, and `OpenWrite` methods? The answer is that text can be stored in different formats. It can be plain text (UTF-8 encoding), ASCII text, or Unicode text. The StreamReader object associated with the text file will perform the necessary conversions, and you will always read the correct text from the file. The default encoding for the `OpenText` method is UTF-8.

OpenWrite

This method opens an existing file in write mode and returns a FileStrem object associated with this file. You can use this stream to write to the file, as you will see later in this chapter.

The syntax of the `OpenRead` method is as follows, where *`path`* is the path of the file:

```
Dim FStream As FileStream = File.OpenWrite(path)
```

To write data to the file, use the methods of the FileStream object, which are discussed later in this chapter. This ends our discussion of the Directory and File classes, which are the two major objects for manipulating files and folders. In the following section, I will present the DriveInfo, DirectoryInfo, and FileInfo classes briefly, and then we'll build an application that puts together much of the information presented so far.

Drive, Folder, and File Properties

The IO namespace provides three objects that represent drives, folders, and files: the DriveInfo, DirectoryInfo, and FileInfo classes. These classes, in turn, expose a number of basic properties of the entities they represent. Notice that they're instance objects, and you must create a new instance of the corresponding class by specifying the name of a drive/folder/file in its constructor.

The same three objects are returned by the `GetDriveInfo`, `GetDirectoryInfo` and `GetFile-Info` methods of the FileSystem object.

The DriveInfo Class

The DriveInfo class provides basic information about a drive. Its constructor accepts as an argument a drive name, and you can use the object returned by the method to retrieve information about the specific drive, as shown here:

```
Dim Drive As New DriveInfo("C")
```

The argument is the name of a drive (you can include the colon, if you want). Notice that you can't specify a Universal Naming Convention (UNC) path with the constructor of the DriveInfo object. You can only access local drives or network drives that have been mapped to a drive name on the target system.

To retrieve information about the specified drive, use the following properties of the DriveInfo class:

DriveFormat A string describing the drive's format (FAT32, NTFS).

DriveType A string describing the drive's type (fixed, CD-ROM, and so on).

TotalSize The drive's total capacity, in bytes.

TotalFreeSize The total free space on the drive, in bytes.

AvailableFreeSpace The available free space on the drive, in bytes.

VolumeLabel The drive's label. You can change the drive's label by setting this property.

IsReady A True/False value indicating whether the drive is ready to be used. Retrieve this property's setting before calling any of the other properties to make sure that you're not attempting to access an empty floppy or CD drive.

Discovering the System's Drives

The DriveInfo class exposes the `GetDrives` method, which returns an array of DriveInfo objects, one for each drive on the system. This method is similar to the `GetLogicalDrives` method of the Directory object, which is a shared method and doesn't require that you create an object explicitly.

The DirectoryInfo Class

To create a new instance of the DirectoryInfo class that references a specific folder, supply the folder's path in the class's constructor:

```
Dim DI As New DirectoryInfo(path)
```

The members of the DirectoryInfo class are equivalent to the members of the Directory class, and you will recognize them as soon as you see them in the IntelliSense drop-down list. Here are a couple of methods that are unique to the DirectoryInfo class.

CreateSubdirectory

This method creates a subfolder under the folder specified by the current instance of the class, and its syntax is as follows:

```
DI.CreateSubdirectory(path)
```

The `CreateSubdirectory` method returns a DirectoryInfo object that represents the new subfolder. The *`path`* argument need not be a single folder's name. If you specified multiple nested folders, the `CreateSubdirectory` method will create the appropriate hierarchy, similar to the `CreateDirectory` method of the Directory class.

GetFileSystemInfos

This method returns an array of FileSystemInfo objects, one for each item in the folder referenced by the current instance of the class. The items can be either folders or files. To retrieve information about all the entries in a folder, create an instance of the DirectoryInfo class and then call its `GetFileSystemInfos` method:

```
Dim DI As New DirectoryInfo(path)
Dim itemsInfo() As FileSystemInfo
itemsInfo = DI.GetFileSystemInfos()
```

You can also specify an optional search pattern as an argument when you call this method:

```
itemsInfo = DI.GetFileSystemInfos(pattern)
```

The FileSystemInfo objects expose a few properties, which are not new to you. The `Name`, `Full-Name`, and `Extension` properties return a file's or folder's name, or full path, or a file's extension, respectively. `CreationTime`, `LastAccessTime`, and `LastWriteTime` are also properties of the FileSystemInfo object, as well as the `Attributes` property.

You will notice that there are no properties that determine whether the current item is a folder or a file. To find out the type of an item, use the `Directory` member of the `Attributes` property:

```
If itemsInfo(i).Attributes And FileAttributes.Directory Then
   { current item is a folder }
Else
   { current item is a file }
End If
```

The code in Listing 15.6 retrieves all the items in the `C:\Program Files` folder and prints their names along with the `FOLDER` or `FILE` characterization.

Listing 15.6: Processing a Folder's Items with the FileSystemInfo Object

```
Dim path As String = "C:\Program Files"
Dim DI As New DirectoryInfo(path)
Dim itemsInfo() As FileSystemInfo
itemsInfo = DI.GetFileSystemInfos()
Dim item As FileSystemInfo
For Each item In itemsInfo
   If (item.Attributes And FileAttributes.Directory)= _
            FileAttributes.Directory Then
      Debug.Write("FOLDER ")
   Else
      Debug.Write("FILE  ")
   End If
   Debug.WriteLine(item.Name)
Next
```

Notice the differences between the `GetFileSystemInfos` method of the DirectoryInfo class and the `GetFileSystemEntries` of the Directory object. `GetFileSystemInfos` returns an array of objects that contains information about the current item (file or folder). `GetFileSystemEntries` returns an array of strings (the names of the folders and files).

The FileInfo Class

The FileInfo class exposes many properties and methods, which are equivalent to the members of the File class, so I'm not going to repeat all of them here. The `Copy`/`Delete`/`Move` methods allow you to manipulate the file represented by the current instance of the FileInfo class, similar to the methods by the same name of the File class. Although there's substantial overlap between the members of the FileInfo and File classes, the difference is that with FileInfo you don't have to specify a path; its members act on the file represented by the current instance of the FileInfo class, and this file is passed as an argument to the constructor of the FileInfo class. The FileInfo class exposes a few rather trivial properties, which are mentioned briefly here.

Length Property

This property returns the size of the file represented by the FileInfo object in bytes. The File class doesn't provide an equivalent property or method.

CreationTime, LastAccessTime, LastWriteTime Properties

These properties return a date value, which is the date the file was created, accessed for the last time, or written to for the last time, respectively. They are equivalent to the methods of the File object by the same name and the `Get` prefix.

Name, FullName, Extension Properties

These properties return the filename, full path, and extension, respectively, of the file represented by the current instance of the FileInfo class. They have no equivalents in the File class because the File class's methods require that you specify the path of the file, so its path and extension are known.

CopyTo, MoveTo Methods

These two methods copy or move, respectively, the file represented by the current instance of the FileInfo class. Both methods accept a single argument, which is the destination of the operation (the path to which the file will be copied or moved). If the destination file exists already, you can overwrite it by specifying a second optional argument, which has a True/False value:

```
FileInfo.CopyTo(path, force)
```

Both methods return an instance of the FileInfo class, which represents the new file — if the operation completed successfully.

Directory Method

This method returns a DirectoryInfo value that contains information about the file's parent directory.

DirectoryName Method

This method returns a string with the name of the file's parent directory. The following statements return the two (identical) strings shown highlighted in this code segment:

```
Dim FI As FileInfo
FI = New FileInfo("c:\folder1\folder2\folder3\test.txt")
Debug.WriteLine(FI.Directory().FullName)
c:\folder1\folder2\folder3
Debug.WriteLine(FI.DirectoryName()) c:\folder1\folder2\folder3
```

Of course, the `Directory` method returns an object, which you can use to retrieve other properties of the parent folder.

The Path Class

The Path class contains an interesting collection of methods, which you can think of as utilities. The Path class's methods perform simple tasks such as retrieving a file's name and extension, returning the full path description of a relative path, and so on. The Path class's members are shared, and you must specify the path on which they will act as an argument.

Properties

The Path class exposes the following properties. Notice that none of these properties applies to a specific path; they're general properties that return settings of the operating system. The FileSystem component doesn't provide equivalent properties to the ones discussed in this section.

DirectorySeparatorChar

This property returns the directory separator character, which is the backslash character (\).

InvalidPathChars

This property returns the list of invalid characters in a path as an array of the following characters:

```
/  \  "  <  >  -
```

You can use these characters to validate user input or pathnames read from a file. If you have a choice, let the user select the files through the Open dialog box, so that their pathnames will always be valid.

PathSeparator, VolumeSeparatorChar

These properties return the separator characters that appear between multiple paths (:) and volumes (;), respectively.

Methods

The most useful methods exposed by the Path class are utilities for manipulating filenames and pathnames, described in the following sections. Notice that the methods of the Path class are shared: You must specify the path on which they will act as an argument.

ChangeExtension

This method changes the extension of a file. Its syntax is as follows:

```
newExtension = Path.ChangeExtension(path, extension)
```

The return value is the new extension of the file (a string value), and you can examine it from within your code to make sure that the operation completed successfully. The first argument is the file's path, and the second argument is the file's new extension. If you want to remove the file's extension, set the second argument to `Nothing`. The following statement changes the extension of the specified file from `.bin` to `.dat`:

```
Dim path As String = "c:\My Documents\NewSales.bin"
Dim newExt As String = ".dat"
Path.ChangeExtension(path, newExt)
```

Combine

This method combines two path specifications into one. Its syntax is as follows:

```
newPath = Path.Combine(path1, path2)
```

Use this method to combine a folder path with a file path. The following expression will return the highlighted string:

```
Path.Combine("c:\textFiles", "test.txt")
c:\textFiles\test.txt
```

Notice that the `Combine` method inserted the separator, as needed. It's a simple operation, but if you had to code it yourself, you'd have to examine each path and determine whether a separator must be inserted.

GetDirectoryName

This method returns the directory name of a path. The following statement:

```
Path.GetDirectoryName("C:\folder1\folder2\folder3\Test.txt")
```

will return this string:

```
C:\folder1\folder2\folder3
```

GetFileName, GetFileNameWithoutExtension

These two methods return the filename in a path, with and without its extension, respectively.

GetFullPath

This method returns the full path of the specified path; you can use it to convert relative pathnames to fully qualified pathnames. The following statement returned the highlighted string on my computer (it will be quite different on your computer, depending on the current directory):

```
Console.WriteLine(Path.GetFullPath("..\..\Test.txt"))
C:\WorkFiles\Mastering VB\Chapters\Chapter 15\Projects\Test.txt
```

The pathname passed to the method as an argument need not exist. The `GetFullPath` method will return the fully qualified pathname of a nonexistent file, as long as the path doesn't contain invalid characters.

GetTempFile, GetTempPath

The `GetTempFile` method returns a unique filename, which you can use as a temporary storage area from within your application. The name of the temporary file can be anything, because no user will ever access it. In addition, the `GetTempFile` method creates a zero-length file on the disk, which you can open with the `Open` method. A typical temporary filename is the following:

```
C:\DOCUME~1\TOOLKI~1\LOCALS~1\Temp\tmp105.tmp
```

It was returned by the following statement on my system:

```
Debug.WriteLine(Path.GetTempFile)
```

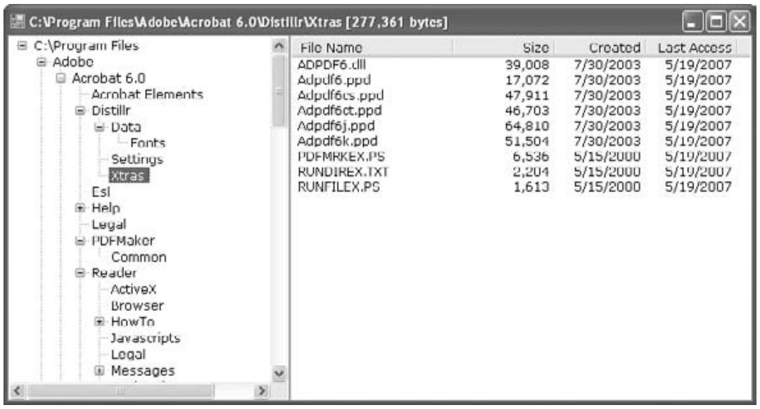
The `GetTempPath` method returns the system's temporary folder. All temporary files should be created in this folder, so that the operating system can remove them when it's running out of space. Your applications should remove all the temporary files they create, but more often than not, programmers leave temporary files around.

HasExtension

This method returns a True/False value, indicating whether a path includes a file extension.

VB 2008 at Work: The CustomExplorer Project

The CustomExplorer application, which demonstrates the basic properties and methods of the Directory and File classes, duplicates the functionality of Windows Explorer. Its user interface, shown in Figure 15.2, was discussed in Chapter 9, ''The TreeView and ListView Controls.'' In this chapter, you'll see how to access the file system and populate the two controls with folder names and filenames, using the basic members of the Directory and File classes of the System.IO namespace. You can implement the same application with the FileSystem component as an exercise. I will post the same application implemented with the FileSystem component of the My object in this chapter's Projects folder for your convenience.



File Name	Size	Created	Last Access
ADPDF6.dll	39,008	7/30/2003	5/19/2007
Adpdf6.ppd	17,072	7/30/2003	5/19/2007
Adpdf6cs.ppd	47,911	7/30/2003	5/19/2007
Adpdf6ct.ppd	46,703	7/30/2003	5/19/2007
Adpdf6j.ppd	64,810	7/30/2003	5/19/2007
Adpdf6k.ppd	51,504	7/30/2003	5/19/2007
PDFMRKEX.PS	6,536	5/15/2000	5/19/2007
RUNDIREX.TXT	2,204	5/15/2000	5/19/2007
RUNFILEX.PS	1,613	5/15/2000	5/19/2007

FIGURE 15.2 The CustomExplorer project

When you start the application, the names of the subfolders under the `C:\Program Files` folder are displayed in the TreeView control, in a hierarchical structure. This operation takes a few seconds because the code must scan the entire folder, including its subfolders, and generate the necessary nodes on the TreeView control. The more programs you have installed on your `C:` drive, the longer it will take to scan them. Change the value of the *`initFolder`* variable in Listing 15.7 to scan a different folder. As the code iterates through the subfolders, it displays the name of the current folder on the form's title bar, so that you can monitor the progress of the operation. This is one form of feedback you can provide for this operation.

LISTING 15.7: CustomExplorer's *Form_Load* Event Handler

```
Private Sub Form1_Load(...) Handles Me.Load
    Dim Nd As New TreeNode
    Dim initFolder As String = "C:\Program Files"
    Nd = TreeView1.Nodes.Add(initFolder)
    Me.Show()
    Application.DoEvents()
    Me.Cursor = Cursors.WaitCursor
    ScanFolder(initFolder, Nd)
    Me.Cursor = Cursors.Default
End Sub
```

As you can guess, all the work is done by the `ScanFolder()` subroutine, which accepts as arguments the path of the folder to scan and the current node on the TreeView control. The subfolders of the specified folder will be added to the TreeView control as child nodes of the current node, and this is why the `ScanFolder()` subroutine needs a reference to the current node.

The `ScanFolder()` subroutine iterates through the subfolders of the specified folder recursively and creates a new node for each subfolder. If a subfolder contains subfolders of its own, the `ScanFolder()` subroutine calls itself, passing the name of the subfolder as an argument. This way, each folder is scanned completely, regardless of it depth (the levels of nested subfolders). Listing 15.8 shows the code of the `ScanFolders()` subroutine.

LISTING 15.8: Displaying the Subfolders of the Selected Folder

```
Sub ScanFolder(ByVal folderSpec As String, _
               ByRef currentNode As TreeNode)
    Dim thisFolder As String
    Dim allFolders() As String
    allFolders = IO.Directory.GetDirectories(folderSpec)
    For Each thisFolder In allFolders
        Dim Nd As TreeNode
        Nd = New TreeNode(Path.GetFileName(thisFolder))
        currentNode.Nodes.Add(Nd)
        folderSpec = thisFolder
        ScanFolder(folderSpec, Nd)
        Me.Text = "Scanning " & folderSpec
        Me.Refresh()
    Next
End Sub
```

The `ScanFolder()` subroutine is surprisingly simple because it's recursive: It calls itself again and again to iterate through all the subfolders of the specified folder. The `GetDirectories` method retrieves the names of the subfolders of the current folder and returns them as an array of strings: the `allFolders` array. The following loop iterates through the elements of this array. For each element, it adds a new node under the current node on the TreeView control and then calls itself, passing the current folder's name as an argument. If the current folder contains subfolders, they will be added to the TreeView control as well.

In the Form's `Load` event handler, we populate the TreeView control with the hierarchy of the initial folder's subfolders. You should probably provide a Browse For Folder dialog box to allow users to select the folder (or drive) to be mapped on the TreeView control. To view the files in a specific folder, you can click the folder's name in the TreeView control. The ListView control on the left will be populated with the names and basic properties of the files in the selected folder. The code that displays the list of files in the selected folder resides in the `AfterSelect` event handler of the TreeView control (it's shown in Listing 15.9).

Listing 15.9: Displaying a Folder's Files

```
Private Sub TreeView1_AfterSelect( _
       ByVal sender As System.Object, _
       ByVal e As System.Windows.Forms.TreeViewEventArgs) _
       Handles TreeView1.AfterSelect
    Dim Nd As TreeNode
    Dim pathName As String
    Nd = TreeView1.SelectedNode
    pathName = Nd.FullPath
    Me.Text = pathName
    ShowFiles(pathName)
End Sub
```

The `ShowFiles()` subroutine accepts as an argument a folder path and displays the files in the folder, along with their basic properties. Its code, which is shown in Listing 15.10, iterates through the array with the filenames returned by the `Directory.GetFiles` method and uses the FileInfo class to retrieve each file's basic properties.

Listing 15.10: *ShowFiles()* Subroutine

```
Sub ShowFiles(ByVal selFolder As String)
    ListView1.Items.Clear()
    Dim files() As String
    Dim file As String
    files = IO.Directory.GetFiles(selFolder)
    Dim TotalSize As Long
    Dim FI As IO.FileInfo
    For Each file In files
        Dim LItem As New ListViewItem
        LItem.Text = IO.Path.GetFileName(file)
        FI = New IO.FileInfo(file)
        LItem.SubItems.Add(FI.Length.ToString(''#,###''))
        LItem.SubItems.Add( _
                  FI.CreationTime.ToShortDateString)
        LItem.SubItems.Add( _
                  FI.LastAccessTime.ToShortDateString)
        ListView1.Items.Add(LItem)
        TotalSize += FI.Length
    Next
    Me.Text = Me.Text & " [" & TotalSize.ToString("#,###") & " bytes]"
End Sub
```

Accessing Files

In the first half of the chapter, you learned how to manipulate files and folders. Now we'll discuss how to access files (write data to files and read it back). You've already seen the various `Open` methods of the File class, which return a Stream object. It's this object that provides the methods for writing to and reading from files.

There are two types of files: text files and binary files. Of course, you can classify files in any way you like, but when it comes to writing to and reading from files, it's convenient to treat them as either text or binary. A *binary file* is any file that doesn't contain plain text. *Text files* are usually read line by line or in their entirety into a String variable. Binary files must be read according to the type of information stored in them. A bitmap file, for instance, must be read 1 byte at a time. Each pixel is usually represented by 3 or 4 bytes, and you must combine the values read to reconstruct the pixel's color. You can also read Long values from an image file. Or you can read a Color variable directly from the stream. Most binary files contain multiple data types, and you must know the organization of a file before you can read it.

To access a file, you must first set up a Stream object. *Stream objects* are created by the various methods that open or create files, as you have seen in the previous sections, and they return information about the file they're connected to.

After the Stream object is in place, you create a Reader or Writer object, which enables you to read information from or write information into the Stream, respectively. The Reader and Writer classes are abstracts that you can't use directly in your code. There are two classes that inherit from the Reader class: the StreamReader class for text files and the BinaryReader class for binary files. Likewise, there are two classes that inherit the Writer class: the StreamWriter and the BinaryWriter classes. These classes expose a few properties and methods for writing to files and reading from them, and their members are discussed shortly.

Using Streams, Readers, and Writers

The FileStream class is derived from the Stream abstract class and represents a stream of bytes. Use its methods to find the length of the stream, to lock the stream, and to navigate to a specific location in the stream. FileStream also provides methods of writing to the stream and reading from it. The `Write` and `WriteByte` methods write an array of bytes and a single byte to the stream, respectively. The `Read` and `ReadByte` methods read the same data back from the stream. These methods are not used frequently, because developers usually manipulate more-specific data types (such as strings and decimals) or custom data types, not bytes. If you're dealing with bytes, or you don't mind converting your data to and from Byte arrays, use the methods of the FileStream class to write data to a file. For most applications, however, you'll use the methods of the Stream class.

Typical Windows applications set up a FileStream object to open a channel between the application and a file, and then a StreamWriter/StreamReader object on top of it. These two classes provide more-flexible methods for sending data to the underlying file and reading them back. For binary files, use the BinaryWriter/BinaryReader classes. The Reader/Writer classes know how to send data to a Stream object and read them back, while the Stream object knows how to interact with the underlying file. The IO namespace provides many ways to exchange data with a file and you'll rarely use the FileStream class's methods to write or read data directly to or from a file.

Using Streams

You can think of the Stream as a channel between your application and the source or destination of the data. In most cases, the source (or destination) is a file. The Stream abstracts a very basic operation: the operation of sending or receiving data. Does it really make any difference whether you write to a file or send data to a web client? Technically, it's a world of difference, but wouldn't it be nice if we could specify the destination and then send the data (or request data from a source)? The Framework abstracts this basic operation by establishing a Stream between the application and the source, or destination, of the data. Consider an application that successfully writes data to a file. If you change the definition of the stream, you can send the same data to a different destination via the same statements. You can send the data to another machine or a web client.

Another benefit of using streams is that you can combine them. The typical example is that of encrypting and decrypting data. Data is encrypted through a special type of Stream, the CryptoStream. You write plain data to the CryptoStream, and they're encrypted by the stream itself. In other words, the CryptoStream object accepts plain data and emits encrypted data. You can connect the CryptoStream object to another Stream object that represents a file and write the encrypted data directly to the file. Your code uses simple statements to write data to the CryptoStream object, which encrypts the data and then passes it to another stream that writes the encrypted data to a file.

The FileStream Class

The Stream class is an abstract one, and you can't use it directly in your code. To prepare your application to write to a file, you must set up a FileStream object, which is the channel between your application and the file. The methods for writing and reading data are provided by the StreamReader/StreamWriter or BinaryReader/BinaryWriter classes, which are created on top of the FileStream object.

The FileStream object's constructor is overloaded; its most common forms require that you specify the path of the file and the mode in which the file will be opened (for reading, appending, writing, and so on). The simpler form of the constructor is as follows:

```
Dim FS As New FileStream(path, fileMode)
```

The *`fileMode`* argument is a member of the `FileMode` enumeration (see Table 15.3). It's the same argument used by the `Open` method of the File class. Also similar to the `Open` method of the File class, another overloaded form of the constructor allows you to specify the file's access mode; the syntax of this method is the following:

```
Dim FS As New FileStream(path, fileMode, fileAccess)
```

The last argument is a member of the `FileAccess` enumeration (see Table 15.4). The last overloaded form of the constructor accepts a fourth argument, which determines the file's sharing mode:

```
Dim FS As New FileStream(path, fileMode, fileAccess, fileShare)
```

The *`fileShare`* argument's value is a member of the `FileShare` enumeration (see Table 15.5).

Properties

You can use the following properties of the FileStream object to retrieve information about the underlying file.

CanRead, CanSeek, CanWrite

These three properties are read-only and they determine whether the current stream supports reading, seeking, and writing, respectively. If the file associated with a specific FileStream object can be read, the `CanRead` property returns True. A seek operation in the context of files doesn't locate a specific value in the file. It simply moves the current position to any location within the file. The `CanWrite` property is a True/False value that's True if the file associated with a specific FileStream object can be written to and False if the file can't be written.

Length

This read-only property returns the length of the file associated with the FileStream current object in bytes.

Position

This property gets or sets the current position within the stream. You can compare the `Position` property to the `Length` property to find out whether you have reached the end of an existing file. When these two properties are equal, there are no more data to read.

Methods

The FileStream object exposes a few methods, which are discussed here. The methods for accessing a file's contents are discussed in the following section.

Lock

This method allows you to lock the file you're accessing, or part of it. The syntax of the `Lock` method is the following, where *`position`* is the starting position and *`length`* is the length of the range to be locked:

```
Lock(position, length)
```

To lock the entire file, use this statement:

```
FileStream.Lock(1, FileStream.Length)
```

Seek

This method sets the current position in the file represented by the FileStream object:

```
FileStream.Seek(offset, origin)
```

The new position is *`offset`* bytes from the *`origin`*. In place of the *`origin`* argument, use one of the `SeekOrigin` enumeration members, listed in Table 15.6.

TABLE 15.6: *SeekOrigin* Enumeration

VALUE	EFFECT
`Begin`	The offset is relative to the beginning of the file.
`Current`	The offset is relative to the current position in the file.
`End`	The offset is relative to the end of the file.

SetLength

This method sets the length of the file represented by the FileStream object. Use this method after you have written to an existing file to truncate its length. The syntax of the `SetLength` method is this:

```
FileStream.SetLength(newLength)
```

If the specified value is less than the length of the file, the file is truncated; otherwise, the file is expanded. To completely overwrite the contents of an existing file, call this method as soon as you open the file to set its length to zero — in effect, initializing the file.

THE STREAMWRITER CLASS

The StreamWriter class is the channel through which you send data to a text file. To create a new StreamWriter object, declare a variable of the StreamWriter type. The first overloaded form of the constructor accepts a file's path as an argument and creates a new StreamWriter object for the file:

```
Dim SW As New StreamWriter(path)
```

The new object has the default encoding and the default buffer size. The encoding scheme determines how characters are saved (the default encoding is UTF-8), and the buffer size determines the size of a buffer where data are stored before they're sent to the file. The following statement creates a new StreamWriter object and associates it with the specified file:

```
Dim SW As New StreamWriter("c:\TextFile.txt")
```

Another form of the same constructor creates a new StreamWriter object for the specified file by using the default encoding and buffer size, but it allows you to overwrite existing files. If the *`overwrite`* argument is True, you can overwrite the contents of an existing file.

```
Dim SW As New StreamWriter(path, overwrite)
```

You can also specify the encoding for the StreamWriter with the following form of the constructor:

```
Dim SW As New StreamWriter(path, overwrite, encoding)
```

The last form of the constructor that accepts a file's path allows you to specify both the encoding and the buffer size:

```
Dim SW As New StreamWriter(path, overwrite, encoding, bufferSize)
```

The same forms of the constructor can be used with a FileStream object. The simplest form of its constructor is as follows:

```
Dim SW As New StreamWriter(stream)
```

This form creates a new StreamWriter object for the FileStream specified by the *stream* argument. To use this form of the constructor, you must first create a new FileStream object and then use it to instantiate a StreamWriter object:

```
Dim FS As FileStream
FS = New FileStream(''C:\TextData.txt'', FileMode.Create)
Dim SW As StreamWriter
SW = New StreamWriter(FS)
```

Finally, there are two more forms of the StreamWriter constructor that accept a FileStream object as the first argument. These forms are simply listed here:

```
New StreamWriter(stream, encoding)
New StreamWriter(stream, encoding, bufferSize)
```

After you have created the StreamWriter object, you can call its members to manipulate the underlying file. They are described in the following sections.

NewLine Property

The StreamWriter object provides a handy property, the `NewLine` property, which allows you to change the string used to terminate each line in the file. This terminator is written to the text file by the `WriteLine` method, following the text. The default line-terminator string is a carriage return followed by a line feed (`\r\n`). The StreamReader object doesn't provide a similar property. It reads lines terminated by the carriage return (`\r`), line feed (`\n`), or carriage return/line feed (`\r\n`) characters only.

Methods

To send information to the underlying file, use the following methods of the StreamWriter object.

AutoFlush

This property is a True/False value that determines whether the methods that write to the file (the `Write` and `WriteString` methods) will also flush their buffer. If you set this property to False, the buffer will be flushed when the operating system gets a chance, when the `Flush` method is called, or when you close the FileStream object. When `AutoFlush` is True, the buffer is flushed with every write operation.

Close

This method closes the StreamWriter object and releases the resources associated with it to the system. Always call the `Close` method after you finish using the StreamWriter object. If you have created the StreamWriter object on top of a FileStream object, you must also close the underlying stream too.

Flush

This method writes any data in the buffer to the underlying file.

Write(data)

This method writes the value specified by the *`data`* argument to the Writer object on which it's applied. The `Write` method is overloaded and can accept any data type as an argument. When you pass a numeric value as an argument, the `Write` method stores it to the file as a string. This is the same string you'd get with the number's `ToString` method. To save dates to a text file, you must convert them to strings with one of the methods of the Date data type.

There's one form of the `Write` method I want to discuss here. This overloaded form accepts a string with embedded format arguments, followed by a list of values, one for each argument. The following statement writes a string with two embedded numeric values in it, as shown in the following line:

```
SW.Write(''Your price is ${0} plus ${1} for shipping'', 86.50, 12.99)
Your price is $86.50 plus $12.99 for shipping
```

WriteLine(data)

This method is identical to the `Write` method, but it appends a line break after saving the data to the file. You will find examples on using the StreamWriter class after we discuss the methods of the StreamReader class.

The StreamReader Class

The StreamReader class provides the necessary methods for reading from a text file and exposes methods that match those of the StreamWriter class (the `Write` and `WriteLine` methods).

The StreamReader class's constructor is overloaded. You can specify the FileStream object it will use to read data from the file, the encoding scheme, and the buffer size. The simplest form of the constructor is the following:

```
Dim SR As New StreamReader(FS)
```

This declaration associates the *SR* variable with the file on which the *FS* FileStream object was created. This is the most common form of the StreamReader class's constructor. To prepare your application for reading the contents of the file C:\My Documents\Meeting.txt, use the following statements:

```
Dim FS As FileStream
Dim SR As StreamReader
FS = New FileStream(''c:\My Documents\Meeting.txt'', _
           System.IO.FileMode.OpenOrCreate, System.IO.FileAccess.Write)
SR = New StreamReader(FS)
```

You can also create a new StreamReader object directly on a file, with the following form of the constructor:

```
Dim SR As New StreamReader(path)
```

With both forms of the constructor, you can specify the character encoding with a second argument, as well as a third argument that determines the size of the buffer to be used for the IO operations.

Methods

The StreamReader class provides the following methods for writing data to the underlying file.

Close

The Close method closes the current instance of the StreamReader class and releases any system resources associated with this object.

Peek

The Peek method returns the next character as an integer value, without actually removing it from the input stream. The Peek method doesn't change the current position in the stream. If there are no more characters left in the stream, the value −1 is returned. The Peek method will also return −1 if the current stream doesn't allow peeking.

Read

This method reads a number of characters from the StreamReader class to which it's applied and returns the number of characters read. This value is usually the same as the number of characters you specified unless there aren't as many characters in the file. If you have reached the end of the stream (which is the end of the file), the method returns the value −1. The syntax of the Read method is as follows, where *count* is the number of characters to be read, starting at the *startIndex* location in the file:

```
charsRead = SR.Read(chars, startIndex, count)
```

The characters are stored in the *chars* array of characters, starting at the index specified by the second argument. A simpler form of the Read method reads the next character from the stream and returns it as an integer value, where *SR* is a properly declared StreamReader class:

```
Dim newChar As Integer
newChar = SR.Read()
```

ReadBlock

This method reads a number of characters from a text file and stores them in an array of characters. It accepts the same arguments as the `Read` method and returns the number of characters read.

```
Dim chars(count - 1) As Char
charsRead = SR.Read(chars, startIndex, count)
```

ReadLine

This method reads the next line from the text file associated with the StreamReader class and returns a string. If you're at the end of the file, the method returns the Null value. The syntax of the `ReadLine` method is the following:

```
Dim txtLine As String
txtLine = SR.ReadLine()
```

A text line is a sequence of characters followed by a carriage return (`\r`), line feed (`\n`), or carriage return and line feed (`\r\n`). Notice that the `NewLine` character you might have specified for the specific file with the StreamWriter class is ignored by the `ReadLine` method. The string returned by the method doesn't include the line terminator.

ReadToEnd

The last method for reading characters from a text file reads all the characters from the current position to the end of the file. We usually call this method once to read the entire file with a single statement and store its contents to a string variable. The syntax of the `ReadToEnd` method is as follows:

```
allText = SR.ReadToEnd()
```

To make sure you're reading the entire file with the `ReadToEnd` method, reposition the file pointer at the beginning of the file with the `Seek` method of the underlying stream before calling the `ReadToEnd` method of the StreamReader class:

```
FS.Seek (0, SeekOrigin.Begin)
```

Sending Data to a File

The statements in Listing 15.11 demonstrate how to send various data types to a file. You can place the statements of this listing in a button's `Click` event handler and then open the file with Notepad to see its contents. Everything is in text format, including the numeric values. Don't forget to import the System.IO namespace to your project.

Listing 15.11: Writing Data to a Text File

```
Dim SW As StreamWriter
Dim FS As FileStream
```

```
    FS = New FileStream(''C:\TextData.txt'', FileMode.Create)
    SW = New StreamWriter(FS)
    SW.WriteLine(9.009)
    SW.WriteLine(1 / 3)
    SW.Write(''The current date is '')
    SW.Write(Now())
    SW.WriteLine()
    SW.WriteLine(True)
    SW.WriteLine(New Rectangle(1, 1, 100, 200))
    SW.WriteLine(Color.YellowGreen)
    SW.Close()
    FS.Close()
```

The contents of the `TextData.txt` file that was generated by the statements of Listing 15.11 are shown next:

```
9.009
0.333333333333333
The current date is 9/19/2005 9:06:46 AM
True
{X=1,Y=1,Width=100,Height=200}
Color [YellowGreen]
```

Notice that the `WriteLine` method without an argument inserts a new line character in the file. The statement `SW.Write(Now())` prints the current date but doesn't switch to another line. The following statements demonstrate a more-complicated use of the `Write` method with formatting arguments:

```
Dim BDate As Date = #2/8/1960 1:04:00 PM#
SW.WriteLine(''Your age in years is {0}, in months is {1}, '' & _
             ''in days is {2}, and in hours is {3}.'', _
             DateDiff(DateInterval.year, BDate, Now), _
             DateDiff(DateInterval.month, BDate, Now), _
             DateDiff(DateInterval.day, BDate, Now), _
             DateDiff(DateInterval.hour, BDate, Now))
```

The *SW* variable must be declared with the statements at the beginning of Listing 15.11. The day I tested these statements, the following string was written to the file:

```
Your age in years is 47, in months is 569, in days is 17321, and in hours is 415726.
```

Of course, the data to be stored to a text file need not be hard-coded in your application. The code of Listing 15.12 stores the contents of a TextBox control to a text file. If you compare it to the single statement it takes to write the same data to a file with the FileSystem component, you'll understand how much the My object can simplify file IO operations.

LISTING 15.12: Storing the Contents of a TextBox Control to a Text File

```
Dim SW As StreamWriter
Dim FS As FileStream
FS = New FileStream(''C:\TextData.txt'', FileMode.Create)
SW = New StreamWriter(FS)
SW.Write(TextBox1.Text)
SW.Close ()
FS.Close ()
```

THE BINARYWRITER CLASS

To prepare your application to write to a binary file, you must set up a BinaryWriter object, with the statement shown here, where *FS* is a properly initialized FileStream object:

```
Dim BW As New BinaryWriter(FS)
```

You can also create a new BinaryWriter class directly on a file with the following form of the constructor:

```
Dim BW As New StreamReader(path)
```

To specify the encoding of the text in the binary file, use the following form of the method:

```
Dim BW As New BinaryWriter(FS, encoding)
Dim BW As New BinaryWriter(path, encoding)
```

You can also specify a third argument indicating the size of the buffer to be used with the file input/output operations:

```
Dim BW As New BinaryWriter(FS, encoding, bufferSize)
Dim BW As New BinaryWriter(path, encoding, bufferSize)
```

Methods

The BinaryWriter class exposes the following methods for manipulating binary files.

Close

This method flushes and closes the current BinaryWriter and releases any system resources associated with it.

Flush

This method clears all buffers for the current writer and writes all buffered data to the underlying file.

Seek

This method sets the position within the current stream. Its syntax is the following, where *`origin`* is a member of the `SeekOrigin` enumeration (see Table 15.6) and *`offset`* is the distance from the origin:

```
Seek(offset, origin)
```

Write

The `Write` method writes a value to the current stream. This method is heavily overloaded, but it accepts a single argument, which is the value to be written to the file. The data type of its argument determines how it will be written. The `Write` method can save all the base types to the file in their native format, unlike the `Write` method of the TextWriter class, which stores them as strings.

WriteString

Whereas all other data types can be written to a binary file with the `Write` method, strings must be written with the `WriteString` method. This method writes a length-prefixed string to the file and advances the current position by the appropriate number of bytes. The string is encoded by the current encoding scheme, and the default value is UTF8Encoding.

You will find examples of using the `Write` and `WriteString` methods of the BinaryWriter object at the end of the following section, which describes the methods of the BinaryReader class.

The BinaryReader Class

The BinaryReader class provides the methods you need to read data from a binary file. As you have seen, binary files might also hold text, and the BinaryReader class provides the `ReadString` method to read strings written to the file by the `WriteString` method.

To use the methods of the BinaryReader class in your code, you must first create an instance of the class. The BinaryReader object must be associated with a FileStream object, and the simplest form of its constructor is the following, where *`streamObj`* is the FileStream object:

```
Dim BR As New BinaryReader(streamObj)
```

You can also specify the character-encoding scheme to be used with the *`BR`* object, using the following form of the constructor:

```
Dim BR As New BinaryReader(streamObj, encoding)
```

If you omit the *`encoding`* argument, the default UTF-8Encoding will be used.

Methods

The BinaryReader class exposes the following methods for accessing the contents of a binary file.

Close

This method is the same as the `Close` method of the StreamReader class. It closes the current reader and releases the underlying stream.

PeekChar

This method returns the next available character from the stream without repositioning the current pointer. The character read is returned as an integer, or −1 if there are no more characters to be read from the stream. The name of the method doesn't quite comply with the BinaryReader class, and here's why. Peeking at the next byte makes sense only if the next byte is a character. Reading the first byte of a Double value, for example, wouldn't help you much. A character is usually stored in a single byte (ASCII text), but it can also be stored in 2 bytes (Unicode text). The `PeekChar` method knows how many bytes it must read from the text (they're determined by the current encoding), and it always returns a character, regardless of its size in bytes. The `PeekChar` method's return value is an integer, not a character.

The Read Methods

The BinaryReader class exposes methods for reading the same base data types you can write to a file through the BinaryWriter class. Whereas there's only one `Write` method that writes any data type to the binary file, there are many methods to read the same data. Each method returns a value of the corresponding type (the `ReadBoolean` method returns a Boolean value, the `ReadDecimal` returns a Decimal type, and so on) and only a single value of this type. To read multiple values of the same type, you must call the same method repeatedly. The various methods for reading the base data types from the file are briefly described in Table 15.7.

To use these methods, you're supposed to know the structure of the data stored in the file. A file with a price list, for example, contains the same items for each product. The first two fields are the product's ID and description, followed by the product's price and other pieces of information, which are repeated for each product. If you know the types of values stored in the file, you can call the appropriate methods to read the correct values. If you misread even a single value, none of the following values will be read correctly.

VB 2008 at Work: The RecordSave Project

Let's look at the code for saving structured information to a binary file. In this section, you'll build the RecordSave application, which demonstrates how to store a price list to a disk file and read it later from the same file. The main form of the application is shown in Figure 15.3. The Save Records button creates a few records and then saves them to disk. The Read Records button reads the records from the file and displays them in the ListBox control.

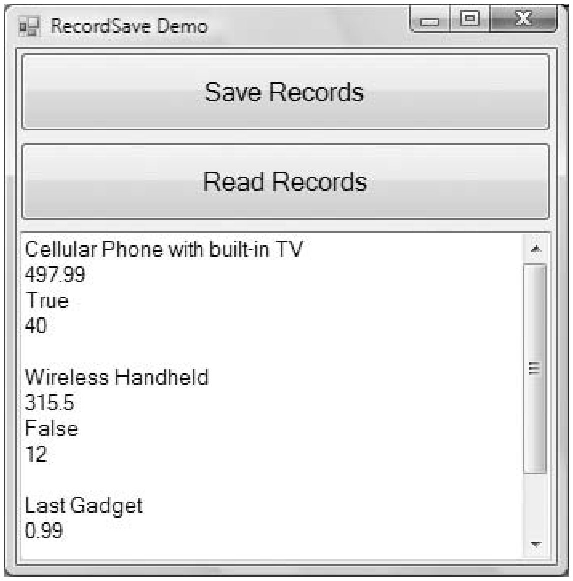
Each record of the price list contains the following fields:

- The product's ID (a String)
- The product's description (a String)
- The product's price (a Single value)
- The product's availability (a Boolean value)
- The minimum reorder quantity (an Integer value)

The program saves each field as a separate entity, using the `Write` method of the BinaryStream class. Only the string is written to the file with the `WriteString` method because we want to be able to read the string back with the `ReadString` method.

TABLE 15.7: The Read Methods of the BinaryReader Class

VALUE	EFFECT
`Read`	Reads the next character from the stream and returns it as an integer value. An overloaded form of the method reads a number of characters into an array of characters.
`ReadBoolean`	Reads and returns a True/False value.
`ReadByte`	Reads and returns a single byte.
`ReadBytes(byteArray, count)`	Reads and returns *count* bytes from the file and stores them into the Byte array passed as the first argument.
`ReadChar`	Reads and returns a character. Depending on how text was stored in the file, the `ReadChar` method might read 1 or 2 bytes (in the case of Unicode text), but it always returns a character.
`ReadChars(charArray, count)`	Reads and returns *`count`* characters from the file and stores them in the character array specified as the first argument.
`ReadDecimal`	Reads and returns a Decimal value from the file.
`ReadDouble`	Reads and returns a Double value from the file.
`ReadInt16`	Reads and returns a short Integer (two-byte) value.
`ReadInt32`	Reads and returns an Integer (four-byte) value.
`ReadInt64`	Reads and returns a Long Integer (eight-byte) value.
`ReadSByte`	Reads and returns a signed byte.
`ReadSingle`	Reads a Single (four-byte) value from the file.
`ReadString`	Reads and returns a string from the file. The string must be stored in the file prefixed by its length. This is how the `WriteString` method stored strings to a text file, so there's nothing you have to do from within your code. If the string isn't prefixed by its length, the `ReadString` method will read a string with the wrong number of characters. The method will interpret the first byte as the string's length.
`ReadUInt16`	Reads and returns an unsigned short Integer (two-byte) value.
`ReadUInt32`	Reads and returns an unsigned Integer (four-byte) value.
`ReadUInt64`	Reads and returns an unsigned long Integer (eight-byte) value.



```
Cellular Phone with built-in TV
497.99
True
40

Wireless Handheld
315.5
False
12

Last Gadget
0.99
```

FIGURE 15.3 The RecordSave project demonstrates how to store records in a binary file.

Because the price list contains many products, you will most likely store it in an array of custom structures. The `Product` structure shown next is a simple, yet quite adequate, structure for our price list:

```
Structure Product
   Dim ProdID As String
   Dim prodDescription As String
   Dim listPrice As Single
   Dim available As Boolean
   Dim minStock As Integer
End Structure
```

The code that writes the structure to a binary file is shown in Listing 15.13.

LISTING 15.13: Saving a Record to a Binary File

```
Private Sub bttnSave_Click(...) Handles bttnSave.Click
   Dim BW As BinaryWriter
   Dim FS As FileStream
   FS = New FileStream("Records.bin", System.IO.FileMode.OpenOrCreate, _
                    System.IO.FileAccess.Write)
   BW = New BinaryWriter(FS)
   BW.BaseStream.Seek(0, SeekOrigin.Begin)
   Dim p As New Product()
' Save first record
   p.ProdID = "100-A39"
   p.prodDescription = "Cellular Phone with built-in TV"
```

```
      p.listPrice = 497.99
      p.available = True
      p.minStock = 40
      SaveRecord(BW, p)
   ' Save second record
      p = New Product()
      p.ProdID = "100-U300"
      p.prodDescription = "Wireless Handheld"
      p.listPrice = 315.5
      p.available = False
      p.minStock = 12
      SaveRecord(BW, p)
   ' Save third record
      p = New Product()
      p.ProdID = "ZZZ"
      p.prodDescription = "Last Gadget"
      p.listPrice = .99
      p.available = True
      p.minStock = 1000
      SaveRecord(BW, p)

      BW.Close()
      FS.Close()
   End Sub
```

The code of the `SaveRecord()` subroutine is shown in Listing 15.14. It accepts as arguments the BinaryWriter class that represents the binary file to which the data will be written and a Product structure to be saved to the file.

LISTING 15.14: *SaveRecord()* Subroutine

```
Sub SaveRecord(ByVal writer As BinaryWriter, ByVal record As Product)
   writer.Write(record.ProdID)
   writer.Write(record.prodDescription)
   writer.Write(record.listPrice)
   writer.Write(record.available)
   writer.Write(record.minStock)
End Sub
```

To read the records stored in the file, set up a BinaryReader associated with the `Records.bin` file and call the appropriate Read method for each field of the record. Because we don't know in advance how many records are in the file, we set up a loop that keeps reading one record at a time, while the current position (property `Position` of the FileStream object) is less than the length of the file (property `Length` of the FileStream object). Listing 15.15 is the code behind the Read Records button.

Listing 15.15: Reading Records from a Binary File

```
Private Sub bttnRead_Click(...) Handles bttnRead.Click
   Dim BR As BinaryReader
   Dim FS As FileStream
   FS = New System.IO.FileStream("Records.bin", FileMode.Open, _
                                 FileAccess.Read)
   BR = New System.IO.BinaryReader(FS)
   BR.BaseStream.Seek(0, SeekOrigin.Begin)
   Dim p As New Product()
   TextBox1.Clear()
   Dim c As Integer
   c = BR.PeekChar
   While FS.Position < FS.Length
      p = Nothing
      ' Read fields and populate structure
      p.ProdID = BR.ReadString
      p.prodDescription = BR.ReadString
      p.listPrice = BR.ReadSingle
      p.available = BR.ReadBoolean
      p.minStock = BR.ReadInt32
      ' Display structure
      ShowRecord(p)
      c = BR.PeekChar
   End While
   BR.Close()
   FS.Close()
End Sub
```

Notice that the product's price is read with the `ReadSingle` method because it was saved as a Single variable. The `ShowRecord()` subroutine appends the fields of the current structure to the TextBox control at the bottom of the form.

Using a custom structure to store the fields simplifies the structure of the application at large, but it doesn't help the file I/O operation much. It's quicker to use the Serializer class to store an entire collection to the file at once, rather than each member of the collection individually. The Serializer class, which is discussed in detail in Chapter 16, addresses many of the file I/O needs of your applications. There will be situations, however, in which you must store widely different pieces of information to a text or binary file, and the information presented in this chapter should be adequate for these situations.

Reading Legacy Data with the FileSystem Object

A convenient feature of the My.Computer.FileSystem component is the `OpenTextFieldParser` method, which reads the fields of a delimited or fixed-width text file. These files are created by many applications when they export their data in text format. The ParseLegacyData project demonstrates how to use the FileSystem object, and the `OpenTextFieldParser` method in specific, to read legacy data. In the project's folder you'll find a Readme file that explains the process in detail.

The FileSystemWatcher Component

FileSystemWatcher is a special component that has no visible interface and allows your application to monitor changes in the file system. You can use the FileSystemWatcher component to monitor changes in the local computer's file system, a network drive, and even a remote computer's file system (as long as the remote machine is running Windows 2000 or a later version). The component exposes a few properties that let you specify what types of changes you want to monitor and the folders/files that will be monitored. Once activated, the FileSystemWatcher component fires an event every time one of the specified items changes.

The *items* you can monitor are folders and files. You can specify the folders you want to monitor as well as the file types to be monitored. You can also specify the types of *actions* you want to monitor; each action fires its own event. The actions you can monitor are the *creation*, *deletion*, and *renaming* of a file or folder and the *modification* of a file. The corresponding events are appropriately named `Changed`, `Created`, `Deleted`, and `Renamed`. There's also a special event, the `Error` event, that is fired when too many changes occur and the FileSystemWatcher component can't keep track of them all. (The internal buffer overflows, and this condition is signaled with the `Error` event.) It's not recommended that you monitor very large folders, or folders with lots of activity.

Properties

To use a FileSystemWatcher component in your project, open the Components tab of the Toolbox and double-click the FileSystemWatcher component's icon. An instance of the component will be placed in your project, and you can set the following properties in the Properties window.

NotifyFilter

This property determines the types of changes you want to monitor. It can have one of the values shown in Table 15.8, which are the members of the `IO.NotifyFilters` enumeration.

Table 15.8: The *NotifyFilters* Enumeration

Value	Description
`Attributes`	The attributes of the file or folder
`CreationTime`	The date of the file's or folder's creation
`DirectoryName`	The directory name
`FileName`	The filename
`LastAccess`	The date of the file's or folder's last access
`LastWrite`	The date of the file's or folder's last edit
`Security`	The security settings of the file or folder
`Size`	The size of the file or folder

You can combine multiple types of changes by using the `Or` operator. The following statement prepares the *`FileSystemWatcher1`* component to monitor for changes in the date and time of a file's last write and last access:

```
FileSystemWatcher1.NotifyFilter =
   IO.NotifyFilters.LastWrite Or
   IO.NotifyFilters.LastAccess
```

The `NotifyFilter` property can also be set in the Properties window, but you can't combine multiple notification types there.

Path, IncludeSubdirectories

Set the `Path` property to the path you want to monitor. The component will watch for changes in files in the specified path. If you want to include the path's subfolders, set the `Include Subdirectories` property to True. The default value of this property is False.

Filter

This property filters the files you want to monitor through a string with wildcards. A `Filter` value of `*.txt` tells the component to monitor for changes in text files only. The default value of the `Filter` property is `*.*`, which includes all the files. An empty string will also have the same effect. Notice that you can't specify multiple extensions with the `Filter` property.

EnableRaisingEvents

To start monitoring for changes in the file system, set the `EnableRaisingEvents` property to True. While the `EnableRaisingEvents` property is True, the FileSystemWatcher component fires an event for the changes you have specified through its properties. To stop monitoring the changes in the file system, set this property to False.

Events

To notify your application about the changes, the FileSystemWatcher component raises the following events, which you can handle from within your code: `Changed`, `Created`, `Deleted`, and `Renamed`. Like all events, they include two arguments: the `sender` and the `e` argument. The second argument of these events carries information about the type of the change through the `ChangeType` property. The `e.ChangeType` property's value is a member of the `IO.WatcherChangeTypes` enumeration: *`All`*, *`Changed`*, *`Created`*, *`Deleted`*, and *`Renamed`*. The `e.FullPath` and `e.Name` properties are the path and filename of the file that was changed, created, or deleted. In the case of a folder, use the `FullPath` property to retrieve the name of the changed folder. Finally, the `Renamed` event's argument exposes the *`OldFullPath`* and *`OldName`* members, which let you retrieve the old path and name of the renamed file.

You can write a common event handler for the `Changed`, `Created`, and `Deleted` events because they share the same arguments. The `Rename` event must have its own handler, because the `e` argument is of a different type.

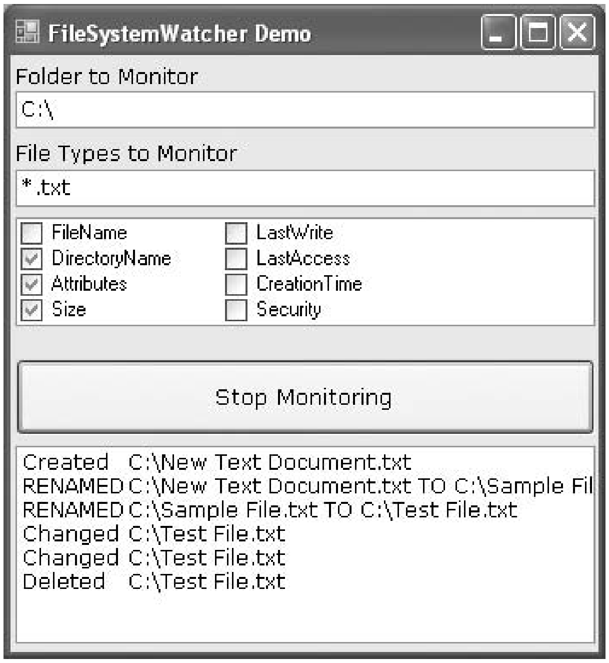
All the changes detected by the FileSystemWatcher component are stored in an internal buffer, which can overflow if too many changes take place in a short period of time. To avoid overflowing the buffer, you should limit the number of files you monitor by setting the `Filter` and `Path` properties appropriately. You should always limit the type of changes. (You'll rarely have to monitor for all types of changes in a folder.) If the buffer overflows, the `Error` event will be raised.

In this event's handler, you can increase the size of the buffer by setting the `InternalBufferSize` property. You can double the buffer's size from within the `Error` event handler to prevent the loss of additional events by using the following statement:

```
FileSystemWatcher1.InternalBufferSize = _
       2 * FileSystemWatcher1.InternalBufferSize
```

VB 2008 at Work: The FileSystemWatcher Project

The FileSystemWatcher project, shown in Figure 15.4, demonstrates how to set up a FileSystemWatcher component and how to process the events raised by the component. The FileSystemWatcher component is initialized when the Start Monitoring button is clicked. This button's `Click` event handler prepares the FileSystemWatcher component to monitor changes in text files on the root of the `C:` drive. I've chosen the root folder because it's easy to locate and it has very few files on most systems. You can create, edit, rename, and then delete a few text files in the root folder to test the application.



```
Created   C:\New Text Document.txt
RENAMED C:\New Text Document.txt TO C:\Sample Fil
RENAMED C:\Sample File.txt TO C:\Test File.txt
Changed C:\Test File.txt
Changed C:\Test File.txt
Deleted   C:\Test File.txt
```

FIGURE 15.4
FileSystemWatcher project

After setting the `Path`, `Filter` and `NotifyFilter` properties, the code sets the component's `EnableRaisingEvents` property to True to start watching for changes. These changes will be signaled though the component's events, which are programmed to print in the ListBox control at the bottom of the form the type of change detected and the name of the corresponding file. The type of change is reported to the event handler through the `ChangeType` member of the `e` argument. When a file is renamed, the program prints both the old and the new name.

The Start Monitoring button is a toggle. When clicked for the first time, its caption changes to Stop Monitoring; if you click it again, it will stop monitoring the file system.

The properties of the FileSystemWatcher component are set in the form's `Load` event, which is shown in Listing 15.16.

LISTING 15.16: Programming the FileSystemWatcher Component

```
Private Sub Form1_Load(ByVal sender As Object, _
             ByVal e As System.EventArgs) Handles Me.Load
   FileSystemWatcher1.Path = "c:\"
   FileSystemWatcher1.IncludeSubdirectories = False
   FileSystemWatcher1.Filter = "*.txt"
   FileSystemWatcher1.NotifyFilter = IO.NotifyFilters.CreationTime Or _
       IO.NotifyFilters.LastWrite Or IO.NotifyFilters.LastAccess Or _
       IO.NotifyFilters.FileName
   FileSystemWatcher1.EnableRaisingEvents = False
End Sub
```

The code behind the button's `Click` event handler (Listing 15.17) toggles the `EnableRaisingEvents` property and the button's caption. When this property is set to True, the FileSystemWatcher component starts monitoring the changes in the file system.

LISTING 15.17: Code of the Start Monitoring Button

```
Private Sub Button1_Click(...) Handles Button1.Click
   If Button1.Text = "Start Monitoring" Then
      FileSystemWatcher1.EnableRaisingEvents = True
      Button1.Text = "Stop Monitoring"
   Else
      FileSystemWatcher1.EnableRaisingEvents = False
      Button1.Text = "Start Monitoring"
   End If
End Sub
```

Now you must program the handlers of the FileSystemWatcher component. You need not program all the events, only the ones you want to monitor. Because the `Changed`, `Created`, and `Deleted` event handlers have the same arguments, you can write a common handler for all three and a separate one for the `Renamed` event. Listing 15.18 details the event handlers of the sample applications.

LISTING 15.18: Event Handlers of the FileSystemWatcher Component

```
Private Sub WatcherHandler(...) Handles _
               FileSystemWatcher1.Changed, _
               FileSystemWatcher1.Created, _
```

```
                FileSystemWatcher1.Deleted
   ListBox1.Items.Add(e.ChangeType & vbTab & e.FullPath)
End Sub

Private Sub FileSystemWatcher1_Renamed(...) Handles _
                FileSystemWatcher1.Renamed
   ListBox1.Items.Add(e.ChangeType & vbTab & _
                e.OldFullPath & " TO " & e.FullPath)
End Sub
```

If you want to handle the `Error` event, you must stop monitoring the file system momentarily, double the value of the `InternalBufferSize` property, and then enable the monitoring again, as shown in Listing 15.19.

Listing 15.19: Programming the FileSystemWatcher's Error Event

```
Private Sub FileSystemWatcher1_Error(..) Handles FileSystemWatcher1.Error
   Dim status As Boolean
   status = FileSystemWatcher1.EnableRaisingEvents
   FileSystemWatcher1.EnableRaisingEvents = False
   FileSystemWatcher1.InternalBufferSize = _
               2 * FileSystemWatcher1.InternalBufferSize
   FileSystemWatcher1.EnableRaisingEvents = status
End Sub
```

Some file operations might cause multiple events. The actions of moving and copying a file from one folder to another fire the `Changed` event several times. The same happens when you create a file with the desktop context menu because several attributes of the new file are set as soon as it's created. To avoid multiple notifications, you should monitor for a few events only. A common use of the FileSystemWatcher component is to detect the creation of a new file in a special folder and act on it (such as when applications or users leave a file to a specific folder or when remote users upload a file to an FTP server). To detect the creation of new files, leave the `NotifyFilter` property to its default value and program the control's `Created` event handler.

The Bottom Line

Handle Files with the My object. The simplest method of saving data to a file is to call one of the `WriteAllBytes` or `WriteAllText` methods of the My.Computer.FileSystem object. You can also use the IO namespace to set up a Writer object to send data to a file, and a Reader object to read data from the file.

Master It Show the statements that save a TextBox control's contents to a file and the statements that reload the same control from the data file. Use the My.Computer.FileSystem component.

Write data to a file with the IO namespace To send data to a file you must set up a FileStream object, which is a channel between the application and the file. To send data to a file, create a StreamWriter or BinaryWriter object on the appropriate FileStream object. Likewise, to read from a file, create a StreamReader or BinaryReader on the appropriate FileStream object. To send data to a file, use the `Write` and `WriteString` methods of the appropriate StreamWriter object. To read data from the file, use the `Read`, `ReadBlock`, `ReadLine`, and `ReadToEnd` methods of the StreamReader object.

Master It Write the contents of a TextBox control to a file using the methods of the IO namespace.

Manipulate folders and files. The IO namespace provides the Directory and File classes, which represent the corresponding entities. Both classes expose a large number of methods for manipulating folders (`CreateDirectory`, `Delete`, `GetFiles`, and so on) and files (`Create`, `Delete`, `Copy`, `OpenRead`, and so on).

Master It How will you retrieve the attributes of a drive, folder, and file using the IO namespace's classes?

Monitor changes in the file system and react to them. The FileSystemWatcher is a special component that allows your application to monitor changes in the file system. You can specify the types of changes you want to monitor by using the `NotifyFilter` property, the types of files you want to monitor by using the `Filter` property, and the path you want to monitor by using the `Path` property. The FileSystemWatcher component fires the `Changed`, `Created`, `Deleted`, and `Renamed` events, depending on the type of change(s) you specified. Once activated, the FileSystemWatcher component fires an event every time one of the specified items changes.

Master It Assume that an application running on a remote computer creates a file in the `E:\Downloaded\Orders` folder for each new order. How will you set up a FileSystemWatcher component to monitor this folder and notify your application about the arrival of each new order?

Chapter 16

Serialization and XML

You have seen how the various collections store items, how to access their elements, and even how to sort and search the collections. To make the most of collections and to use them to store large sets of data, you should also learn how to persist collections to disk files.

Persisting data means to store them on disk at the end of one session and reload them into the same application in a later session. The file with the persisted data can also be shared among different applications and even different computers, as long as there's an application that knows what to do with the data. What good is it to create a large collection if your application can't save it and retrieve it from a disk file in another session?

Since time immemorial, programmers had to write code to save their data to disk. In this chapter, you'll see how to convert objects to streams with a technique known as *serialization*, which is the process of converting arbitrary objects to streams of bytes. After you obtain the serialized stream for a specific object, you can persist the object to disk, as well as read it back. The process of reconstructing an object from its serialized form is called *deserialization*. Together, serialization and deserialization allow you to store objects of any type to disk files and reuse them at a later session. Serialization is not limited to saving data to files, however. You can serialize objects to any stream, including a memory stream, a network stream, or even a cryptographic stream that emits encrypted data.

In this chapter, you'll learn how to do the following:

- Serialize objects and collections into byte streams
- Deserialize streams and reconstruct the original objects
- Create XML files in your code

Understanding Serialization Types

There are three types of serialization: binary serialization, SOAP (Simple Object Access Protocol or Service Oriented Architecture Protocol) serialization, and XML (Extensible Markup Language) serialization. Binary and SOAP serialization are very similar; XML serialization is a little different, but it allows you to customize the serialization process.

Binary serialization is performed with the BinaryFormatter class, and it converts the values of the object's properties into a binary stream. The result of the binary serialization is compact and efficient. However, binary-serialized objects can be used only by applications that have access to the class that produced the objects and can't be used outside .NET. Another limitation of binary serialization is that the output it produces is not human readable, and you can't do much with a file that contains a binary serialized object without access to the original application's code. Because binary serialization is very compact and very efficient, it's used almost exclusively to persist objects between sessions of an application or between applications that share the same classes. For example, you can create an ArrayList of Person objects, serialize them to a file, and reload the collection of the serialized objects from the file at a later time or another session.

SOAP serialization produces a SOAP-compliant envelope that describes its contents and serializes the objects in SOAP-compliant format. SOAP-serialized data is suitable for transmission to any system that understands SOAP, and it's implemented by the SoapFormatter class. Unlike binary serialization, SOAP-serialized data are firewall friendly, and SOAP serialization is used to remote objects to a server on a different domain.

XML serialization, which is implemented by the XmlSerializer class, is a different type of serialization. The XmlSerializer class serializes public, read-write properties only. Because it doesn't serialize the private members or read-only properties, XmlSerializer doesn't quite preserve the state of the object. Another limitation of XmlSerializer is that it doesn't serialize collections, with the exception of arrays and ArrayLists. However, it can be customized with the use of attributes (special keywords that prefix the members of a class), and it's as close as we can get to a universal data-exchange format. As you will see at the end of this chapter, you can open the file generated by XmlSerializer with an XML editor and not only view, but also edit it. In the following chapter, you will learn how to query the contents of an XML file.

Using Binary and SOAP Serialization

Let's start with binary serialization, which is implemented in the following namespace (you must import it into your application):

```
Imports System.Runtime.Serialization.Formatters.Binary
```

This namespace isn't loaded by default, and you must add a reference to the corresponding namespace. Right-click the project's name in the Solution Explorer and choose Add Reference from the context menu. In the Add Reference dialog box that appears, select the same namespace as in the `Imports` statement shown earlier.

To use a SOAP serializer, add the appropriate reference and import the following namespace:

```
Imports System.Runtime.Serialization.Formatters.Soap
```

After the appropriate namespace has been imported to the current project, you can serialize individual objects as well as collections of objects. Let's start with single-object serialization.

Serializing Individual Objects

To serialize an object, you must call the `Serialize` method of the System.Runtime.Serialization .Formatters.Binary object. First, declare an object of this type with a statement like the following:

```
Dim BFormatter As New BinaryFormatter()
```

The BinaryFormatter class persists objects in binary format. You can also persist objects in text format by using the SoapFormatter class. SoapFormatter persists the objects in XML format, which is quite verbose, and the corresponding files are considerably lengthier. To use the SoapFormatter class, declare a `SoapFormatter` variable with the following statement:

```
Dim SFormatter As Soap.SoapFormatter
```

SOAP is a protocol for accessing objects over HTTP — in other words, it's a protocol that allows the encoding of objects in text format. SOAP was designed to enable distributed computing over the Internet. SOAP uses text to transfer all kinds of objects, including images and audio, and therefore it's not rejected by firewalls.

The methods of BinaryFormatter and SoapFormatter are equivalent, so I will use Binary Formatter in the examples of this section. To serialize an object, call the `Serialize` method of the appropriate formatter, whose syntax is the following, where ***`stream`*** is a variable that represents a stream and ***`object`*** is the object you want to serialize:

```
BFormatter.Serialize(stream, object)
```

Because we want to persist our objects to disk files, the ***`stream`*** argument represents a stream to a file where the serialized data will be stored. The File object and its methods were discussed in detail in Chapter 15, ''Accessing Folders and Files.'' Here I will explain only briefly the statements we'll use to store data to a disk file and read it back. The following statements create such a Stream object:

```
Dim saveFile As FileStream
saveFile = File.Create("C:\Shapes.bin")
```

The ***`saveFile`*** variable represents the stream to a specific file on the disk, and the `Create` method of the same variable creates a stream to this file.

After you have set up the Stream and BinaryFormatter objects, you can call the `Serialize` method to serialize any object. To serialize a Rectangle object, for example, use the following statements:

```
Dim R As New Rectangle(0, 0, 100, 100)
BFormatter.Serialize(saveFile, R)
```

Listing 16.1 serializes two Rectangle objects to the `Shapes.bin` file in the root folder. The file's extension can be anything. Because the file is binary, I used the `BIN` extension.

LISTING 16.1: Serializing Distinct Objects

```
Dim R1 As New Rectangle()
R1.X = 1
R1.Y = 1
R1.Size.Width = 10
R1.Size.Height = 20
Dim R2 As New Rectangle()
R2.X = 10
R2.Y = 10
R2.Size.Width = 100
R2.Size.Height = 200
Dim saveFile As FileStream
saveFile = File.Create("C:\SHAPES.BIN")
Dim formatter As BinaryFormatter
formatter = New BinaryFormatter()
formatter.Serialize(saveFile, R1)
formatter.Serialize(saveFile, R2)
saveFile.Close()
```

Notice that the Serialize method serializes a single object at a time. To save the two rectangles, the code calls the Serialize method once for each rectangle. To serialize multiple objects with a single statement, you must create a collection, append all the objects to the collection, and then serialize the collection itself, as explained in the following section. If you serialize multiple objects of different types into the same stream, you can't deserialize them unless you know the order in which the objects were serialized and deserialize them in the same order.

Deserializing Individual Objects

To deserialize a serialized object, you must create a new binary or SOAP formatter object and call its Deserialize method. Because the serialized data doesn't contain any information about the original object, you can't reconstruct the original object from the serialized data, unless you know the type of object that was serialized. Deserialization is always more difficult than serialization, and you have to know what you're doing. Whereas the Serialize method will always serialize the object you pass as an argument, the Deserialize method won't reconstruct the original object unless you know the type of the object you're deserializing. The Shapes.bin file of Listing 16.1 contains the serialized versions of two Rectangle objects. The Deserialize method needs to know that it will deserialize two Rectangle objects. If you attempt to extract the information of this file into any other type of object, a runtime exception will occur.

To deserialize the contents of a file, create a formatter object as you did for the serialization process, by using one of the following statements (depending on the type of serialization):

```
Dim SFormatter As Soap.SoapFormatter
Dim BFormatter As BinaryFormatter
```

Then establish a stream to the source of the serialized data, which in our case is the Shapes.bin file:

```
Dim Strm As New FileStream("..\Objects.Bin", FileMode.Open)
```

Finally, deserialize the stream's data by calling the Deserialize method. The Deserialize method accepts a single argument, which is the stream from which the data is coming, and it returns an object, which is the persisted object. We usually cast the object returned by the Deserialize method to the appropriate type:

```
Dim R1, R2 As Rectangle
R1 = CType(SFormatter.Deserialize(Strm), Rectangle)
R2 = CType(SFormatter.Deserialize(Strm), Rectangle)
```

You can serialize as many objects as you like into the same stream, one after the other, and read them back in the same order. You can open the files with the serialized data and view their data. The contents of a SOAP file with two serialized Rectangle objects is shown next:

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
```

```
<SOAP-ENV:Body>
<a1:Rectangle id="ref-1" xmlns:a1="http://schemas.microsoft.com/clr/nsassem
         /System.Drawing/System.Drawing%
   2C%20Version%3D2.0.3600.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%
   3Db03f5f7f11d50a3a">
<x>0</x>
<y>0</y>
<width>100</width>
<height>100</height>
</a1:Rectangle>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<a1:Rectangle id="ref-1" xmlns:a1="http://schemas.microsoft.com/clr/nsassem
          /System.Drawing/System.Drawing%
   2C%20Version%3D2.0.3600.0%2C%20Culture%3D
      neutral%2C%20PublicKeyToken%3Db03f5f7f11d50a3a">
<x>65</x>
<y>30</y>
<width>19</width>
<height>199</height>
</a1:Rectangle>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

You'll never have to create your own SOAP files, so don't panic if they look complicated. There are, however, a few points of interest. First, you see a reference to the System.Drawing class, which indicates that the serialized data can't be used outside the context of the Framework; this file contains serialized data describing an instance of a specific class. The section of the file with the data contains the values of the two basic properties of the Rectangle object. Second, the SOAP format uses an XML notation to delimit its fields, but it's not an XML file. If you attempt to open the same file with Internet Explorer, you'll see a message indicating that it's not a valid XML document. Don't worry if you're not familiar with XML; I'll discuss this format a little later in this chapter. Just keep in mind that SOAP is not the same as XML, even though they're similar. If you remove the SOAP-related tags, you'll be left with a valid XML file.

Serializing Collections

Serializing a collection is quite similar to serializing any single object, because collections are objects themselves. The second argument to the `Serialize` method is the object you want to serialize, and this object can be anything, including a collection. To demonstrate the serialization of an ArrayList, we'll modify the previous code a little, so that instead of persisting individual items, it will persist an entire collection. Declare the two Rectangle objects as before, but append them to an ArrayList collection. Then add a few Color values to the collection, as shown in Listing 16.2, which serializes an ArrayList collection to the file `C:\ShapesColors.bin`.

LISTING 16.2: Serializing a Collection

```
Private Sub Button2_Click(...) Handles Button2.Click
   Dim R1 As New Rectangle()
   R1.X = 1
   R1.Y = 1
   R1.Width = 10
   R1.Height = 20
   Dim R2 As New Rectangle()
   R2.X = 10
   R2.Y = 10
   R2.Width = 100
   R2.Height = 200
   Dim shapes As New ArrayList()
   shapes.Add(R1)
   shapes.Add(R2)
   shapes.Add(Color.Chartreuse)
   shapes.Add(Color.DarkKhaki.GetBrightness)
   shapes.Add(Color.DarkKhaki.GetHue)
   shapes.Add(Color.DarkKhaki.GetSaturation)
   Dim saveFile As FileStream
   saveFile = File.OpenWrite("C:\ShapesColors.bin")
   saveFile.Seek(0, SeekOrigin.End)
   Dim formatter As BinaryFormatter = New BinaryFormatter()
   formatter.Serialize(saveFile, shapes)
   saveFile.Close()
   MsgBox("ArrayList serialized successfully")
End Sub
```

The last three calls to the `Add` method add the components of another color to the collection. Instead of adding the color as is, we're adding three color components, from which we can reconstruct the color `Color.DarkKhaki`. Then we proceed to save the entire collection to a file by using the same statements as before. The difference is that we don't call the `Serialize` method for each object. We call it once and pass the entire ArrayList as an argument.

Deserializing Collections

To read a file with the description of an object that has been persisted with the `Serialize` method, you simply call the formatter object's `Deserialize` method and assign the result to an appropriately declared variable. In the preceding example, the value returned by the `Deserialize` method must be assigned to an ArrayList variable. The syntax of the `Deserialize` method is the following, where *`str`* is a Stream object pointing to the file with the data:

```
object = Bformatter.Deserialize(str)
```

Because the Deserialize method returns an Object variable, you must cast it to the ArrayList type with the CType() function. To use the Deserialize method, declare a variable that can hold the value returned by the method. If the data to be deserialized is a Rectangle, declare a Rectangle variable. If it's a collection, declare a variable of the same collection type. Then call the Deserialize method and cast the value returned to the appropriate type. The following statements outline the process:

```
Dim object As <type>
{ code to set up a Stream variable (str) and BinaryFormatter}
object = CType(Bformatter.Serialize(str), <type>)
```

Listing 16.3 is the code that retrieves the items from the ShapesColors.bin file and stores them into an ArrayList. I added a few statements to print all the items of the ArrayList.

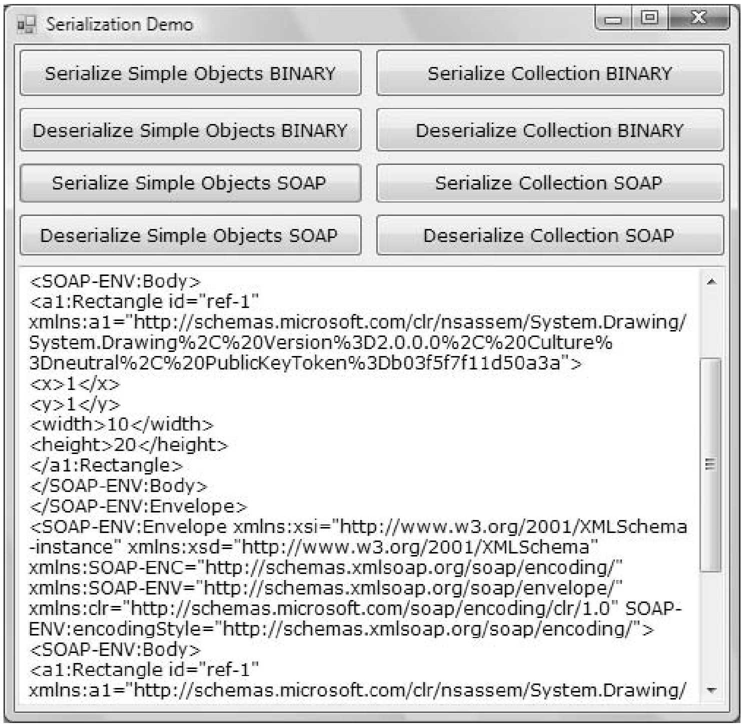
LISTING 16.3: Deserializing a Collection

```
Private Sub Button1_Click(...) Handles Button1.Click
   Dim readFile As FileStream
   readFile = File.OpenRead("C:\ShapesColors.bin")
   Dim BFormatter As BinaryFormatter
   BFormatter = New BinaryFormatter()
   Dim Shapes As New ArrayList()
   Dim R1 As Rectangle
   Shapes = CType(BFormatter.Deserialize(readFile), ArrayList)
   Dim i As Integer
   TextBox1.AppendText("The ArrayList contains " & Shapes.Count & _
                    " objects " & vbCrLf & vbCrLf)
   For i = 0 To Shapes.Count - 1
      TextBox1.AppendText(Shapes(i).ToString & vbCrLf)
   Next
End Sub
```

You can find the code presented in this section in the SimpleSerialization sample project. The application's main form contains buttons to serialize and deserialize both individual objects and collections in binary and SOAP formats. The SOAP-serialized data are displayed in a TextBox control on the form, as shown in Figure 16.1.

The serialization classes can substantially simplify an application's code because they allow you to save objects, or large sets of objects, with a few statements. Moreover, you can easily deserialize the data from within any other application. A fairly common task in programming is to save large sets of data to a file and read them back in a later session. You've seen two such examples so far: the WordFrequencies application of Chapter 14, ''Storing Data in Collections,'' persisted a collection of words and their frequencies to a disk file, and the Globe application of Chapter 9, ''The TreeView and ListView Controls,'' persisted the Nodes collection of a TreeView control to a file between sessions. I didn't discuss the code in the corresponding chapters, and this is a good place to explain the code that serializes HashTables and the Nodes collection.

FIGURE 16.1
The SimpleSerialization project demonstrates the process of binary and SOAP serialization.


Serialization Demo
Serialize Simple Objects BINARY
Serialize Collection BINARY
Deserialize Simple Objects BINARY
Deserialize Collection BINARY
Serialize Simple Objects SOAP
Serialize Collection SOAP
Deserialize Simple Objects SOAP
Deserialize Collection SOAP

```
<SOAP-ENV:Body>
<a1:Rectangle id="ref-1"
xmlns:a1="http://schemas.microsoft.com/clr/nsassem/System.Drawing/
System.Drawing%2C%20Version%3D2.0.0.0%2C%20Culture%
3Dneutral%2C%20PublicKeyToken%3Db03f5f7f11d50a3a">
<x>1</x>
<y>1</y>
<width>10</width>
<height>20</height>
</a1:Rectangle>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema
-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<a1:Rectangle id="ref-1"
xmlns:a1="http://schemas.microsoft.com/clr/nsassem/System.Drawing/
```

Persisting a HashTable

In this section, you'll look at the code for persisting a HashTable to disk. The code I present in this section belongs to the WordFrequencies project, which was presented in Chapter 14. The WordFrequencies application calculates word frequencies, stores the results to a HashTable, and persists them to a file between sessions. Figure 16.2 shows the interface of the WordFrequencies sample project.

This process allows us to process one document at a time yet accumulate the results over many documents. Each unique word is a key to the HashTable, and the word's count is the corresponding item's value. The application's main menu, the Frequency Table menu, contains four commands, which save the HashTable to, and read it from, a text file and a binary file. Table 16.1 shows the four commands of the menu.

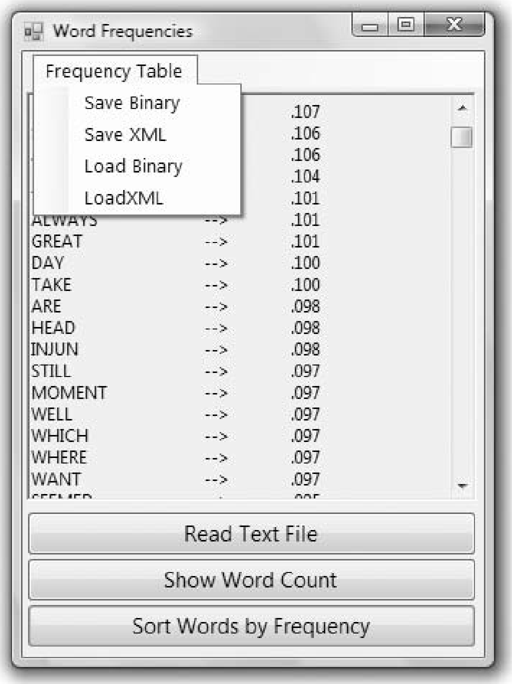
By the way, the Save XML and Load XML commands use a Soap Formatter, but the files they produce (or consume) are XML files, so I've chosen not to use the SOAP term in the menu commands. The code behind the Save Binary command is shown in Listing 16.4. The code is quite simple: It creates an instance of the BinaryFormatter class (variable *`Formatter`*) and uses its `Serialize` method to persist the entire HashTable with a single statement.

LISTING 16.4: Persisting the HashTable to a Binary File

```
Private Sub SaveBin(...) Handles SaveBinary.Click
   Dim saveFile As FileStream
   SaveFileDialog1.DefaultExt = "BIN"
```

```
        If SaveFileDialog1.ShowDialog = DialogResult.OK Then
           saveFile = File.OpenWrite(SaveFileDialog1.FileName)
           saveFile.Seek(0, SeekOrigin.End)
           Dim Formatter As BinaryFormatter = New BinaryFormatter()
           Formatter.Serialize(saveFile, WordFrequencies)
           saveFile.Close()
        End If
     End Sub
```

FIGURE 16.2
The WordFrequencies project uses serialization to persist word frequencies between sessions.



```
                            .107
                            .106
                            .106
                            .104
                            .101
ALWAYS          -->         .101
GREAT           -->         .101
DAY             -->         .100
TAKE            -->         .100
ARE             -->         .098
HEAD            -->         .098
INJUN           -->         .098
STILL           -->         .097
MOMENT          -->         .097
WELL            -->         .097
WHICH           -->         .097
WHERE           -->         .097
WANT            -->         .097
```

TABLE 16.1: The Four Commands of the Frequency Table Menu

COMMAND	**EFFECT**
Save Binary	Saves the HashTable to a binary file with the default extension BIN
Load Binary	Loads the HashTable with data from a binary file
Save XML	Saves the HashTable to a text file with the default extension XML
Load XML	Loads the HashTable with data from a text file

The equivalent Load Binary command is just as simple. It sets up a BinaryFormatter object and calls its `Deserialize` method to read the data. The code of the Save XML command (Listing 16.5) sets up a SoapFormatter object and uses its `Serialize` method to persist the HashTable. The code that reads the data from the file and populates the HashTable is equally simple, and it's shown in Listing 16.6.

Listing 16.5: Persisting the HashTable to a Text File

```
Private Sub SaveText(...) Handles SaveText.Click
   Dim saveFile As FileStream
   SaveFileDialog1.DefaultExt = "XML"
   If SaveFileDialog1.ShowDialog = DialogResult.OK Then
      saveFile = File.OpenWrite(SaveFileDialog1.FileName)
      saveFile.Seek(0, SeekOrigin.End)
      Dim Formatter As Soap.SoapFormatter = New Soap.SoapFormatter()
      Formatter.Serialize(saveFile, WordFrequencies)
      saveFile.Close()
   End If
End Sub
```

Listing 16.6: Loading a HashTable from a Text File

```
Private Sub LoadText(...) Handles LoadText.Click
   Dim readFile As FileStream
   OpenFileDialog1.DefaultExt = "XML"
   If OpenFileDialog1.ShowDialog = DialogResult.OK Then
      readFile = File.OpenRead(OpenFileDialog1.FileName)
      Dim Formatter As Soap.SoapFormatter
      Formatter = New Soap.SoapFormatter()
      WordFrequencies = CType(Formatter.Deserialize(readFile), HashTable)
      readFile.Close
   End If
End Sub
```

You can open the binary file with a text editor, and you will see the words but not the numeric values, which are stored in binary format. If you open the text file, you will see a SOAP file with the words and their counts. The words are in the first half of the file, and their counts are in the second half. Here are the first few lines of this file (I omitted the headers):

```
<item id="ref-5" xsi:type="SOAP-ENC:string">A</item>
<item id="ref-6" xsi:type="SOAP-ENC:string">ABADDIRS</item>
<item id="ref-7" xsi:type="SOAP-ENC:string">ABANDON</item>
<item id="ref-8" xsi:type="SOAP-ENC:string">ABANDONED</item>
<item id="ref-9" xsi:type="SOAP-ENC:string">ABANDONING</item>
```

The corresponding counts are the following:

```
<item xsi:type"="xsd:int"">2064</item>
<item xsi:type=""xsd:int"">1</item>
<item xsi:type=""xsd:int"">5</item>
<item xsi:type=""xsd:int"">10</item>
<item xsi:type=""xsd:int"">2</item>
```

Most of us shouldn't really care how the `Serialize` method stores the data to the file (SOAP or binary), as long as the `Deserialize` method can read them back and load them into an object so that we don't have to write code to parse this file.

Persisting a TreeView's Nodes

In Chapter 9, you learned how to populate the TreeView and ListView controls, how to manipulate them at runtime, and how to sort the ListView control in any way you want. But what good are all these techniques unless you can save the tree's nodes or the ListViewItems to a disk file and then reuse them in a later session?

It would be nice if the TreeNode object were serializable — you could serialize the root node and all the nodes under it with a single call to the `Serialize` method. Unfortunately, this is not the case. Well, how about subclassing the TreeNode object? Create a new class that inherits from the TreeNode class and is serializable. This is an option, but it's not simple.

The main reason that the `Nodes` collection can't be easily serialized is that a node's `Tag` property can store an object that's not serializable. To serialize the `Nodes` collection, we must make a few assumptions that are specific to an application, or a class of applications, but not to all TreeView controls. In this section, I assume that the `Tag` property won't be persisted. You can easily modify the code to persist nodes whose tags are strings, numbers, or any serializable object. The code presented in this section can be used to persist most TreeView controls, but not any TreeView control you can throw at it. Just remember that serialization is limited to the objects that are themselves serializable, and not all classes are serializable.

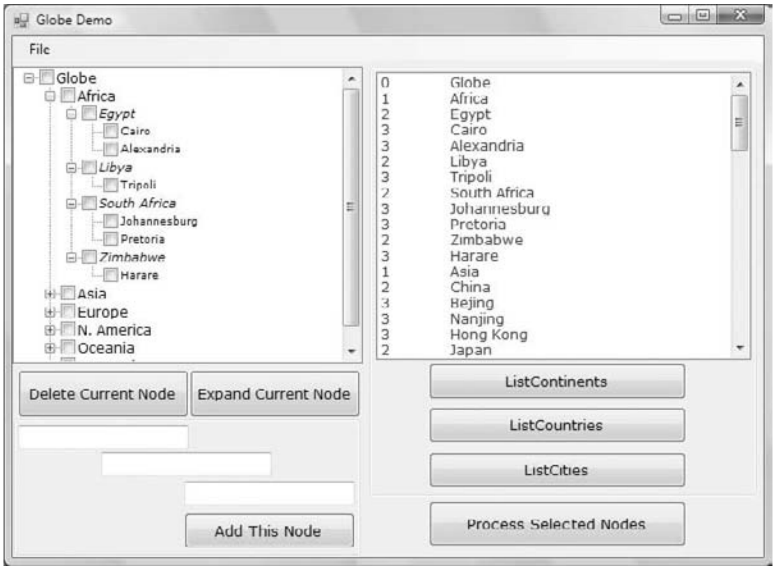
To serialize the nodes of a TreeView control, we'll store the individual nodes in an ArrayList and then serialize the ArrayList, as discussed earlier in this chapter. The code of this section serializes the strings displayed on a TreeView control. You know how to scan the nodes of a TreeView control, and the code for serializing the control's nodes seems trivial. It's not quite so.

The ArrayList has a linear structure: Each item is independent of any other. The TreeView control, however, has a hierarchical structure. Most of its nodes are children of other nodes, as well as parents of other nodes. Therefore, we must store not only the data (strings), but also their structure. To store this information, we'll create a new structure with two fields: one for the node's value and another one for the node's indentation:

```
<Serializable()> Structure sNode
    Dim node As String
    Dim level As Integer
End Structure
```

We want to be able to serialize this structure, so we must prefix it with the `<Serializable>` attribute. The *`level`* field is the node's indentation. The level field of all root nodes is zero. The nodes immediately under the root have a level of 1, and so on. To serialize the TreeView control, we'll iterate through its nodes and store each node to an `sNode` variable. Each time we switch to a child node, we'll increase the current value of the `level` variable by one; each time we move up to a parent node, we'll decrease the same value accordingly. All the `sNode` structures will be added to an ArrayList, which will then be serialized.

Likewise, when we read the ArrayList from the disk file, we must reconstruct the original tree. Items with a level value of zero are root nodes. The first item with a level value of 1 is the first child node under the most recently added root node. As long as the level field doesn't change, the new nodes are added under the same parent. When this value increases, we must create a new child node under the current node. When this value decreases, we must move up to the current node's parent and create a new child under it. The only complication is that a level value might decrease by more than one. In this case, we must move up to the parent's parent — or even higher in the hierarchy. Figure 16.3 shows a typical TreeView control and how its nodes are stored in the ArrayList.



```
0    Globe
1    Africa
2    Egypt
3    Cairo
3    Alexandria
2    Libya
3    Tripoli
2    South Africa
3    Johannesburg
3    Pretoria
2    Zimbabwe
3    Harare
1    Asia
2    China
3    Bejing
3    Nanjing
3    Hong Kong
2    Japan
```

FIGURE 16.3
The structure of the nodes of a TreeView control

The control on the left is a TreeView control, populated at design time. The control on the right is a ListBox control with the items of the ArrayList. The first column is the level field (the node's indentation), whereas the second column is the node's text.

Now we can look at the code for serializing the control. The code presented in this section is part of the Globe project — namely, it's the code behind the Save Nodes and Load Nodes commands of the File menu. The File ➢ Save Nodes command prompts the user with the File Save dialog box for the path of a file in which the nodes will be stored. Then it calls the `CreateList()` subroutine, passing the root node of the control and the path of the file where the items will be stored. Listing 16.7 shows this menu item's `Click` event handler.

LISTING 16.7: File ➤ Save Nodes Menu Item's Event Handler

```
Private Sub FileSave_Click(...) Handles FileSave.Click
    SaveFileDialog1.DefaultExt = "XML"
    If SaveFileDialog1.ShowDialog = DialogResult.OK Then
        CreateList(GlobeTree.Nodes(0), SaveFileDialog1.FileName)
    End If
End Sub
```

The `CreateList()` subroutine goes through the subnodes of the root node and stores them into the *`GlobeNodes`* ArrayList. This ArrayList is declared at the form level with the following statement:

```
Dim GlobeNodes As New ArrayList()
```

`CreateList()` is a recursive subroutine that scans the immediate children of the node passed as an argument. If a child node contains its own children, the subroutine calls itself to iterate through the children. This process may continue to any depth. The code of the subroutine is shown in Listing 16.8.

LISTING 16.8: The *CreateList()* Subroutine

```
Sub CreateList(ByVal node As TreeNode, ByVal fName As String)
    Static level As Integer
    Dim thisNode As TreeNode
    Dim myNode As sNode
    Application.DoEvents()
    myNode.level = level
    myNode.node = node.Text
    GlobeNodes.Add(myNode)
    If node.Nodes.Count > 0 Then
        level = level + 1
        For Each thisNode In node.Nodes
            CreateList(thisNode, fName)
        Next
        level = level - 1
    End If
    SaveNodes(fName)
End Sub
```

After the ArrayList has been populated, the code calls the `SaveNodes()` subroutine, which persists the ArrayList to a disk file. The path of the file is the second argument of the `CreateList()`

subroutine. `SaveNodes()`, shown in Listing 16.9, is a straightforward subroutine that serializes the *`GlobeNodes`* ArrayList to disk. (The process of serializing ArrayLists and other collections was discussed earlier in this chapter.)

LISTING 16.9: The *SaveNodes()* Subroutine

```
Sub SaveNodes(ByVal fName As String)
    Dim formatter As SoapFormatter
    Dim saveFile As FileStream
    saveFile = File.Create(fName)
    formatter = New SoapFormatter()
    formatter.Serialize(saveFile, GlobeNodes)
    saveFile.Close()
End Sub
```

The File ➢ Load Nodes command prompts the user for a filename and then calls the `LoadNodes()` subroutine to read the ArrayList persisted in this file and load the control with its nodes. The `Click` event handler of the Load Nodes command is shown in Listing 16.10.

LISTING 16.10: Reading the Persisted Nodes

```
Private Sub FileLoad_Click(...) Handles FileLoad.Click
    OpenFileDialog1.DefaultExt = "XML"
    If OpenFileDialog1.ShowDialog = DialogResult.OK Then
        LoadNodes(GlobeTree, OpenFileDialog1.FileName)
    End If
End Sub
```

The `LoadNodes()` subroutine loads the items read from the file into the *`GlobeNodes`* ArrayList and then calls the `ShowNodes()` subroutine to load the nodes from the ArrayList onto the control. The `LoadNodes()` subroutine is shown in Listing 16.11.

LISTING 16.11: Loading the *GlobeNodes* ArrayList

```
Sub LoadNodes(ByVal TV As TreeView, ByVal fName As String)
    TV.Nodes.Clear()
    Dim formatter As SoapFormatter
    Dim openFile As FileStream
    openFile = File.Open(fName, FileMode.Open)
    formatter = New SoapFormatter()
    GlobeNodes = CType(formatter.Deserialize(openFile), ArrayList)
    openFile.Close()
    ShowNodes(TV)
End Sub
```

The most interesting code is in the ShowNodes() subroutine, which goes through the items in the ArrayList and re-creates the original structure of the TreeView control. At each iteration, the subroutine examines the value of the item's level field. If it's the same as the current node's level, the new node is added under the same node as the current node (we're on the same indentation level.) If the current item's level field is larger than the current node's level, the new node is added under the current node (it's a child of the current node.) Finally, if the current item's level field is smaller than the current node's level, the code moves up to the parent of the current node. This step can be repeated several times, depending on the difference between the two levels. If the current node's level is 4 and the level field of the new node is 1, the code will move up three levels. (It will actually be added under the most recent root node.) Listing 16.12 is the code of the ShowNodes() subroutine.

LISTING 16.12: The *ShowNodes()* Subroutine

```
Sub ShowNodes(ByVal TV As TreeView)
    Dim o As Object
    Dim currNode As TreeNode
    Dim level As Integer = 0
    Dim fromLowerLevel As Integer

    Dim i As Integer
    For i = 0 To GlobeNodes.Count - 1
        o = GlobeNodes(i)
        If o.level = level Then
            If currNode Is Nothing Then
                currNode = TV.Nodes.Add(o.node.ToString)
            Else
                currNode = currNode.Parent.Nodes.Add(o.node.ToString)
            End If
        Else
            If o.level > level Then
                currNode = currNode.Nodes.Add(o.node.ToString)
                level = o.level
            Else
                While o.level <= level
                    currNode = currNode.Parent
                    level = level - 1
                End While
                currNode = currNode.Nodes.Add(o.node.ToString)
            End If
        End If
        TV.ExpandAll()
        Application.DoEvents()
    Next
End Sub
```

Why did I use a SoapFormatter and not a BinaryFormatter to persist the data? I just wanted to see the structure of the data in text format. You will probably change the code to save the data in binary format because it's much more compact. Of course, XML and SOAP are quite fashionable these days. You can also claim that the data can be read on any other system and that you're following industry standards. I suggest that you use mostly the binary format for storing application data. If you want to exchange data with another system, use the XmlSerialization class instead.

The technique shown here persists the strings displayed on the control, and it works with most applications. If you're using a TreeView control to store objects, you must adjust the code of this section to persist the objects, not just strings. It goes without saying that all objects you store to the TreeView control must be serializable; if not, you won't be able to serialize the `Nodes` collection.

If you're wondering what the persisted nodes look like in the XML file, here's how the first few items of the Globe tree are persisted.

```
- <item xsi:type="a3:NodeSerializer+sNode"
    xmlns:a3="http://schemas.microsoft.com/clr/
            nsassem/Globe/Globe%2C%20
            Version%3D1.0.638.15776%2C%20
            Culture%3Dneutral%2C%20
            PublicKeyToken%3Dnull">
  <node id="ref-4">Globe</node>
  <level>0</level>
  </item>
- <item xsi:type="a3:NodeSerializer+sNode"
    xmlns:a3="http://schemas.microsoft.com/clr/
             nsassem/Globe/Globe%2C%20
             Version%3D1.0.638.15776%2C%20
             Culture%3Dneutral%2C%20
             PublicKeyToken%3Dnull">
  <node id="ref-5">Africa</node>
  <level>1</level>
  </item>
- <item xsi:type="a3:NodeSerializer+sNode"
    xmlns:a3="http://schemas.microsoft.com/clr/
             nsassem/Globe/Globe%2C%20
             Version%3D1.0.638.15776%2C%20
             Culture%3Dneutral%2C%20
             PublicKeyToken%3Dnull">
  <node id="ref-6">Egypt</node>
  <level>2</level>
  </item>
- <item xsi:type="a3:NodeSerializer+sNode"
    xmlns:a3="http://schemas.microsoft.com/clr/
             nsassem/Globe/Globe%2C%20
             Version%3D1.0.638.15776%2C%20
             Culture%3Dneutral%2C%20
             PublicKeyToken%3Dnull">
  <node id="ref-7">Alexandria</node>
  <level>3</level>
  </item>
```

Persisting the items of a ListView control is even simpler. You must create a new structure that reflects the structure of each row (the item and subitems of each row) and then create an ArrayList with items of this type. Persisting the ArrayList is straightforward, and so is the loading of the control, because the ListView control doesn't have a hierarchical structure. Its items are organized in a linear fashion, just like the items of the ArrayList.

To reuse the subroutines that serialize and deserialize the nodes of a TreeView control, you can create a new class that exposes the `CreateList()` and `LoadNodes()` subroutines as methods. The other two subroutines that save the ArrayList to disk and load a disk file into the ArrayList are private to the class and can be called only from within the code of the two methods.

The Globe sample project contains the NodeSerializer custom class. This class contains the code and the declarations discussed in this section, and I will not repeat the code here. To use this class in your code, you must create an instance of the class and call the appropriate method. To persist the TreeView control to a file, use the following statements:

```
Dim NS As New NodeSerializer()
NS.CreateList(GlobeTree.Nodes(0), SaveFileDialog1.FileName)
```

To load a TreeView control previously saved to a file, use the following statements:

```
Dim NS As New NodeSerializer()
NS.LoadNodes(GlobeTree, OpenFileDialog1.FileName)
```

I included these statements in the Globe project, but they're commented out. To test the Globe application's File menu commands, add a few items to the TreeView control (countries and cities) and save the tree to a disk file. Then select the root node and delete it by clicking the Delete Current Node button, and load the file you just saved to disk.

One last remark about the code that loads a TreeView control from a disk file: Because the TreeView is persisted to an XML file, the user might attempt to open an XML file that contains irrelevant data. You must insert a structured exception handler to avoid runtime errors or use a new extension for these files. After looking at the XML files generated by the `Serialize` method for a couple of TreeView controls, you should change the SoapFormatter to a Binary Formatter.

In this section, you learned how to serialize the TreeView control's `Nodes` collection, which holds the control's data. You might wish to persist the appearance of the control as well, by including each node's font, background, and foreground colors, and so on. To serialize each node's attributes, add more fields to the `sNode` structure. To store each node's font along with the text, add a new member to the structure (the `nodeFont` field) and set this field to the current node's `Font` property.

Using XML Serialization

In addition to the Serialization namespace, which contains the SOAP and BinaryFormatter classes for serializing data into SOAP and binary format, the .NET Framework provides another name space for serializing data: the XmlSerialization namespace. As you can guess, the XmlSerialization namespace provides methods for serializing and deserializing objects in XML format. XML serialization differs from the other two serialization forms in that it serializes only public properties and fields; read-only and private properties are not serialized. Therefore, XML serialization doesn't preserve the state of the object being serialized. The output of XML serialization is both human and machine readable and doesn't require that classes be marked with the <`Serializable`> attribute. Moreover, you have control over the schema of the XML document that's produced with the help of attributes.

As you already know, XML is a standard protocol for transferring data between computers and operating systems. The result of XML serialization is an XML document, which can be used outside the context of the specific application or even the Framework itself. In other words, you don't need the classes that describe the objects being serialized to use the XML document.

To use XML serialization, you must create an instance of the XmlSerializer class and then call its `Serialize` method (or the `Deserialize` method to extract data from an XML stream and populate an instance of a custom class). There's a major difference, however. The XmlSerializer class must be told in its constructor the type of object it's going to serialize. The constructor of the XmlSerializer class requires an argument, which is the type of objects it will serialize or deserialize. Here's how to set up a new instance of the XmlSerializer class:

```
Imports System.Xml.Serialization
Dim serializer As New XmlSerializer(CO.GetType)
Dim FS As FileStream
FS = New FileStream(path, FileMode.Create)
serializer.Serialize(FS, CO)
FS.Close()
```

The first statement imports the System.Xml.Serialization namespace so that we won't have to fully qualify our references to the members of this class. The `CO` variable is an instance of the custom class, whose instances we intend to serialize through the XmlSerializer class. You can also pass the name of the class itself to the constructor by using a statement such as the following:

```
Dim serializer As New XmlSerializer(GetType(CustomClass))
```

The `serializer` object can be used to serialize only instances of the specific class, and it will throw an exception if you attempt to deserialize a different class with it. Note also that all classes are XML-serializable by default, and you don't have to prefix them with the <`Serializable`> attribute.

XmlSerializer can't serialize arbitrary objects. You must tell the XmlSerializer class the type of object it will serialize. In the background, CLR will create a temporary assembly, a process that will take a few moments. The temporary assembly, however, will remain in memory as long as the application is running. After the initial delay, XML serialization will be quite fast.

Serializing and Deserializing Individual Objects

Serializing a single object in XML format is as simple as the other types of serialization. However, you can't serialize multiple objects into the same file because of the specifications of an XML document. An XML document has a single root element, so all the objects must have a common root. In other words, you can't serialize two Rectangle objects one after the other, as you did in the previous section with the SOAP and binary formatters. However, you can create an array with as many objects as you like and serialize it in XML format. The array is a single object, which can be serialized. Because XML is a universal standard, you can't use it to serialize data structures that are specific to the Framework. You can't serialize ArrayLists, HashTables, and other collections to XML format.

How about the objects you can serialize? The XmlSerializer class serializes the public properties of objects and doesn't care about the source of the data. When you serialize a Rectangle object, the following data is created:

```
<?xml version="1.0" encoding="utf-8"?>
<Rectangle xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Location>
    <X>10</X>
    <Y>10</Y>
  </Location>
  <Size>
    <Width>100</Width>
    <Height>160</Height>
  </Size>
  <X>10</X>
  <Y>10</Y>
  <Width>100</Width>
  <Height>160</Height>
</Rectangle>
```

This is a well-formed XML document. XML is discussed in more detail later in this chapter, but for the purposes of XML serialization, you don't have to know anything about XML. Just keep in mind that every item in this document is delimited by a pair of tags in angle brackets. The tags are named after the properties of the object, and you need not be a rocket scientist to understand that this document represents a Rectangle object. As you can see, the XmlSerializer extracted the basic properties of the Rectangle object. There's some redundancy in this file because the values of the properties appear twice. This isn't part of the XML specification; the document contains the values of the following properties: `Location`, `Size`, `X`, `Y`, `Width`, and `Height`. The Rectangle object exposes additional properties, such as the `Top`, `Bottom`, and so on, but these values aren't serialized. The `Location` property is an object, which in turn exposes the `X` and `Y` properties. The values of these properties appear within the `Location` segment of the document, as well as separate values near the end of the file. The same happens with the `Size` property.

To deserialize the data and create a new Rectangle object with the same properties as the original one, set up a Stream object and an XmlSerializer object, and then call the XmlSerializer object's `Deserialize` method:

```
Imports System.Xml.Serialization
Dim serializer As New XmlSerializer(Rectangle.GetType)
Dim FS As FileStream
FS = New FileStream(path, FileMode.Open)
Dim R As Rectangle
R = serializer.Deserialize(FS)
FS.Close()
```

Serializing Custom Objects

Listing 16.13 shows a class that describes books. The Book class is quite trivial, except that each book can have any number of authors. The authors are stored in an array of Book.Author objects.

LISTING 16.13: Book Class

```
Public Class Book
    Private _title As String
    Private _pages As Integer
    Private _price As Decimal
    Private _authors() As Author

    Public Sub New()

    End Sub

    Public Property Title() As String
        Get
            Return _title
        End Get
        Set(ByVal Value As String)
            If Value.Length > 100 Then
                _title = Value.Substring(0, 99)
            Else
                _title = Value
            End If
        End Set
    End Property

    Public Property Pages() As Integer
        Get
            Return _pages
        End Get
        Set(ByVal Value As Integer)
            _pages = Value
        End Set
    End Property

    Public Property Price() As Decimal
        Get
            Return _price
        End Get
        Set(ByVal Value As Decimal)
            _price = Value
        End Set
    End Property

    Public Property Authors() As Author()
        Get
            Return (_authors)
        End Get
        Set(ByVal Value As Author())
```

```
                _authors = Value
            End Set
        End Property

        Public Class Author
            Private _firstname As String
            Private _lastname As String

            Public Property FirstName() As String
                Get
                    Return _firstname
                End Get
                Set(ByVal Value As String)
                    If Value.Length > 50 Then
                        _firstname = Value.Substring(0, 49)
                    Else
                        _firstname = Value
                    End If
                End Set
            End Property

            Public Property LastName() As String
                Get
                    Return _lastname
                End Get
                Set(ByVal Value As String)
                    If Value.Length > 50 Then
                        _lastname = Value.Substring(0, 49)
                    Else
                        _lastname = Value
                    End If
                End Set
            End Property
        End Class

    End Class
```

The following statements create a new Book object, which includes three authors:

```
Private BK0 As New Book
Dim authors(2) As Book.Author
authors(0) = New Book.Author
authors(0).FirstName = "Author1 First"
authors(0).LastName = "Author1 Last"
authors(1) = New Book.Author
authors(1).FirstName = "Author2 First"
authors(1).LastName = "Author2 Last"
authors(2) = New Book.Author
authors(2).FirstName = "Author3 First"
authors(2).LastName = "Author3 Last"
BK0.Title = "Book Title"
BK0.Pages = 234
BK0.Price = 29.95
BK0.Authors = authors
```

Let's see how to serialize arrays of objects in XML format, starting with a hard rule: The array to be serialized must be typed. All the elements of the array should have the same type, which must match the type you pass to the constructor of the XmlSerializer class. You can't create an array of objects and store objects of different types to it. No warning will be issued at design time, but a runtime exception will be thrown as soon as your code reaches the `Serialize` method. The XmlSerializer is constructed for a specific type of object, and any given instance of this class can handle objects of the specific type and nothing else.

To serialize the Book objects created with the preceding statements, we'll first create an array of the Book type and store a few properly initialized instances of the Book class to its elements. Then we'll pass this array to the `Serialize` method of the XmlSerializer class, as shown in Listing 16.14.

Listing 16.14: XML Serialization of an Array of Objects

```
Private Sub bttnSaveArrayXML_Click(...)Handles bttnSaveArrayXML.Click
    Me.Cursor = Cursors.WaitCursor
    Dim AllBooks(3) As Book
    AllBooks(0) = BK0
    AllBooks(1) = BK1
    AllBooks(2) = BK2
    AllBooks(3) = BK3

    Dim serializer As New XmlSerializer(AllBooks.GetType)

    Dim FS As FileStream
    Try
        FS = New FileStream("..\SerializedXMLArray.xml", _
                        FileMode.Create)
        serializer.Serialize(FS, AllBooks)
    Catch exc As Exception
        MsgBox(exc.InnerException.ToString)
        Exit Sub
    Finally
        FS.Close()
    End Try
    Me.Cursor = Cursors.Default
    bttnLoadArrayXML.Enabled = True
    TextBox1.Clear()
    TextBox1.Text = _
        "Array of Book objects saved in file SerializedXMLArray.xml"
End Sub
```

The XmlSerializer class's constructor accepts as an argument the array type. Because the array is typed, it can figure out the type of custom objects it will serialize.

There's a substantial overhead the first time you create an instance of the XmlSerializer class, but the process isn't repeated during the course of the application. There is overhead because the CLR creates a temporary assembly for serializing and deserializing the specific type. This assembly, however, remains in memory for the course of the application, and the initial overhead won't recur. This means that although there will be an additional delay of a couple of seconds when the application starts (or whenever you load the settings), you can persist the class with the application's configuration every time the user changes one of the settings without any performance penalty.

Serializing ArrayLists and HashTables

Although ArrayLists and HashTables aren't serializable in XML format, there will be occasions when you want to serialize data stored in collections of these two types (and perhaps other types of nonserializable collections). I singled out ArrayLists and HashTables because they're the most common collections in Windows programming. Because the only collection that XML serialization classes can serialize is the array, it's possible to serialize any collection by converting it to an array. It's trivial to export the elements of an ArrayList to an array and then serialize the array. Of course, the array should contain elements of the same type, which must appear in the array's declaration.

Let's consider an ArrayList of Book objects: the *BooksList* collection. You can move the elements of the ArrayList to an array of Book objects with a few statements like the following:

```
Dim BooksArray(BooksList.Count - 1) As Book
Dim BK As Book, i As Integer
For Each BK In BooksList
    BooksArray(i) = BK
    i += 1
Next
```

Having populated the *BooksArray* array with the collection of Book objects, you can serialize them into XML with the techniques discussed already. You can also export the ArrayList's data to an array by using the method ToArray. The ArrayList's elements will be exported to an array of the appropriate type. The reverse process will be used to deserialize the collection: First, you deserialize the data into an array, and then you move the array's elements into the ArrayList:

```
'' Statements to deserialize the BooksArray array
Dim i As Integer
For i = 0 To BooksArray.GetLength(0) - 1
    BooksList.Add BooksArray(i)
Next
```

Serializing HashTables might take a few more statements because it involves keys, not just data. Usually, each element's key is one of the custom object's properties. For Book objects, the most likely candidate for a key is the book's ISBN. In this case, you just export all the objects from the HashTable into an ArrayList and proceed as explained already. If the collection's key, however, isn't a property of the custom object, you must create a new class that has the exact same structure as the Book class and an additional property for the key.

```
Public Class KeyBook
   Public Book As Book
   Public Key As String
End Class
```

Couldn't I take advantage of inheritance and derive the KeyBook class from the Book class? The answer is no, because the Book class can't be cast into a derived class. In other words, you can cast KeyBook objects to Book objects but not the opposite. (Remember widening versus narrowing conversions from Chapter 3, ''Programming Fundamentals''?) That's why I had to create a new class with the custom type and a field for the key. (I'm not using a property procedure for the Key property because no application will ever access this member, except for the code that serializes/deserializes the collection.) The following statements extract the Book objects from the HashTable, convert them to KeyBook objects, and store them into the BooksArray array:

```
Dim BooksArray(HT.Count - 1) As KeyBook
Dim bkey As Integer, i As Integer = 0
Dim KeyBook As New KeyBook
For Each bkey In HT.Keys
    KeyBook.Book = HT(bkey)
    KeyBook.Key = bkey
    BooksArray(i) = KeyBook
    i += 1
Next
```

Serializing Generic Collections

As you'll recall from Chapter 14, there are several collections that belong to the Collections.Generic namespace, and they're typed collections. The drawback of these collections is that they can be used to store objects of the same type — but in most applications this is an advantage, not a drawback. Typical collections contain objects of the same type, and you should use generic collections whenever possible.

Unlike the general collections, such as ArrayLists and HashTables, the generic collections are typed and can be serialized, because the compiler knows the type of objects stored in them. The List collection, for example, is the typed equivalent of the ArrayList collection. Although the ArrayList collection can't be serialized with the XmlSerializer class, the List collection can be serialized with the same class. When you create a new instance of the XmlSerializer class, you pass the type of the object you intend to serialize as an argument:

```
Dim Persons As New System.Collections.Generic.List(Of Person)
Dim XMLSRLZR As New XmlSerializer(Persons.GetType)
```

The compiler knows that the collection will be used to store objects of the Person type, and it can generate the appropriate XML structure. The statements for serializing and deserializing the collection are identical to the ones you'd use to serialize any other object:

```
' To serialize a typed collection:
Dim strmWriter As New IO.StreamWriter(file_path)
XMLSRLZR.Serialize(strmWriter, Persons)
' To deserialize a typed collection:
Dim Rstrm As New IO.FileStream(file_path, IO.FileMode.Open)
newPersons = CType(XMLSRLZR.Deserialize(Rstrm), _
             System.Collections.Generic.List(Of Person))
```

Working with XML Files

Two advantages of the XML format are that humans can read it and there are many tools for manipulating it. To pave the way to the following chapter, let's examine the structure of an XML document and look briefly at the tools for editing XML documents.

Understanding XML Structure

Switch to the XMLBooks project and add the XML file with the serialized books to the project or to a new Windows project. Right-click the name of the project and from the context menu choose Add Existing Item to open the Add Existing Item dialog box. Select Data Files in the File Type combo box and move up to the project's main folder. Select the `SerializedXMLArray.xml` file and add it to the project by selecting the Add option from the Add drop-down list on the dialog box.

Then double-click the newly added file to open it in the editor, as shown in Figure 16.4. The XML editor will figure out the structure of the document and indent the file's contents. XML files are made up of tags, which are embedded in angle brackets. Each tag has a name, and the tags go in pairs: the opening tag and the closing tag. Everything between an opening and closing tag is an *element*. As you can see, elements may contain other elements, to any depth. In fact, the entire document (with the exception of its header) is enclosed in a pair of <`ArrayOfBook`> and </`ArrayOfBook`> tags. This is the document's root element, and a valid XML file should have a single root element that encloses all other elements. Moreover, the remaining elements must nest properly; each element must be closed under the parent element in which it was opened, just like nesting flow control structures in VB.

The first line in the file is the document's prologue, and it identifies an XML file. The `version` keyword identifies the version of the XML specification. Note that the prologue tag has no closing tag.

The document's body is embedded in the `ArrayOfBook` element. This is the document's root element, and each XML document must have one, and only one, root element.

Each entity is mapped to an element, and each element may have zero or more subelements. An XML file with customers should probably have a <`Customers`> root element, which in turn should contain any number of <`Customer`> elements. Each <`Customer`> element is made up of more-specific elements, such as <`Name`>, <`Address`>, <`Email`>, and so on. The names of the tags make sense to us humans, but to the computer, they're just names. Computers can't manipulate an XML document based on the names of the elements; they can understand only the structure of the document. For example, you can request the total number of elements in the document, the child elements of a specific element, the `Email` element of a specific `Customer` element, and so on.

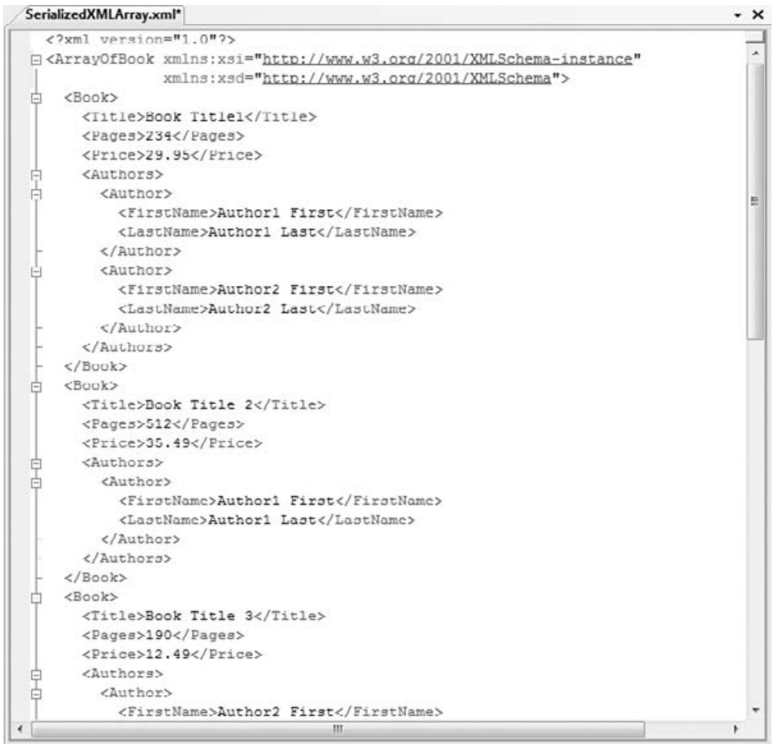


FIGURE 16.4
The XML Editor of Visual Studio

```
<?xml version="1.0"?>
<ArrayOfBook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Book>
    <Title>Book Title1</Title>
    <Pages>234</Pages>
    <Price>29.95</Price>
    <Authors>
      <Author>
        <FirstName>Author1 First</FirstName>
        <LastName>Author1 Last</LastName>
      </Author>
      <Author>
        <FirstName>Author2 First</FirstName>
        <LastName>Author2 Last</LastName>
      </Author>
    </Authors>
  </Book>
  <Book>
    <Title>Book Title 2</Title>
    <Pages>512</Pages>
    <Price>35.49</Price>
    <Authors>
      <Author>
        <FirstName>Author1 First</FirstName>
        <LastName>Author1 Last</LastName>
      </Author>
    </Authors>
  </Book>
  <Book>
    <Title>Book Title 3</Title>
    <Pages>190</Pages>
    <Price>12.49</Price>
    <Authors>
      <Author>
        <FirstName>Author2 First</FirstName>
```

Each element may have one or more attributes. Instead of specifying all the items as elements, we can specify them as *attributes* of the element to which they belong. Instead of nesting the `LastName` and `FirstName` elements under the `Author` element, you can specify the `LastName` and `FirstName` fields as attributes:

```
<Author FirstName="author First" LastName="author Last"/>
```

Using attributes in place of elements is just a way to organize the information in an XML document. Using attributes doesn't change the nature of the document, just its structure.

You can actually determine how the XmlSerializer class will create the XML file with the proper attributes. Say you modify the declaration of the Authors class in the Books sample project by prefixing the `FirstName` and `LastName` properties with the `<XmlAttribute>` attribute, as shown here:

```
<Serializable()> Public Class Author
<XmlAttribute(AttributeName:="FirstName")> _
        Public Property FirstName() As String
    Get
        Return _firstname
    End Get
    Set(ByVal Value As String)
        If Value.Length > 50 Then
```

```
                _firstname = Value.Substring(0, 49)
            Else
                _firstname = Value
            End If
        End Set
    End Property
    <XmlAttribute(AttributeName:="LastName")> _
            Property LastName() As String
        Get
            Return _lastname
        End Get
        Set(ByVal Value As String)
            If Value.Length > 50 Then
                _lastname = Value.Substring(0, 49)
            Else
                _lastname = Value
            End If
        End Set
    End Property
```

The XmlSerializer will insert the author's first and last names as attributes of the `<Author>` tag. Here's an `<Author>` tag generated by the XmlSerializer with the revised class definition:

```
<Authors>
  <Author FirstName="Author1 First" LastName="Author1 Last" />
  <Author FirstName="Author2 First" LastName="Author2 Last" />
</Authors>
```

There are other attributes for customizing the output of the XmlSerializer, but a discussion of these attributes is beyond the scope of this book.

Editing XML Files

You can easily edit this file. Apart from changing the values of existing elements, such as a book's price or page count, you can insert new elements.

Let's add a new author to the second book. Create a new line before the closing tag `</Authors>` of the second book, and insert the following tag: **`Author`**. As soon as you type the closing bracket, the editor inserts the matching closing tag, and leaves the pointer between the two tags, so you can enter the value of the newly inserted element. The `Author` element is not a simple element; it contains two nested elements, `FirstName` and `LastName`. Enter the tag **`FirstName`**, and the editor inserts the matching closing tag. Type the author's first name between the two tags. Then move after the closing `</Author>` tag and press Enter. In the new line, enter another author.

If you experiment a little with the XML Editor, you'll realize that this isn't a straight text editor. It reacts to your actions and makes sure that the XML document is constantly a valid document.

If you deserialize the edited file, you'll see that the `AllBooks` array will be populated with the data of the edited XML file: One of the books will have an additional author. It's fairly straightforward to edit existing XML files or to create new ones with the XML Editor that comes with Visual Studio. The XML Editor makes sure that the file you're editing follows the rules of a valid XML file — that is, the file's tags open and close within the element they belong to, and there are no spaces or invalid characters in the tag names. If you make a mistake, the editor will instantly underline the offending element. If you hover the pointer over the element in error, you'll see a description of the error.

The XML Editor can do more than help you generate valid XML files. With the `SerializedXMLArray.xml` file open in the editor, choose the Create Schema command from the XML menu. A new file is generated, the `SerializedXMLArray.xsd` file. *XSD* stands for *XML Schema Definition*, and the XSD files describe the schema (structure) of a class of XML files. The schema of the XML file with the books is another XML document, made up of nested tags. These tags, however, describe the names and structure of the elements that may appear in the document and their type. The following statements specify the data types of three elements:

```
<xs:element name="Title" type="xs:string" />
<xs:element name="Pages" type="xs:unsignedShort" />
<xs:element name="Price" type="xs:decimal" />
```

`Authors` is a complex element that may contain one or more `Author` elements, and each `Author` tag has two attributes: the `FirstName` and `LastName` attributes.

```
<xs:element name="Authors">
   <xs:complexType>
      <xs:sequence>
         <xs:element maxOccurs="unbounded" name="Author">
           <xs:complexType>
             <xs:attribute name="FirstName"
                 type="xs:string" use="required" />
             <xs:attribute name="LastName"

                 type="xs:string" use="required" />
            </xs:complexType>
         </xs:element>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

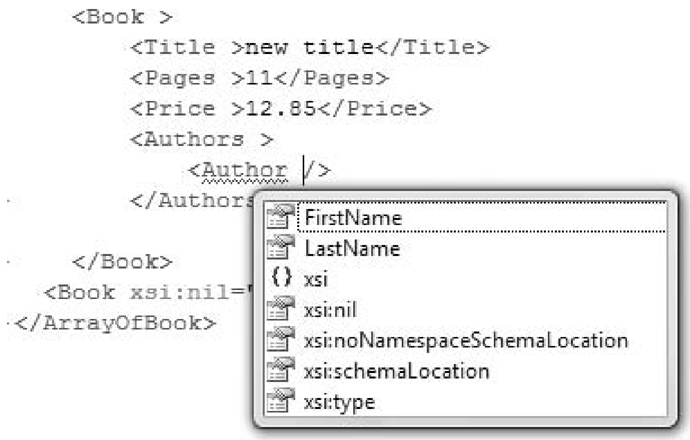
It's fairly easy to understand the structure of both XML and XSD files, but it's not nearly as easy to create an XSD file from scratch. The XML Editor, however, can apply a schema file to an XML document and make sure that the file you're generating conforms to its schema. And this is how schemas are used: The editor reads the schema and makes sure that the XML document you're reading complies with the specified schema.

Enter a new `<Book>` tag in the file and you will see in IntelliSense the valid tags that may appear within this element every time you type the opening bracket. If you enter the `<Author>` tag, which must appear under an `<Authors>` tag, you will see in the IntelliSense box the two attributes that may appear in the `<Author>` tag: the `LastName` and `FirstName` attributes (see Figure 16.5).

If you omit the `Authors` element, the editor will underline the closing `</Book>` tag. This indicates that the `<Book>` element may be invalid. Hover the pointer over the tag in error and you'll see the error description in a ToolTip box. The error message indicates that the cause of the problem could be a missing `<Authors>` tag. If you examine the schema of the XML file, you'll see that some of the elements contain a `MinOccurs` and a `MaxOccurs` attribute. The <`Authors`> tag doesn't contain any of these attributes, and the editor thinks that each `Book` element should contain at least one `Authors` subelement. This happened because the schema was generated from a sample XML document in which all books had at least one author. Edit the XSD file by inserting the `MinOccurs` attribute in the definition of the `Authors` element:

```
<xs:element name="Authors" minOccurs="0">
```

FIGURE 16.5
The XML Editor can assist you in conforming to a specific schema.

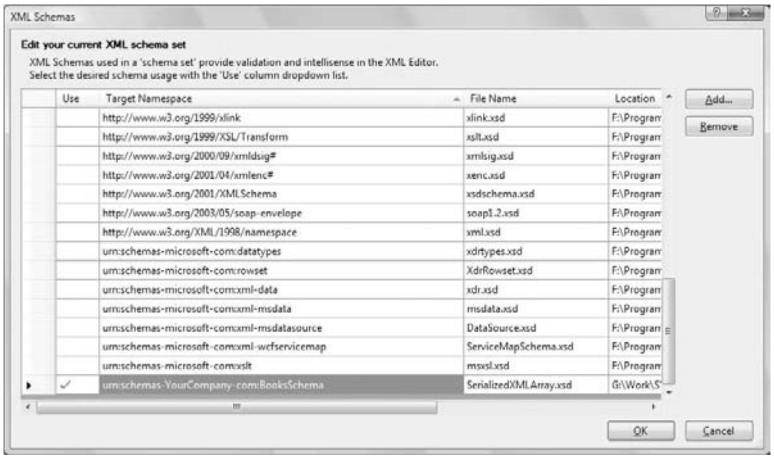


```
    <Book >
        <Title >new title</Title>
        <Pages >11</Pages>
        <Price >12.85</Price>
        <Authors >
            <Author />
        </Authors

    </Book>
  <Book xsi:nil="
</ArrayOfBook>
```

The warning will go away, because the revised schema allows books to have no authors.

The newly generated schema is applied to the open document immediately. You can specify which schema is applied to the document you're editing with the `Schemas` property of the document. Click the ellipses button in the `Schemas` property to see the XML Schemas dialog box, which contains a number of schemata. Click the Add button to open the File Open dialog box and locate the desired schema in your hard drive. To apply a specific schema to the current document, select the Use option in the first column of the XML Schemas dialog box, as shown in Figure 16.6.

FIGURE 16.6
Specifying the schema of the XML document you're editing in Visual Studio



XML Schemas

Edit your current XML schema set

XML Schemas used in a 'schema set' provide validation and intellisense in the XML Editor.
Select the desired schema usage with the 'Use' column dropdown list.

	Use	Target Namespace	File Name	Location
		http://www.w3.org/1999/xlink	xlink.xsd	F:\Progran
		http://www.w3.org/1999/XSL/Transform	xslt.xsd	F:\Progran
		http://www.w3.org/2000/09/xmldsig#	xmlsig.xsd	F:\Progran
		http://www.w3.org/2001/04/xmlenc#	xenc.xsd	F:\Progran
		http://www.w3.org/2001/XMLSchema	xsdschema.xsd	F:\Progran
		http://www.w3.org/2003/05/soap-envelope	soap1.2.xsd	F:\Progran
		http://www.w3.org/XML/1998/namespace	xml.xsd	F:\Progran
		urn:schemas-microsoft-com:datatypes	xdrtypes.xsd	F:\Progran
		urn:schemas-microsoft-com:rowset	XdrRowset.xsd	F:\Progran
		urn:schemas-microsoft-com:xml-data	xdr.xsd	F:\Progran
		urn:schemas-microsoft-com:xml-msdata	msdata.xsd	F:\Progran
		urn:schemas-microsoft-com:xml-msdatasource	DataSource.xsd	F:\Progran
		urn:schemas-microsoft-com:xml-wcfservicemap	ServiceMapSchema.xsd	F:\Progran
		urn:schemas-microsoft-com:xslt	msxsl.xsd	F:\Progran
▶	✓	urn:schemas-YourCompany-com:BooksSchema	SerializedXMLArray.xsd	G:\Work\S

Initially, the Target Namespace column for the schema created by Visual Studio will be empty. This column's setting won't affect your document in any way, but it does help you select the proper schema when you need it. To specify a value for the target namespace, insert a `targetNamespace` attribute in the `<xsd:schema>` tag at the beginning of the file:

```
targetNamespace="urn:schemas-YourCompany-com:BooksSchema"
```

You may be thinking, ''Why bother with editing XML files?'' The reason for this short introduction to XML will become obvious in the following chapter, where you'll learn how to create XML documents programmatically and how to process XML files in your application.

The Bottom Line

Serialize objects and collections into byte streams. Serialization is the process of converting an object into a stream of bytes. This process (affectionately known as dehydration) generates a stream of bytes or characters, which can be stored or transported. To serialize an object, you can use the BinaryFormatter or SoapFormatter class. You can also use the XmlSerializer class to convert objects into XML documents. All three classes expose a Serialize class that accepts as arguments the object to be serialized and a stream object, and writes the serialized version of the object to the specified stream.

Master It Describe the process of serializing an object with a binary or SOAP formatter.

Deserialize streams to reconstruct the original objects. The opposite of serialization is called deserialization. To reconstruct the original object, you use the `Deserialize` method of the same class you used to serialize the object.

Master It Describe the process of serializing an object with a binary or SOAP formatter.

Create XML files in your code. XML is a standard for storing data. In addition to the data, an XML document also describes the structure of its contents by using elements and attributes. Elements represent entities and their properties. Attributes represent the properties of the elements to which they're applied.

Master It How would you create an XML document to describe structured data?

Chapter 17

Querying Collections and XML with LINQ

In Chapter 14, ''Storing Data in Collections,'' you learned how to create collections, from simple arrays to specialized collections such as HashTables and Lists, and how to iterate through them with loops to locate items. Typical collections contain objects, and you already know how to create and manipulate custom objects. In Chapter 16, ''Serialization and XML,'' you learned how to serialize these collections into XML documents, as well as how to create XML documents from scratch. And in Chapter 2, ''Basic Concepts of Relational Databases,'' you'll learn how to manipulate large amounts of data stored in databases.

Each data source provides its own technique for searching and manipulating individual items. What's common in all data sources is the operations we perform with the data: We want to be able to query the data and select the values we're interested in. It's therefore reasonable to assume a common query language for all data sources. This common query language was introduced with version 3 of the Framework and is now part of all .NET languages. It's the LINQ component.

LINQ stands for *Language Integrated Query*, a small language for querying data sources. For all practical purposes, it's an extension to Visual Basic. However, LINQ has a peculiar syntax. More specifically, LINQ consists of statements that you can embed into a program to select items from a collection based on various criteria. Unlike a loop that examines each object's properties and either selects or rejects it, LINQ is a declarative language: It allows you to specify the criteria, instead of specifying how to select the objects. A declarative language, as opposed to a procedural language, specifies the operation you want to perform, and not the steps to take. VB is a procedural language; the language of SQL Server, T-SQL, is a declarative language.

In this chapter, you'll learn how to do the following:

- Perform simple LINQ queries
- Create and process XML files with LINQ to XML
- Process relational data with LINQ to SQL

What Is LINQ?

Although defining LINQ is tricky, a simple example will demonstrate the structure of LINQ and its role in an application. Let's consider an array of integers:

```
Dim data() As Int16 = {3, 2, 5, 4, 6, 4, 12, 43, 45, 42, 65}
```

To select specific elements of this array, you'd write a `For...Next` loop, examine each element of the array, and either select it by storing it into a new array or reject it. To select the elements that are numerically smaller than 10, you'd write a loop like the following:

```
Dim smallNumbers(data.Length-1) As Integer
Dim itm As Integer = 0
For i As Integer = 0 To data.Length
    If data(i) < 10 Then
        smallNumbers(itm) = data(i)
        itm += 1
    End If
Next
ReDim smallNumbers(itm)
```

Just the statements for indexing the *`smallNumbers`* array add a degree of complexity to the code. It would be simpler to store the selected elements into an ArrayList by using a loop like the following:

```
Dim smallNumbers As New ArrayList
Dim itm As Integer
For Each itm In data
    If itm < 10 Then
        smallNumbers.Add(itm)
    End If
Next
```

Let's do the same with LINQ:

```
Dim smallNumbers = From n In data _
                    Where n < 10 _
                    Select n
```

This is a peculiar statement indeed, unless you're familiar with SQL, in which case you can easily spot the similarities. LINQ, however, is not based on SQL, and not every operation has an equivalent in both. Both SQL and LINQ, however, are declarative languages that have many similarities. If you're familiar with SQL, you have already spotted the similarities and the fact that LINQ rearranges the basic elements. The equivalent SQL statement would be something like the following:

```
SELECT *
FROM data
WHERE data.n < 10
```

(You can't process arrays or other data structures with SQL; this example assumes the existence of a database with a table called `data`, and that this table contains a column named `n`.) You'd use the exact same LINQ query to select items from an ArrayList, and a similar statement to select elements from an XML document.

Let's start with the structure where the selected elements will be stored, which is the result of the query. The *smallNumbers* variable is declared without a type, because its type is determined by the type of the collection where the data will come from. We select elements from the *data* array, so *smallNumbers* is an array of integers. Actually, it's not exactly an array of integers; it's a typed collection of integers that implements the `IEnumerable` interface. The LINQ query starts with the `From` keyword, which is followed by a variable that represents the current item in the collection, followed by the `In` keyword and the name of the collection. The first part of the query specifies the collection we're going to query. As with the result of the query, the variable need not be declared; it has the same type as the elements of the collection.

Then comes the `Where` keyword that limits the selection. The `Where` keyword is followed by an expression that involves the variable of the `From` clause; the expression limits our selection. In this extremely trivial example, we select the elements that are less than 10. The last keyword in the expression, the `Select` keyword, determines what we're selecting. In most cases, we select the same value we specified after the `From` keyword, but not always. Here's a variation of the previous query expression:

```
Dim = From n In data _
       Where m mod 2 = 0
       Select "Number " & n.ToString & " is even"
```

Here we select even numbers from the original array and then form a string for each of the selected values. The `Where` part of the statement is an expression, which evaluates to a True/False value and determines whether the current element will be included in the result of the query. As you will see shortly, the criteria can get quite complicated, but the idea is to express a filtering expression that limits our selection.

But why bother with a new component to select values from an array? A `For...Each` loop that processes each item in the collection is not really complicated and is quite efficient. For the time being, LINQ is actually less efficient than the equivalent loop. The promise of LINQ isn't efficiency (not yet, at least), but its potential for becoming a universal querying language. LINQ isn't limited to arrays: It applies to collections, XML files, objects, even relational data, and it provides a uniform querying language regardless of the data source. LINQ is an extension to the .NET Framework that allows developers to query any data source that implements the `IEnumerable` or `IQueryable` interfaces. Collections, XML files, and DataSets implement these interfaces and can be queried with LINQ. The DataSet is a structure for storing data you retrieve from a database at the client, and it's discussed in detail in Chapters 22 and 23.

LINQ Components

To support such a wide range of data sources, LINQ is comprised by multiple components, which are the following:

LINQ to XML This component enables you to search XML documents in many ways. In effect, it replaces XQuery expressions that are used today to select the items of interest in an XML document. Because of LINQ to XML, some new classes that support XML were introduced to Visual Basic, and XML has become a basic data type of the language. The following statement declares an XML variable, and it's quite valid VB code:

```
Dim Employees = <Employees>
                    <Employee ID="1001">
                       <Title>Developer</Title>
```

```
        <Name>John Doe</Name>
    </Employee>
    <Employee ID="1002">
        <Title>Manager</Title>
        <Name>Joe Doe</Name>
    </Employee>
</Employees>
```

If you enter this statement in a VB project and hover the pointer over the *Employees* variable, you'll see that its type is XElement. This type belongs to the System.Xlinq namespace and is new to VB 2008 and treated by VB as a new type. Notice that there are no line-continuation symbols in the XML segment, because line breaks are of no consequence to XML documents. Moreover, as you enter XML statements in the editor, the XML Editor's facilities are activated.

LINQ to Objects This component enables you to search collections of built-in or custom objects. If you have a collection of Color objects, for example, you can select the colors with an intensity of 0.5 or more via the following expression:

```
Dim colors() As Color = {Color.White, _
         Color.LightYellow, Color.Cornsilk, _
         Color.Linen, Color.Blue, Color.Violet}
Dim brightColors = From c In colors _
                  Where c.GetBrightness > 0.5
```

Likewise, you can select the rectangles with a minimum or maximum area by using a query like the following:

```
Dim rects() As Rectangle = _
            {New Rectangle(0, 0, 100, 120), _
             New Rectangle(10, 10, 6, 8)}
Dim query = From R In rects _
           Where R.Width * R.Height > 100
```

LINQ to SQL This component enables you to query relational data by using LINQ rather than SQL. You will find examples of LINQ to SQL samples later in this chapter.

LINQ to DataSet This component is similar to LINQ to SQL, in the sense that they both query relational data. The LINQ to DataSet component allows you query data that have already been stored in a DataSet at the client. DataSets are discussed in detail later in this book, but I won't discuss the LINQ to DataSet component, because the DataSet is an extremely rich object and quite functional on its own.

LINQ to Entities This is similar to the LINQ to Objects component, only the objects are based on relational data. Entities are not discussed in this book.

LINQ to Objects

This section focuses on querying collections of objects. As you can guess, the most interesting application of LINQ to Objects is to select items from a collection of custom objects. Let's create a custom class to represent products:

```
Public Class Product
Private _productID As String
Private _productName As String
Private _productPrice As Decimal
Private _productExpDate As Date

Public Property ProductID() As String
End Property

Public Property ProductName() As String
End Property

Public Property ProductPrice() As Decimal
End Property

Public Property ProductExpDate() As Date
End Property
```

I'm not showing the implementation of various properties, because they're quite trivial (nothing more than the default setters and getters). The *Products* collection is a List object that contains several instances of the class, and it's populated with statements like the following:

```
Dim Products As New System.Collections.Generic.List(Of Product)
Dim P As Product
P = New Product
P.ProductID = "10A-Y"
P.ProductName = "Product 1"
P.ProductPrice = 21.45
P.ProductExpDate = #8/1/2009#
Products.add(P)
```

Now we can use LINQ to query our collection of products based on any property (or combination of properties) of its items. To find out the products that cost more than $20 and have expired already, we can formulate the following query:

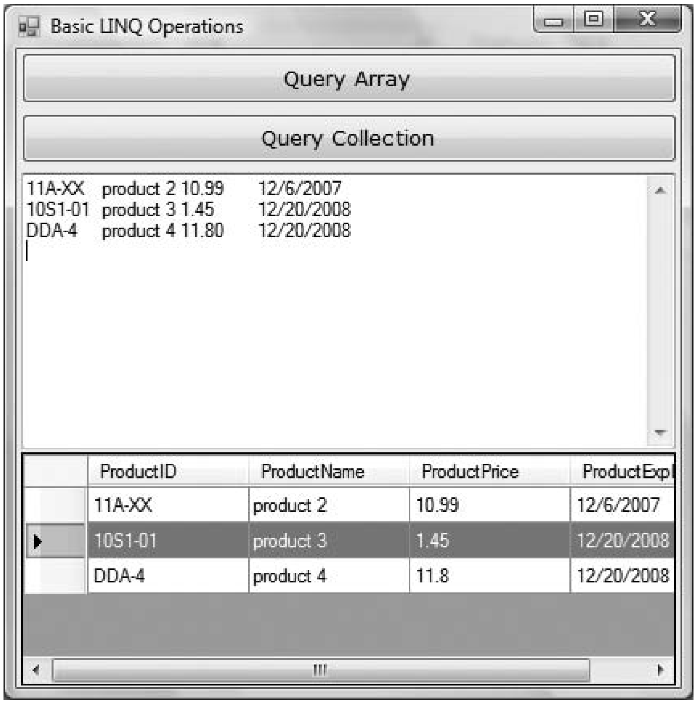
```
Dim query = From prod In products _
            Where prod.ProductPrice < 20 _
                And Year(prod.ProductExpDate) < 2008 _
            Select prod
```

The result of the query is also a List collection, and it contains the products that meet the specified criteria. To iterate through the selected items and display them in a TextBox control, we use a `For...Each` loop, as shown next:

```
For Each P In query
    TextBox1.AppendText(P.ProductID & vbTab & _
        P.ProductName & vbTab & _
        P.ProductPrice.ToString("##0.00") & vbTab & _
        P.ProductExpDate.ToShortDateString & vbCrLf)
Next
```

Okay, if we have to write the loop, why not examine each item in the loop's body and not bother with an embedded query? For starters, the same query will work with other data sources. You can replace the collection with an XML document, and the same query will work. Moreover, displaying the data manually is not your only option. In Chapter 23, ''Building Data-Bound Applications,'' you'll learn about DataGridView, which is a data-bound control. You set the control's `DataSource` property to a collection of data, and the control displays all the data in its data source in a tabular format. The items of the collection are mapped to rows of the control, and the properties of the objects stored in the collection are mapped to columns, as shown in the grid of Figure 17.1. This is the `frmLINQBasics` form of the VBLINQ project. In Chapter 23, you'll learn how to customize the appearance of the DataGridView control as well.

FIGURE 17.1
Querying a collection of custom objects



```
11A-XX   product 2 10.99    12/6/2007
10S1-01  product 3 1.45     12/20/2008
DDA-4    product 4 11.80    12/20/2008
```

ProductID	ProductName	ProductPrice	ProductExp
11A-XX	product 2	10.99	12/6/2007
10S1-01	product 3	1.45	12/20/2008
DDA-4	product 4	11.8	12/20/2008

The DataGridView control not only is a highly customizable control for browsing sets of data, but also allows the editing of its contents. You can edit the selected items on the control and then access them from within your code through the control's `DataSource` property. First, you must cast the control's `DataSource` property to a BindingSource object and then access the rows of the control. Each row must be cast in turn into the Product type. The following loop displays the `ProductName` field of the first row on the control:

```
Dim prods = CType(DataGridView1.DataSource, BindingSource)
Debug.WriteLine(CType(prods(0), Product).ProductName)
```

The DataGridView control is a flexible tool for browsing and editing sets of data. You'll see in Chapter 23 how to bind the DataGridView control to data; in this example, I wanted to show only that you can bind it to any collection.

Another component of a LINQ expression is the `Order By` clause, which determines how the objects will be ordered in the output list. To sort the output of the preceding example in descending order, append the following `Order By` clause to the expression:

```
Dim query = From prod In products _
            Where prod.ProductPrice < 20 _
                And Year(prod.ProductExpDate) < 2010 _
             Select prod _
             Order By prod.ProductName
```

Querying Collections

As I mentioned already, LINQ can be applied to all objects that implement the `IEnumerable` interface. Many methods of the Framework return their results as a collection that implements the `IEnumerable` interface. As you recall from Chapter 15, ''Accessing Folders and Files,'' the `GetFiles` method of the IO.Directory class retrieves the files of a specific folder and returns them as a collection of strings:

```
Dim files = Directory.GetFiles("C:\")
```

I'm assuming that you have turned on type inference for this project (it's on by default), so I'm not declaring the type of the *`files`* collection. If you hover the pointer over the *`files`* keyword, you'll see that its type is `String()` — an array of strings. This is the `GetFiles` method's return type, so we need not declare the *`files`* variable with the same type. The variable's type is inferred from its value.

The `GetFiles` method returns an array of strings. To find out the properties of each file, you must create a new FileInfo object for each file and then examine the values of the FileInfo object's properties. To create an instance of the FileInfo class that represents a file, you'd use the following statement:

```
Dim FI As New FileInfo(file_name)
```

(As a reminder, the FileInfo class as well as the Directory class belong to the IO namespace. You must either import the namespace into the current project or prefix the class names with the

IO namespace: IO.FileInfo). The value of the *FI* object must now be used in the `Where` clause of the expression to specify a filter for the query:

```
Dim smallFiles = _
             From file In Directory.GetFiles("C:\") _
             Where New FileInfo(file).Length > 10000 _
             Order By file
             Select file
```

The *file* variable is local to the query, and you cannot access it from the rest of the code. You can actually create a new *file* variable in the loop that iterates through the selected files, as shown in the following code segment:

```
For Each file In smallFiles
    Debug.WriteLine(file)
Next
```

The selection part of the query is not limited to the same variable as specified in the `From` clause. To select the name of the qualifying files, instead of their paths, use the following selection clause:

```
Select New FileInfo(file).Name
```

The *smallFiles* object should still be an array of strings, right? Not quite. This time, if you hover the pointer over the name of the *smallFiles* variable, you'll see that its type is `IEnumerable(Of String)`. And it makes sense, because the result of the query is not of the same type as its source. This time we created a new string for each of the selected items, and so *smallFiles* is an `IEnumerable` type. Let's select each file's name and size with the following query:

```
Dim smallFiles = _
         From file In Directory.GetFiles("C:\") _
         Where New FileInfo(file).Length > 10000 _
         Select New FileInfo(file).Name, _
                New FileInfo(file).Length
```

This time, *smallFiles* is of the `IEnumerable(Of <anonymous type>)` type. And what exactly is the anonymous type? It's simply a type with no name. Its structure is known, but there's no name for this type. Because we have selected the two properties of interest in the query itself (the file's name and size), we can display them with the following loop:

```
For Each file In smallFiles
    Debug.WriteLine(file.Name & vbTab & _
                    file.Length.ToString)
Next
```

As soon as you type in the name of the *file* variable and the following period, you will see the Name and Length properties of the anonymous type in the IntelliSense box. As you can see, the editor created a new type behind the scenes for you that exposes the selected values as properties.

The properties of the new type are named after the items you specified in the `Select` clause and they have the same type. Because the type has no name, it's called an *anonymous type*. What this means, practically, is that you can't declare a new variable of the same type, except by assigning a value of this type to the variable. You can also control the names of the properties of the anonymous type with the following syntax:

```
Select New With {.FileName = New FileInfo(file).Name, _
                 .FileSize = New FileInfo(file).Length}
```

This time we select a new object, which is created on-the-fly and has two properties named `FileName` and `FileSize`. The values of the two properties are specified as usual. The new object is still of the anonymous type. To display each selected file's name and size, modify the `For...Each` loop as follows:

```
For Each file In smallFiles
    Debug.WriteLine(file.FileName & vbTab & _
                    file.FileSize.ToString)
Next
```

As you can see, LINQ is not a trivial substitute for a loop that examines the properties of the collection's items; it's a powerful and expressive syntax for querying data in your code, it creates data types on the fly and exposes them in your code.

You can also limit the selection by applying the `Where` method directly to the collection:

```
Dim smallFiles = _
    Directory.GetFiles("C:\").Where (Function(file) _
    (New FileInfo(file).Length > 10000))
```

The functions you specify in certain extended methods are called *lambda functions*, and they're declared either inline, if they're single line functions, or as delegates.

Let me explain how the `Where` clause of the last sample code segment works. The `Where` clause should be followed by an expression that evaluates to a True/False value, the lambda function. First, you specify the signature of a function; in our case, the function accepts a single argument, which is the current item in the collection. Obviously, the `Where` clause will be evaluated for each item in the collection, and for each item, the function will accept a different object as argument. In the following section, you'll see lambda functions that accept two arguments. The name of the argument can be anything; it's a name that you will use in the definition of the function to access the current collection item. Then comes the definition of the function, which is the expression that compares the current file's size to 100,000 bytes. If the size exceeds 100,000 bytes, the function will return True — otherwise, False.

In this example, the lambda function is implemented inline. To implement more-complicated logic, you can write a function and pass the address of this function to the `Where` clause. Let's consider that the function implementing the filtering is the following:

```
Private Function IsLargeTIFFFile(ByVal fileName As String) As Boolean
    Dim file As FileInfo
    file = New FileInfo(fileName)
    If file.Length > 100000 And file.Extension.ToUpper = ".TIF" Then
```

```
            Return True
        Else
            Return False
        End If
    End Function
```

To call this function from within a LINQ expression, use the following syntax:

```
Dim largeImages = _
        Directory.GetFiles("C:\").Where(AddressOf IsLargeTIFFFile)
    MsgBox(smallFiles.Count)
```

Aggregating with LINQ

LINQ allows you to query for aggregates too. By default, it adds a few extended methods for calculating aggregates to all collections. Let's return to our array of integers, the *data* array. To calculate the count of all values, call the `Count` method of the *data* array. The count of elements in the data array is given with the following expression:

```
Dim count = data.Count
```

Of course, the Array class doesn't provide a `Count` method, so where did it come from? `Count`, as well as a number of other methods, is an extended method. An *extended method* is added to a class without having to inherit the original class and create a derived class (as you recall, the array cannot even be inherited). The Framework allows you to add methods to a class by extending it, and this technique is used heavily by LINQ.

In addition to the `Count` method, any LINQ-capable class exposes the `Sum` method, which sums the values of a specific element or attribute in the collection. To calculate the sum of the selected values from the *`data`* array, use the following LINQ expression:

```
Dim sum = From n data _
          Where n > 10
          Select n.Sum
```

You can also calculate arbitrary aggregates by using the `Aggregate` method, which accepts as an argument a lambda expression. This expression, in turn, accepts two arguments: the current value and the aggregate. The implementation of the function calculates the aggregate. Let's consider a lambda expression that calculates the sum of the squares over a sequence of numeric values. The declaration of the function is as follows:

```
Function(aggregate, value)
```

Its implementation is shown here:

```
aggregate + value ˆ 2
```

To calculate the sum of the squares of all items in the data array, use the following LINQ expression:

```
Dim sumSquares = data.Aggregate( _
           Function(sumSquare As Long, n As Integer) _
                    sumSquare + n ˆ 2
```

The single statement that implements the aggregate adds the square of the current element to the *sumSquare* argument. When we're done, the *sumSquare* variable holds the sum of the squares of the array's elements. Aggregates are not limited to numeric values. Here's an interesting example of a LINQ expression that reverses the words in a sentence. The code starts by splitting the sentence into words, which are returned in an array of strings. Then it calls the `Aggregate` method, passing as an argument a lambda expression. This expression is a function that prefixes the aggregate (the string with words in reverse order) with the current word:

```
Dim sentence = _
        "The quick brown fox jumped over the lazy dog"
Dim reverseSentence = _
           sentence.Split(" ".c).Aggregate( _
           Function (newSentence, word) _
           word & " " & newSentence
```

A few more interesting extended methods are the following:

Take (N) Selects the first *n* elements from the collection

TakeWhile (Expression) Keeps selecting elements from the collection while the expression is True. To select values while they're smaller than 10, use the following lambda expression:

```
Function(n)  n  <  10
```

This expression selects values until it finds one that exceeds 10. The selection stops there, regardless of whether some of the following elements drop below 10.

Skip* and *SkipWhile The `Skip` and `SkipWhile` methods are equivalent to the `Take` and `Take-While` methods: They skip a number of items and select the remaining ones.

Distinct The `Distinct` method, finally, returns the distinct values in the collection:

```
Dim uniqueValues = data.Distinct
```

LINQ to XML

In this section, we'll move on to a more interesting component of LINQ, the LINQ to XML component. XML is gaining in popularity and acceptance, and Microsoft has decided to promote XML to a basic data type. Yes, XML is a data type like integers and strings! To understand how far VB is taking XML, type the following in a procedure or event handler:

```
Dim products = <Books>
                 <Book ISBN="0000000000001">
                   <Name>Book Title 1</Name>
                   <Price>11.95</Price>
                  </Book>
                  <Book ISBN="000000000002">
                     <Name>Book Title 2</Name>
                     <Price>10.25</Price>
                   </Book>
                 </Books>
```

You need not worry too much about getting the document exactly right, because the editor works just like the XML Editor. Every time you type an opening tag, it inserts the matching closing tag and ensures that what you're typing is a valid XML document. You can't apply a schema to the XML document you're creating, but you should expect this feature in a future version of Visual Studio.

You can create a new XML document in your code, but what can you do with it? We need a mechanism to manipulate the XML document with simple tools, and these tools are available through the following XML helper objects:

XDocument represents the XML document.

XComment represents a comment in the XML document.

XElement represents an XML element.

XAttribute represents an attribute in an XML element.

These objects can be used to access the document but also to create it. Instead of creating an XML document directly in your code, you can use the XML helper objects and a structural approach to create the same document. A simple XML document consists of elements, which may include attributes. To create a new XElement object, pass the element's name and value to its constructor:

```
New XElement(element_name, element_value)
```

The following statement will create a very simple XML document:

```
Dim XmlDoc = New XElement("Books")
MsgBox(XmlDoc.ToString)
```

You will see the string `<Books />` in a message box. This is a trivial, yet valid, XML document. To create the same book collection as we did earlier by using the helper objects, insert the following statements in a button's `Click` event handler:

```
Dim doc = _
     New XElement("Books", _
         New XElement("Book", _
             New XAttribute("ISBN", "0000000000001"), _
             New XElement("Price", 11.95), _
             New XElement("Name", "Book Title 1"), _
             New XElement("Stock", _
                 New XAttribute("InStock", 12), _
                 New XAttribute("OnOrder", 24))), _
             New XElement("Book", _
                 New XAttribute("ISBN", "0000000000002"), _
             New XElement("Price", 10.25), _
             New XElement("Name", "Book Title 2"), _
             New XElement("Stock", _
                 New XAttribute("InStock", 7), _
                 New XAttribute("OnOrder", 10))))
```

I've added a twist to the new document to demonstrate the use of multiple attributes in the same element. The Stock element contains two attributes, `InStock` and `OnOrder`. Each element's value can be a basic data type, such as a string or a number, or another element. The Price element is a decimal value, and the Name element is a string. The Book element, however, contains three subelements: the Price, Name, and Stock elements.

The *doc* variable is of the XElement type. An XML document is not necessarily based on the XDocument class. The two basic operations you can perform with an XElement (and XDocument) object are to save it to a file and reload an XElement object from a file. The operations are performed with the `Save` and `Load` methods, which accept the file's name as an argument.

Traversing XML Documents

Let's look at how we can process an XML document by using the XML helper objects. If you're familiar with the XML tools for manipulating XML documents from previous versions of Visual Basic, you'll be impressed by the simplicity of the new approach. Each element may have one or more parent elements, which you can access via the `Ancestors` property, and one or more child elements, which you can access via the `Descendants` property. Both methods return a collection of XElement objects. In addition, elements may have attributes, which you can access via the `Attribute` property.

```
For Each book In doc.Elements("Book")
    Debug.WriteLine("ISBN " & book.Attribute("ISBN").Value.ToString)
    Debug.WriteLine("   Title: " & book.Descendants("Name").Value.ToString)
    Debug.WriteLine("   Price: " & book.Descendants("Price").Value.ToString)
    Dim stock = book.Element("Stock")
    Debug.WriteLine("      Books in stock " & stock.Attribute("InStock").Value)
    Debug.WriteLine("      Books on order " & stock.Attribute("OnOrder").Value)
Next
```

The loop iterates through the Book elements in the XML file. The expression `doc.Elements ("Book")` returns a collection of XElement objects, each of which has an `Attribute` property and a `Descendants` property. The `Attribute` property lets you access each attribute of the current element by name. The `Descendants` property returns a collection of XElement objects, one for each subelement of the element represented by the *book* XElement. One of the elements, the `Stock` element, has its own attributes. To read their values, the code creates a variable that represents the

Stock element and uses its Attribute property to retrieve an attribute by name. The output of the preceding code segment is shown here:

```
ISBN 0000000000001
   Title: Book Title 1
   Price: 32.3
      Books in stock 12
      Books on order 24
ISBN 0000000000002
   Title: Book Title 2
   Price: 12.55
      Books in stock 7
      Books on order 10
```

There's a shorthand notation for accessing attributes, elements, and descendants in an XML file: The @ symbol is shorthand for the Attribute property, a pair of angle brackets (<>) is shorthand for the Element property, and two periods (..) are shorthand for the Descendants property. The following code segment is identical to the preceding one, only this time I'm using the shorthand notation. The output will be exactly the same as before.

```
For Each book In doc.Elements("Book")
    Debug.WriteLine("ISBN " & book.@ISBN.ToString)
    Debug.WriteLine("   Title: " & _
                    book...<Name>.Value.ToString)
    Debug.WriteLine("   Price: " & _
                    book...<Price>.Value.ToString)
    Dim stock = book.Element("Stock")
    Debug.WriteLine("      Books in stock " & _
                    stock.@InStock.ToString)
    Debug.WriteLine("      Books on order " & _
                    stock.@OnOrder.ToString)
Next
```

Notice that attributes are returned as strings and have no Value property.

Adding Dynamic Content to an XML Document

The XML documents we've built in our code so far were static. Because XML support is built into VB, you can also create dynamic context, and this is where things get quite interesting. To insert some dynamic content into an XML document, insert the characters <%=. The editor will automatically insert the closing tag, which is %>. Everything within these two tags is treated as VB code and compiled. The two special tags create a placeholder in the document (or an *expression hole*), and the expression you insert in them is an *embedded expression*: You embed a VB expression in your document, and the compiler evaluates the expression and inserts the result in the XML document.

Here's a trivial XML document with an embedded expression. It's the statement that creates a Books document with a Book element (I copied it from a code segment presented earlier in this chapter), and I inserted the current date as an element:

```
Dim doc = _
   New XElement("Books", _
     New XElement("Book", _
         New XAttribute("ISBN", "0000000000001"), _
         New XAttribute("RecordDate", <%= Today %>), _
         New XElement("Price", 11.95), _
         New XElement("Name", "Book Title 1"), _
         New XElement("Stock", _
             New XAttribute("InStock", 12), _
             New XAttribute("OnOrder", 24))), _
```

Let's say you have an array of Product objects and you want to create an XML document with these objects. Listing 17.1 shows the array with the product names.

Listing 17.1: An Array of Product Objects

```
Dim Products() As Product = _
      {New Product With
         {.ProductID = 3, .ProductName = "Product A", _
          .ProductPrice = 8.75, _
          .ProductExpDate = #2/2/2009#}, _
        New Product With _
         {.ProductID = 4, .ProductName = "Product B", _
          .ProductPrice = 19.5}, _
        New Product With _
         {.ProductID = 5, .ProductName = "Product C", _
          .ProductPrice = 21.25, _
          .ProductExpDate = #12/31/2010#}}
```

The code for generating an XML document with three elements is quite short, but what if you had thousands of products? Let's assume that the *Products* array contains instances of the Product class. You can use the XMLSerializer class to generate an XML document with the array's contents. An alternative approach is to create an inline XML document with embedded expressions, as shown in Listing 17.2.

Listing 17.2: An XML Document with Product Objects

```
Dim prods = <Products>
       <%= From prod In Products _
       Select <Product>
```

```
                <ID><%= prod.ProductID %></ID>
                <Name><%= prod.ProductName %></Name>
                <Price><%= prod.ProductPrice %></Price>
                <ExpirationDate>
                     <%= prod.ProductExpDate %></ExpirationDate>
                </Product> %>
                </Products>
```

This code segment looks pretty ugly, but here's how it works: In the first line, we start a new XML document. (The *prods* variable is actually of the XElement type, but an XElement is in its own right an XML document.) Notice that there's no line-continuation character at the end of the first line of the XML document. Then comes a LINQ query embedded in the XML document with the <%= and %> tags. Notice the line continuation symbol at the end of this line. When we're in an expression hole, we're writing VB code, so line breaks matter. That makes the line continuation symbol necessary. Here's a much simplified version of the same code:

```
Dim prods = <Products>
            <%= From prod In Products _
                Select <Product>some product</Product> %>
            </Products>
```

This code segment will generate the following XML document:

```
<Products>
    <Product>some product</Product>
    <Product>some product</Product>
    <Product>some product</Product>
</Products>
```

The file contains no real data but is a valid XML document. The two tags with the percent sign switch into VB code, and the compiler executes the statements embedded in them. The embedded statement of our example is a LINQ query, which iterates through the elements of the Products array and selects literals (the XML tags shown in the output). To insert data between the tags, we must switch to VB again and insert the values we want to appear in the XML document. In other words, we must replace the string *some product* in the listing with some embedded expressions that return the values you want to insert in the XML document. These values are the properties of the Product class, as shown in Listing 17.1. The code shown in Listing 17.2 will produce the output shown in Listing 17.3.

Listing 17.3: An XML Document with the Data of the Array Initialized in Listing 17.2

```
<Products>
  <Product>
    <ID>3</ID>
    <Name>Product A</Name>
    <Price>8.75</Price>
    <ExpirationDate>2009-02-02T00:00:00</ExpirationDate>
  </Product>
```

```
    <Product>
      <ID>4</ID>
      <Name>Product B</Name>
      <Price>19.5</Price>
      <ExpirationDate>0001-01-01T00:00:00</ExpirationDate>
    </Product>
    <Product>
      <ID>5</ID>
      <Name>Product C</Name>
      <Price>21.25</Price>
      <ExpirationDate>2010-12-31T00:00:00</ExpirationDate>
    </Product>
  </Products>
```

Transforming XML Documents

A common operation is the transformation of an XML document. If you have worked with XML in the past, you already know Extensible Stylesheet Language Transformations (XSLT), which is a language for transforming XML documents. If you're new to XML, you'll probably find it easier to transform XML documents with the LINQ to XML component. Even if you're familiar with XSLT, you should be aware that transforming XML documents with LINQ is straightforward. The idea is to create an inline XML document that contains HTML tags and an embedded LINQ query, like the following:

```
Dim HTML = <htlm><b>Products</b>
           <table border="all"><tr>
           <td>Product</td><td>Price</td>
           <td>Expiration</td></tr>
           <%= From item In prods.Descendants("Product") _
               Select <tr><td><%= item.<Name> %></td>
                      <td><%= item.<Price> %></td>
                      <td><%= Convert.ToDateTime( _
                      item.<ExpirationDate>.Value). _
                            ToShortDateString %>
            </td></tr> %></table>
            </htlm>
HTML.Save("Products.html")
Process.Start("Products.html")
```

The `HTML` variable stores plain HTML code. HTML is a subset of XML, and the editor will treat it like XML: It will insert the closing tags for you and will not let you nest tags in the wrong order. The `Select` keyword in the query is followed by a mix of HTML tags and embedded holes for inline expressions, which are the fields of the *item* object. Note the VB code for formatting the date in the last inline expression.

The last two statements save the HTML file generated by our code and then open it in Internet Explorer (or whichever application you've designated to handle by default the HTML documents).

Real World Scenario

Using Custom Functions with LINQ to XML

The embedded expressions are not limited to simple, inline expressions. You can call custom functions to transform your data. In a hotel reservation system I developed recently, I had to transform an XML file with room details to an HTML page. The transformation involved quite a few lookup operations, which I implemented with custom functions. Here's a simplified version of the XLINQ query I used in the project. I'm showing the query that generates a simple HTML table with the elements of the XML document. The `RoomType` element is a numeric value that specifies the type of the room. This value may differ from one supplier to another, so I had to implement the lookup operation with a custom function.

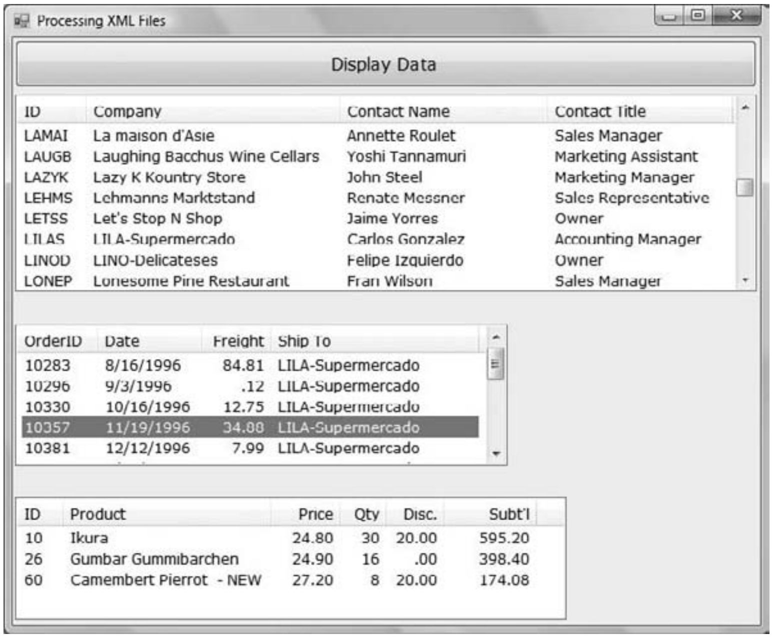
```
Dim hotels = <html>
   <table><tr><td>Hotel</td<td>Room Type</td><td>Price</td></tr>
    <%= From hotel In Hotels _
        Select <tr><td><%= hotel.<HotelName>.Value %></td>
                   <td><%= GetRoomType(hotel.<RoomTypeID>)</td>
                   <td><%= CalculatePrice(hotel.<Base>)</td>
               </tr>
    %>
   </table>
</html>
```

The `GetRoomType()` and `CalculatePrice()` functions must be implemented in the same module that contains the LINQ query. In my case, they accept more arguments than shown here, but you get the idea. To speed up the application, I created HashTables using the IDs of the various entities in their respective tables in the database. The `CalculatePrice()` function, in particular, is quite complicated, because it incorporates the pricing policy of the agency. Yet, all the business logic implemented in a standard VB function was easily incorporated into the LINQ query that generates the HTML page with the available hotels and prices.

Working with XML Files

In this section, we're going to build a functional interface for viewing customers and orders. And this time we aren't going to work with a small sample file. We'll actually get our data from one of the sample databases that come with SQL Server: the Northwind database. The structure of this database is discussed in Chapter 21, ''Basic Concepts of Relational Databases,'' in detail, but for now I'll show you how to extract data in XML format from SQL Server. If you don't have SQL Server installed, or if you're unfamiliar with databases, you can use the sample XML files in the folder of the VBLINQ project. The `frmXMLFiles` form of the VBLINQ project is shown in Figure 17.2.

You may be wondering why you would extract relational data and process them with LINQ instead of executing SQL statements against the database. XML is the standard data- exchange format, and you may get data from any other source in this format. You may get an XML file generated from someone's database, or even an Excel spreadsheet. In the past, you had to convert the data to another, more flexible format and then process it. With LINQ, you can directly query the XML document, transform it into other formats, and of course save it.



ID	Company	Contact Name	Contact Title
LAMAI	La maison d'Asie	Annette Roulet	Sales Manager
LAUGB	Laughing Bacchus Wine Cellars	Yoshi Tannamuri	Marketing Assistant
LAZYK	Lazy K Kountry Store	John Steel	Marketing Manager
LEHMS	Lehmanns Marktstand	Renate Messner	Sales Representative
LETSS	Let's Stop N Shop	Jaime Yorres	Owner
LILAS	LILA-Supermercado	Carlos Gonzalez	Accounting Manager
LINOD	LINO-Delicateses	Felipe Izquierdo	Owner
LONEP	Lonesome Pine Restaurant	Fran Wilson	Sales Manager

OrderID	Date	Freight	Ship To
10283	8/16/1996	84.81	LILA-Supermercado
10296	9/3/1996	.12	LILA-Supermercado
10330	10/16/1996	12.75	LILA-Supermercado
10357	11/19/1996	34.88	LILA-Supermercado
10381	12/12/1996	7.99	LILA-Supermercado

ID	Product	Price	Qty	Disc.	Subt'l
10	Ikura	24.80	30	20.00	595.20
26	Gumbar Gummibarchen	24.90	16	.00	398.40
60	Camembert Pierrot - NEW	27.20	8	20.00	174.08

FIGURE 17.2
Displaying related data from XML files

Start SQL Server and execute the following query:

```
SELECT * FROM Customers FOR XML AUTO
```

This statement selects all columns and all rows for the Customers table and generates an element for each row. The field values are stored in the document as attributes of the corresponding row. The output of this statement is not a valid XML document because its elements are not embedded in a root element. To request an XML document in which all elements are embedded in a root element, use the `ROOT` keyword:

```
SELECT * FROM Customers FOR XML AUTO, ROOT('AllCustomers')
```

I'm using the root element `AllCustomers` because the elements of the XML document are named after the table. The preceding statement will generate an XML document with the following structure:

```
<AllCustomers>
   <Customers CustomerID="..." CompanyName="xxx" ... />
   <Customers CustomerID="..." CompanyName="xxx" ... />
    ...
</AllCustomers>
```

It would make more sense to generate an XML document with the `Customers` root element and name the individual elements `Customer`. To generate this structure, use the following statement:

```
SELECT * FROM Customers Customer FOR XML AUTO, ROOT('Customers')
```

Here's a segment of the XML document with the customers:

```
<Customers>
  <Customer CustomerID="ALFKI" CompanyName=
   "Alfreds Futterkiste" ContactName="Maria Anders"
   ContactTitle="Sales Representative"
   Country="Germany" />
  <Customer CustomerID="ANATR" CompanyName=
   "Ana Trujillo Emparedados y helados"
    ContactName="Ana Trujillo" ContactTitle="Owner"
    Country="Mexico" />
```

Finally, you can create an XML document where the fields are inserted as elements, rather than attributes. To do so, use the ELEMENTS keyword:

```
SELECT * FROM Customers Customer FOR XML AUTO,
   ELEMENTS ROOT('Customers')
```

The other statements that generated the XML files with the rows of the tables Orders, Order Details, and Products are as follows:

```
SELECT * FROM Orders Order FOR XML AUTO,  ROOT('Orders')
SELECT * FROM [Order Details] Detail FOR XML AUTO,
   ELEMENTS, ROOT('Details')
SELECT ProductID, ProductName FROM Products
   FOR XML AUTO, ELEMENTS ROOT('Products')
```

Notice that all files are attribute based, except for the `Details.xml` file, which is element based. I had no specific reason for choosing this structure; I just wanted to demonstrate both styles for processing XML in the sample project's code. Also, the reason I've included the Products table is because the Order Details table, which contains the lines of the order, stores the IDs of the products, not the product names. When displaying orders, as shown in Figure 17.2, we must show product names, not just product IDs. The four collections with the entities we extracted from the Northwind database are declared and populated at the form's level via the following statements:

```
Dim customers As XElement = XElement.Load("..\..\..\Customers.xml")
Dim orders As XElement = XElement.Load("..\..\..\Orders.xml")
Dim details As XElement = XElement.Load("..\..\..\Details.xml")
Dim products As XElement = XElement.Load("..\..\..\Products.xml")
```

As it's apparent from the code, I've placed the four XML files created with the SQL statements shown earlier in the project's folder. The Display Data button populates the top ListView control with the rows of the Customers table, via the following statements:

```
Private Sub bttnShow_Click(...) Handles bttnShow.Click
    For Each c In customers.Descendants("Customer")
        Dim LI As New ListViewItem
        LI.Text = c.@CustomerID
        LI.SubItems.Add(c.@CompanyName)
        LI.SubItems.Add(c.@ContactName)
        LI.SubItems.Add(c.@ContactTitle)
        ListView1.Items.Add(LI)
    Next
End Sub
```

The code is quite simple. It doesn't even use LINQ; it iterates through the Customer elements of the `customers` collection and displays their attributes on the control. Notice the use of the shortcut for the `Attribute` property of the current XElement.

When the user clicks a customer name, the control's `SelectedIndexChanged` event is fired. The code in this handler executes a LINQ statement that selects the rows of the Orders table that correspond to the ID of the selected customer. Then, it iterates through the selected rows, which are the orders of the current customer, and displays their fields on the second ListView control via the following statements:

```
Private Sub ListView1_SelectedIndexChanged(...) _
            Handles ListView1.SelectedIndexChanged
    If ListView1.SelectedItems.Count = 0 Then Exit Sub
    ListView2.Items.Clear()
    Dim scustomerID = ListView1.SelectedItems(0).Text
    Dim query = From o In orders.Descendants("Order") _
                Where Convert.ToString(o.@CustomerID) = scustomerID _
                Select o
    For Each o In query
        Dim LI As New ListViewItem
        LI.Text = o.@OrderID.ToString
        LI.SubItems.Add(Convert.ToDateTime _
                 (o.@OrderDate).ToShortDateString)
        LI.SubItems.Add(Convert.ToDecimal _
(o.@Freight).ToString("#,###.00"))
        LI.SubItems.Add(o.@ShipName.ToString)
        ListView2.Items.Add(LI)
    Next
End Sub
```

The LINQ query selects Order elements based on their `CustomerID` attribute. Finally, when an order is clicked, the following LINQ query retrieves the selected order's details:

```
Dim query = From itm In details.Descendants("Detail") _
            Where Convert.ToInt32(itm.<OrderID>.Value) = orderID _
            Select itm
```

The `Details.xml` file contains elements for all columns, not attributes, and I use statements such as `dtl.<UnitPrice>` to access the subelements of the current element. To display product names, the code selects the row of the `Products` collection that corresponds to the ID of each detail line as follows:

```
Dim product = _
      From p In products.Descendants("Product") _
      Where Convert.ToInt32(p.@ProductID) = _
            Convert.ToInt32(dtl.<ProductID>.Value) _
      Select p
```

The *product* variable is actually a collection of XElements, even though it can never contain more than a single element (product IDs are unique). We access the ProductName column of the selected row with the expression product(0).@productName. You can call the First method to make sure you've selected a single product, no matter what:

```
Dim product = _
      (From p In products.Descendants("Product") _
       Where Convert.ToInt32(p.@ProductID) = _
             Convert.ToInt32(dtl.<ProductID>.Value) _
       Select p).First
```

LINQ to SQL

SQL stands for *Structured Query Language*, a language for querying databases. SQL is discussed in detail in Chapter 21, and as you will see, SQL resembles LINQ. If you are not familiar with databases and SQL, you should read Chapter 21 and then return to this section. SQL is a simple language, and I will explain the SQL statements used in this section's examples; readers who are somewhat familiar with databases should be able to follow the examples of this section.

In this section, we're going to build an application for displaying customers, orders, and order details, just as we did in the preceding section. The difference is that this time we won't get our data from an XML document; we'll retrieve them directly from the database. As you will see, the same LINQ queries will be used to process the rows returned by the queries. The code you'll see in this section comes from the frmDB form of the VBLINQ sample project. This form is identical to the frmXMLFiles form shown in Figure 17.2; it just uses a different data source. The code isn't identical to the code presented in the preceding section, but the differences are minor. The same principles will be applied to a very different data source.

We need a mechanism to connect to the database so we can retrieve data, and this mechanism is the DataContext class. The DataContext class talks to the database, retrieves data, and submits changes back to the database. To create a DataContext object, pass a string with the information about the database server, the specific database, and your credentials to the DataContext class's constructor, as shown here:

```
Dim db As New DataContext("Data Source=localhost;
                  initial catalog=northwind;
                  Integrated Security=True")
```

To use the DataContext class in your code, you must add a reference to the System.Data.Linq namespace and then import it into your code with this statement:

```
Imports System.Data.Linq
```

You will find more information on connecting to databases in Chapter 22. For the purposes of this chapter, the preceding connection string will connect your application to the Northwind database on the local database server, assuming that you have installed SQL Server or SQL Server Express on the same machine as Visual Studio.

After you have initialized the DataContext object, you're ready to read data from tables into variables. To do so, call the GetTable method of the *db* object to retrieve the rows of a table. Note that the name of the table is not specified as an argument. Instead, the table is inferred from the type passed to the GetTable method as an argument. The GetTable(Of *Customer*) method will retrieve the rows of the Customers table, because the name of the table is specified in the definition of the class, as you will see shortly.

```
customers = From cust In db.GetTable(Of Customer)() _
            Select New Customer With _
            {.CustomerID = cust.CustomerID, _
             .CompanyName = cust.CompanyName, _
             .ContactName = cust.ContactName, _
             .ContactTitle = cust.ContactTitle}
orders = From ord In db.GetTable(Of Order)() _
         Select New Order With _
         {.OrderID = ord.OrderID, _
          .OrderDate = ord.OrderDate, _
          .CustomerID = ord.CustomerID, _
          .Freight = ord.Freight, _
          .ShipName = ord.ShipName}
details = From det In db.GetTable(Of Detail)() _
          Select New Detail With _
          {.OrderID = det.OrderID, _
           .ProductID = det.ProductID, _
           .Quantity = det.Quantity, _
           .UnitPrice = det.UnitPrice, _
           .Discount = det.Discount}
products = From prod In db.GetTable(Of NWProduct)() _
           Select New NWProduct With _
           {.ProductID = prod.ProductID, _
            .ProductName = prod.ProductName}
```

The type of the customers, orders, details, and products variables is `IQueryable(of` *`entity`*`)`, where *`entity`* is the appropriate type for the information you're reading from the database. The four variables that will store the rows of the corresponding tables must be declared at the form's level with the following statements:

```
Dim customers As System.Linq.IQueryable(Of Customer)
Dim orders As System.Linq.IQueryable(Of Order)
Dim details As System.Linq.IQueryable(Of Detail)
Dim products As System.Linq.IQueryable(Of NWProduct)
```

The variables must be declared explicitly at the form's level, because they will be accessed from within multiple event handlers.

To make the most of LINQ to SQL, you must first design a separate class for each table that you want to load from the database. You can also specify the mapping between your classes and the tables from which their instances will be loaded, by prefixing them with the appropriate attributes. The Customer class, for example, will be loaded with data from the Customers table. To specify the relationship between the class and the table, use the `Table` attribute, as shown here:

```
<Table(Name:="Customers")>Public Class Customer
End Class
```

Each property of the Customer class will be mapped to a column of the Customers table. In a similar manner, decorate each property with the name of the column that will populate the property:

```
<Column(Name:="CompanyName")>Public Property Name
End Property
```

If the name of the property matches the name of the relevant column, you can omit the column's name:

```
<Column()>Public Property Name
End Property
```

Listing 17.4 shows the definition of the four classes we'll use to store the four tables (Customers, Orders, Order Details and Products).

LISTING 17.4: The Classes for Storing Customers and Orders

```
<Table(Name:="Customers")> Public Class Customer
    Private _CustomerID As String
    Private _CompanyName As String
    Private _ContactName As String
    Private _ContactTitle As String

    <Column()> Public Property CustomerID() As String
        Get
            Return _customerID
        End Get
        Set(ByVal value As String)
            _customerID = value
        End Set
    End Property

    <Column()> Public Property CompanyName() As String
        Get
            Return _CompanyName
        End Get
        Set(ByVal value As String)
            _CompanyName = value
        End Set
    End Property

    <Column()> Public Property ContactName() As String
        ....
    End Property

    <Column()> Public Property ContactTitle() As String
        ....
    End Property
End Class
```

```
<Table(Name:="Orders")> Public Class Order
    Private _OrderID As Integer
    Private _CustomerID As String
    Private _OrderDate As Date
    Private _Freight As Decimal
    Private _ShipName As String

    <Column()> Public Property OrderID() As Integer
        ....
    End Property

    <Column()> Public Property CustomerID() As String
        ....
    End Property

    <Column()> Public Property OrderDate() As Date
        ....
    End Property

    <Column()> Public Property Freight() As Decimal
        ....
    End Property

    <Column()> Public Property ShipName() As String
        ....
    End Property
End Class

<Table(Name:="Order Details")> Public Class Detail
    Private _OrderID As Integer
    Private _ProductID As Integer
    Private _Quantity As Integer
    Private _UnitPrice As Decimal
    Private _Discount As Decimal

    <Column()> Public Property OrderID() As Integer
        ....

    End Property

    <Column()> Public Property ProductID() As Integer
        ....
    End Property

    <Column()> Public Property Quantity() As Short
        ....
    End Property
```

```
        <Column()> Public Property UnitPrice() As Decimal
            ....
        End Property


        <Column()> Public Property Discount() As Double
            ....
        End Property
    End Class

    <Table(Name:="Products")> Public Class NWProduct
        Private _ProductID As Integer
        Private _ProductName As String

        <Column()> Public Property ProductID() As Integer
            ....
        End Property

        <Column()> Public Property ProductName() As String
            ....
        End Property

    End Class
```

I'm not showing the implementation of most properties, because it's trivial. What's interesting in this listing are the Table and Column attributes that determine how the instances of the classes will be populated from the database, as you saw earlier.

The code that displays the selected customer's orders and the selected order's details is similar to the code you saw in the previous section that displays the data from the XML files. It selects the matching rows in the relevant table and shows them in the corresponding ListView control.

Retrieving Data with the *ExecuteQuery* Method

You can also retrieve a subset of the table by executing an SQL query against the database. The `ExecuteQuery` method, which accepts as arguments the `SELECT` statement to be executed and an array with parameter values, returns a collection with the selected rows as objects. To call the `ExecuteQuery` method, you must specify the class that will be used to store the results with the `Of` keyword in parentheses following the method's name. Then you specify the `SELECT` statement that will retrieve the desired rows. If this query contains any parameters, you must also supply an array of objects with the parameter values. Parameters are identified by their order in the query, and not a name. The first parameters is `0`, the second parameter is `1`, and so on. The following statement will retrieve all customers from Germany and store them in instances of the Customer class:

```
Dim params() = {"Germany"}
Dim GermanCustomers = _
          db.ExecuteQuery(Of Customer)( _
          "SELECT CustomerID, CompanyName, " & _
```

```
            "ContactName, ContactTitle " &
            "FROM Customers WHERE Country={0}", params)"
```

After the *GermanCustomers* collection has been populated, you can iterate through its items as usual, with a loop like the following:

```
For Each cust In GermanCustomers
    Debug.WriteLine(cust.CompanyName & " " & _
                     cust.ContactName)
Next
```

You can also execute LINQ queries against it. To find out the number of customers from Germany, use the following expression:

```
Dim custCount = GermanCustomers.Count
```

To apply a filtering expression and then retrieve the count, use the following LINQ expression:

```
Dim g = GermanCustomers.Where(Function(c As Customer) _
            c.CompanyName.ToUpper Like "*DELIKATESSEN*").Count
```

To appreciate the role of the DataContext class in LINQ to SQL, you should examine the `ToString` property of a LINQ query that's executed against the database. Insert a statement to display the expression `GermanCustomers.ToString` in your code and you will see that the DataContext class has generated and executed the following statement against the database. If you're familiar with SQL Server, you can run the SQL Server Profiler and trace all commands executed against SQL Server. Start SQL Server Profiler (or ask the database administrator to create a log of all statements executed by your workstation against a specific database) and then execute a few LINQ to SQL queries. Here's the statement for selecting the German customers as reported by the profiler:

```
exec sp_executesql N'SELECT Customers.CompanyName,
     Orders.OrderID, SUM(UnitPrice*Quantity) AS
         OrderTotal FROM Customers INNER JOIN Orders
     ON Customers.CustomerID = Orders.CustomerID
     INNER JOIN [Order Details] ON
         [Order Details].OrderID = Orders.OrderID
     WHERE Customers.Country=@p0
     GROUP BY Customers.CompanyName,
     Orders.OrderID',N'@p0 nvarchar(7)',@p0=N'Germany'
```

The Bottom Line

Perform simple LINQ queries. A LINQ query starts with the structure `From` *`variable`* `In` *`collection`*, where `variable` is a variable name and *`collection`* is any collection that implements the `IEnumerable` interface (such as an array, a typed collection, or any method that returns a collection of items). The second mandatory part of the query is the `Select` part, which determines the properties of the variable we want in the output. Quite often we select the same variable that we specify in the From keyword. In most cases, we apply a filtering expression with the Where keyword. Here's a typical LINQ query that selects filenames from a specific folder:

```
Dim files = _
        From file In _
          IO.Directory.GetFiles("C:\Documents") _
          Where file.EndsWith("doc") _
        Select file
```

Master It Write a LINQ query that calculates the sum of the squares of the values in an array.

Create and process XML files with LINQ to XML. LINQ to SQL allows you to create XML documents with the XElement and XAttribute classes. You simply create a new XElement object for each element in your document, and a new XAttribute object for each attribute in the current element. Alternatively, you can simply insert XML code in your VB code. To create an XML document dynamically, you can insert embedded expressions that will be evaluated by the compiler and replaced with their results.

Master It How would you create an HTML document with the filenames in a specific folder?

Process relational data with LINQ to SQL. LINQ to SQL allows you to query relational data from a database. To access the database, you must first create a DataContext object. Then you can call this object's GetTable method to retrieve a table's rows, or the ExecuteQuery method to retrieve selected rows from one or more tables with an SQL query. The result is stored in a class designed specifically for the data you're retrieving via the DataContext object.

Master It Explain the attributes you must use in designing a class for storing a table.

Chapter 18

Drawing and Painting with Visual Basic 2008

Some of the most interesting and fun parts of a programming language are its graphics elements. In general, graphics fall into two major categories: vector and bitmap. *Vector graphics* are images generated by graphics methods such as `DrawLine` and `DrawEllipse`. The drawing you create is based on mathematical descriptions of the various shapes. *Bitmap graphics* are images made up of pixels arranged in rows and columns. Each pixel is represented by a Long numeric value, which is the pixel's color. The difference between vector and bitmap graphics is that vector graphics aren't tied to a specific monitor resolution; that is, they can be displayed at various resolutions. Bitmap graphics, on the other hand, have a fixed resolution. An image that is 1,024 pixels wide and 768 pixels tall has that specific resolution. If you attempt to use that image to fill a monitor that's 1,280 pixels wide and 1,024 pixels tall, you'll have to repeat some pixels. Image-processing software can interpolate between pixels, but when you blow up a bitmap, you see its block-like structure.

In this chapter, you will learn how to do the following:

- Display and size images
- Generate graphics by using the drawing methods
- Display text in various ways, including gradient fills

Displaying and Sizing Images

The primary control for displaying images is the PictureBox control. To load an image to a PictureBox control, locate the `Image` property in the Properties window and click the button with the ellipsis next to it. The Select Resource dialog box will appear, in which you can select the image to be displayed (see Figure 18.1). The image, along with every other image or icon you use in the same project, is stored in the Resources folder under the project's folder. As a result, you don't have to distribute the image with your application; it will be included in the setup file that the installer will create for your application.

After the image is loaded, you must make sure that it fills the available space. The PictureBox control exposes the `SizeMode` property, which determines how the image will be sized and aligned on the control. The `SizeMode` property can be set to a member of the `PictureBoxSizeMode` enumeration: AutoSize, CenterImage, Normal, StretchImage, and Zoom. Its default setting is Normal, and in this mode the control displays the image at its normal magnification. If the image is larger than the control, part of the image will be invisible. If the image is smaller than the control, part

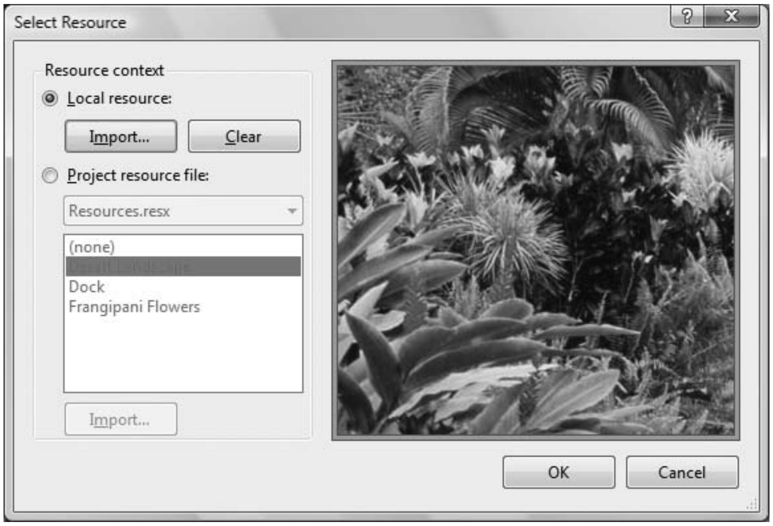


FIGURE 18.1
Adding image resources to a project through the Select Resource dialog box

of the control will be empty. In this case, you can set the `SizeMode` property to CenterImage to center the image on the control.

The StretchImage setting resizes the image so that it fills the control. If the control's aspect ratio isn't the same as the aspect ratio of the image, the image will be distorted in the process. If you want to use the StretchImage setting, you must also resize one of the dimensions of the control, so that the image will be properly resized. You'll see how to do this shortly. The AutoSize setting causes the control to be resized according to the image's dimensions. This is not the most convenient setting because the control might cover other controls on the form.

The Zoom setting of the `SizeMode` property resizes the image without distorting its aspect ratio. In this mode, the control attempts to resize the image as well as it can in the given area while maintaining its aspect ratio. If the image's aspect ratio is different from the aspect ratio of the control, this setting will fill the control vertically or horizontally and will center the image in the other direction.

Figure 18.2 shows a PictureBox control with an image in four of the five settings (the AutoSize mode, which isn't shown in the figure, stretches the PictureBox control to the size of the image) Notice that the Zoom mode filled the PictureBox vertically, but left a margin on either side of the image to avoid distortion of the image's aspect ratio.

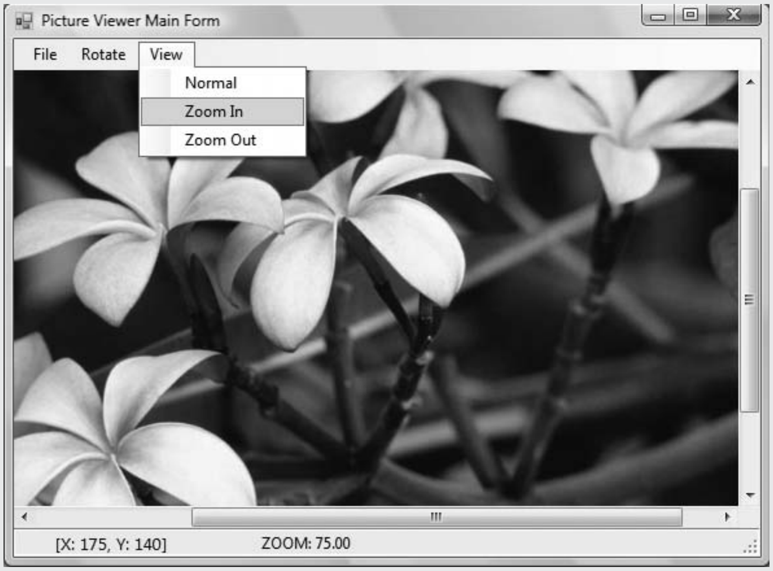
Real World Scenario

DESIGNING A SCROLLING PICTUREBOX

A problem with the PictureBox control is that it doesn't provide an `AutoScroll` property; thus you can't display a large image at its original resolution and scroll any part of it into view at runtime. A scrollable PictureBox would be highly desirable in many applications (images are so common in many types of applications today), but because the control doesn't support this functionality, here's the next best thing you can do:

1. Place a Panel control on the form and set its AutoSize property to True. Set its AutoScroll property to True also, so that the appropriate scroll bars will appear automatically as soon as the control's contents exceed its dimensions. Finally, set it Dock property to *Fill*, so that it will cover the entire form.
2. Place a PictureBox control on the Panel control and set its SizeMode property to *AutoSize*. We want the PictureBox control to be sized according to the image it contains.
3. Finally, assign a large image to the PictureBox control (any of the images in the folder Pictures/Sample Pictures will do). As soon as you assign the image to the control, the necessary scroll bars will be displayed and you can scroll any part of the image into view, even at design time.

Open the Scrolling PictureBox project, shown in the following figure, and experiment with large images. The sample project's main form contains a menu and a status bar, which remain in place as you scroll the PictureBox control with the image in the Panel control. The menu contains commands to zoom in and out of the image as well as commands to rotate the image.



I've also added a few statements to display the coordinates of the upper-left corner of the visible section of the image and the current zoom on the form's status bar. Every time the Panel's contents are scrolled, the Scroll event takes place. I'm using this event handler's arguments to read the horizontal and vertical displacement of the image and print them with the following statements:

```
Private Sub Panel1_Scroll(ByVal sender As Object, _
           ByVal e As System.Windows.Forms.ScrollEventArgs) _
           Handles Panel1.Scroll
    If e.ScrollOrientation = ScrollOrientation.HorizontalScroll Then
        X = e.NewValue
    Else
        Y = e.NewValue
    End If
    ToolStripStatusLabel1.Text = _
              "[X: " & X.ToString & _
              ", Y: " & Y.ToString & "]"
End Sub
```

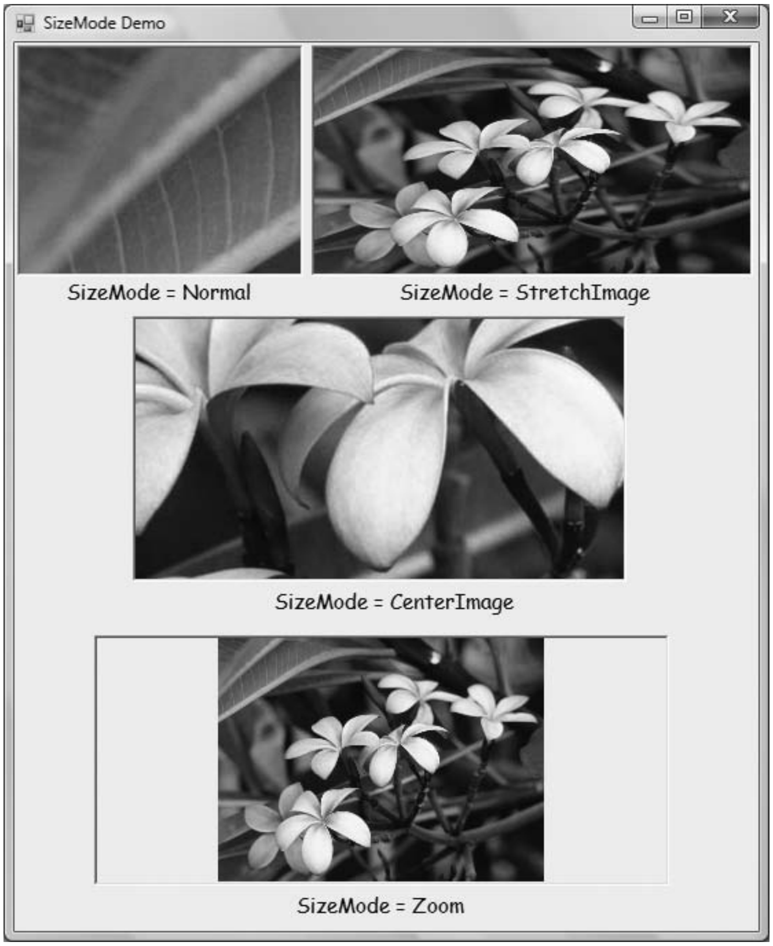


FIGURE 18.2
The settings of the SizeMode property

Drawing with GDI+

You have seen the basics of displaying images on your forms; now let's move on to some real graphics operations, namely how to create your own graphics with the Framework. Windows graphics are based on a graphics engine, known as GDI. *GDI*, which stands for *Graphics Design Interface*, is a collection of classes that enable you to create graphics, text, and images. The most recent version on GDI is called GDI+.

One of the basic characteristics of GDI is that it's stateless. This means that each graphics operation is totally independent of the previous one and can't affect the following one. To draw a line, you must specify a Pen object and the two endpoints of the line. You must do the same for the next line you'll draw. You can't assume that the second line will use the same pen or that it will start at the point where the previous line ended. There isn't even a default font for text-drawing methods. Every time you draw some text, you must specify the font in which the text will be rendered, as well as the Brush object that will be used to draw the text.

The GDI+ classes reside in the following namespaces, and you must import one or more of them in your projects: System.Drawing, System.Drawing2D, System.Drawing.Imaging, and System.Drawing.Text. This chapter explores all three aspects of GDI+ — namely vector drawing, imaging, and typography.

Before you start drawing, you must select the surface you want to draw on, the types of shapes you want to draw, and the instrument you'll use to draw them. The surface on which you can draw is a Graphics object, which is your canvas, and it's the control's `Graphics` property. Most controls expose a `Graphics` property, but most applications draw on either forms or PictureBox controls. The `Graphics` property is an object that exposes numerous methods for drawing basic (and not-so-basic) shapes.

The next step is to decide which instrument you'll use to draw. There are two major drawing instruments: the Pen object and the Brush object. You use pens to draw stroked shapes (lines, rectangles, curves) and brushes to draw filled shapes (any area enclosed by a shape, including text). The main characteristics of the Pen object are its color and its width (the size of the trace left by the pen). The main characteristic of the Brush object is the color or pattern with which it fills the shape. An interesting variation of the Brush object is the gradient brush, which changes color as it moves from one point of the shape you want to fill to another. You can start filling a shape with red in the middle and specify that as you move toward the edges of the shape, the fill color fades to yellow.

After you have specified the drawing surface and the drawing instrument, you draw the actual shape by calling the appropriate method of the Graphics object. To draw lines, call the `DrawLine` method of the Graphics object; to draw text, call the `DrawString` method of the same object. There are many drawing methods, as well as other methods that support the main drawing methods, and they're all discussed later in this chapter. Here are the statements to draw a line on the form:

```
Dim redPen As Pen = New Pen(Color.Red, 2)
Dim point1 As Point = New Point(10,10)
Dim point2 As Point = New Point(120,180)
Me.CreateGraphics.DrawLine(redPen, point1, point2)
```

The first statement declares a new Pen object, which is initialized to draw in red with a width of 2 pixels. The following two statements declare and initialize two points, which are the line's starting and ending points. The coordinates are expressed in pixels, and the origin is at the form's top-left corner. The last statement draws the line by calling the `DrawLine` method. The expression `Me.CreateGraphics` retrieves the Graphics object of the form, which exposes all the drawing methods, including the DrawLine method. You can also create a new Graphics object and associate it with the form:

```
' set up a pen and the two endpoints as before
Dim G As Graphics
G = Me.CreateGraphics
G.DrawLine(redPen, point1, point2)
```

The DrawLine method accepts as an argument the pen it will use to draw and the line's starting and ending points. I have used two Point objects to make the code easier to read. The DrawLine method, like all other drawing methods, is heavily overloaded. You can also omit the declarations of the various objects and initialize them in the same statement that draws the line:

```
Me.CreateGraphics.DrawLine(New Pen(Color.Red, 2), _
            New Point(10, 10), New Point(120, 180))
```

All coordinates are expressed by default in pixels. It's possible to specify coordinates in different units and let GDI+ convert them to pixels before drawing. For now, we'll use pixels, which are quite appropriate for simple objects. After you familiarize yourself with the drawing methods, you can specify different coordinate systems. For more information, see the discussion of the PageUnit property of the Graphics object in the following section.

The Basic Drawing Objects

This is a good point to introduce some of the objects we'll be using all the time when drawing. No matter what you draw or which drawing instrument you use, one or more of the objects discussed in this section will be required.

The Graphics Object

The Graphics object is the drawing surface — your canvas. All the controls you can draw on expose a Graphics property, which is an object, and you can retrieve it with the CreateGraphics method. Conversely, if an object doesn't expose the CreateGraphics method, you can't draw on its surface. It goes without saying that the PictureBox control exposes a Graphics property, but so does the TextBox control, as well as many controls you wouldn't expect. It's not recommended that you draw on a TextBox control, of course, unless you're coding a peculiar application. Bear in mind that anything you draw on the TextBox control will disappear as you start typing. You must first place the text on the control and then draw on its surface — or make the control read-only.

The Graphics object exposes all the methods and properties you will use to create graphics on the control. If you enter the string Me.CreateGraphics and a period, you will see a list of the members of the Graphics object in a drop-down list.

Start by declaring a variable of the Graphics type and initialize it to the Graphics object returned by the control's CreateGraphics method:

```
Dim G As Graphics
G = PictureBox1.CreateGraphics
```

At this point, you're ready to start drawing on the *PictureBox1* control with the methods presented in the following sections. In essence, the CreateGraphics method returns the drawing surface of the control or form on which you wish to draw.

When Do We Initialize a Graphics Object?

The Graphics object is initialized to the control's drawing surface at the moment you create it. If the form is resized at runtime, the Graphics object won't change, and part of the drawing surface might not be available for drawing. If you create a Graphics object to represent a form in the form's `Load` event handler and the form is resized at runtime, the drawing methods you apply to the Graphics object will take effect in part of the form. The most appropriate event for initializing the Graphics object and inserting the painting code is the form's `Paint` event. This event is fired when the form must be redrawn — when the form is uncovered or resized. Insert your drawing code there and create a Graphics object in the `Paint` event handler. Then draw on the Graphics object and release it when you're done.

The Graphics object exposes the following basic properties, in addition to the drawing methods discussed in the following sections.

DpiX, DpiY These two properties return the horizontal and vertical resolutions of the drawing surface, respectively. Resolution is expressed in pixels per inch (or dots per inch, if the drawing surface is your printer). On an average monitor, these two properties return a resolution of 96 dots per inch (dpi).

PageUnit This property determines the units in which you want to express the coordinates on the Graphics object; its value can be a member of the `GraphicsUnit` enumeration (Table 18.1). If you set the `PageUnit` property to *`World`*, you must also set the `PageScale` property to a scaling factor that will be used to convert world units to pixels.

Table 18.1: The *GraphicsUnit* Enumeration

Value	Description
Display	The unit is 1/75 of an inch.
Document	The unit is 1/300 of an inch.
Inch	The unit is 1 inch.
Millimeter	The unit is 1 millimeter.
Pixel	The unit is 1 pixel (the default value).
Point	The unit is a printer's point (1/72 of an inch).
World	The developer specifies the unit to be used.

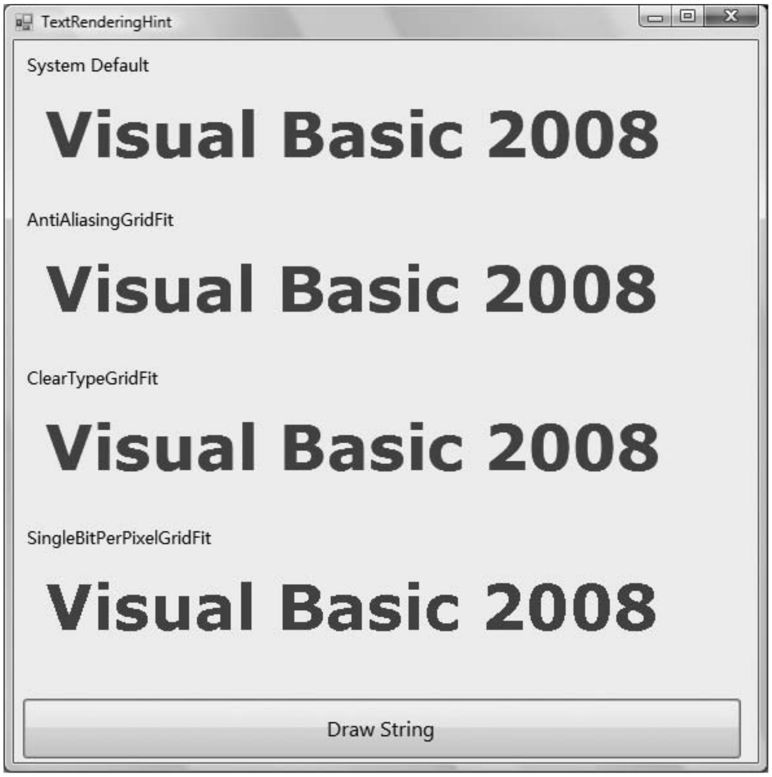
TextRenderingHint This property specifies how the Graphics object will render text; its value is one of the members of the `TextRenderingHint` enumeration: *`AntiAlias`*, *`AntiAliasGridFit`*, *`ClearTypeGridFit`*, *`SingleBitPerPixel`*, *`SingleBitPerPixelGridFit`*, and *`SystemDefault`*.

SmoothingMode This property is similar to the `TextRenderingHint`, but it applies to shapes drawn with the Graphics object's drawing methods. Its value is one of the members of the

SmoothingMode enumeration: *AntiAlias*, *Default*, *HighQuality*, *HighSpeed*, *Invalid*, and *None*.

Figure 18.3 shows the effect of the TextRenderingHint property on text. The anti-aliased text looks much better on the monitor, because anti-aliased text is smoother. The edges of the characters contain shades between the drawing and background colors. The ClearType setting has no effect on Cathode Ray Tube (CRT) monitors. You can see the difference only when you render text on Liquid Crystal Display (LCD) monitors, such as flat-panel or notebook monitors. Text in ClearType style looks best when rendered black on a white background. You won't be able to see the differences among the various settings on the printed image, but you can open the TextRenderingHint project, which I used to create the figure, and examine how the TextRenderingHint property affects the rendering of the text. You can also capture the form by pressing Alt+PrtSc, paste it into Paint or your favorite image-processing application, and zoom into the details of the various characters.

FIGURE 18.3
The effect of the TextRenderingHint setting on the rendering of text



Many of the drawing methods of the Graphics object use some helper classes, such as the Point class that's used to specify coordinates, the Color class that's used to specify colors, and so on. I'll go quickly through these classes, and then I'll discuss the drawing methods in detail.

The Point Class

The Point class represents a point on the drawing surface and is expressed as a pair of (x, y) coordinates. The *x-coordinate* is its horizontal distance from the origin, and the *y-coordinate* is its vertical distance from the origin. The origin is the point with coordinates (0, 0), and this is the top-left corner of the drawing surface.

The constructor of the Point class is the following, where `X` and `Y` are the point's horizontal and vertical distances from the origin:

```
Dim P1 As New Point(X, Y)
```

You can also set the `X` and `Y` properties of the `P1` variable. As you will see later, coordinates can be specified as single numbers, not integers (if you choose to use a coordinate system other than pixels). In this case, use the PointF class, which is identical to the Point class except that its coordinates are nonintegers. (*F* stands for *floating-point*, and floating-point numbers are represented by the Single or Double data type.)

The Rectangle Class

Another class that is often used in drawing is the Rectangle class. The Rectangle object is used to specify areas on the drawing surface. Its constructor accepts as arguments the coordinates of the rectangle's top-left corner and its dimensions:

```
Dim box As Rectangle
box = New Rectangle(X, Y, width, height)
```

The following statement creates a rectangle whose top-left corner is 1 pixel to the right and 1 pixel down from the origin, and its dimensions are 100 by 20 pixels:

```
box = New Rectangle(1, 1, 100, 20)
```

The `box` variable represents a rectangle, but it doesn't generate any output on the monitor. If you want to draw the rectangle, you can pass it as argument to the `DrawRectangle` or `FillRectangle` method, depending on whether you want to draw the outline of the rectangle or a filled rectangle.

Another form of the Rectangle constructor uses a Point and a Size object to specify the location and dimensions of the rectangle:

```
box = New Rectangle(point, size)
```

The `point` argument is a Point object that represents the coordinates of the rectangle's upper-left corner. To create the same Rectangle object as in the preceding example with this form of the constructor, use the following statement:

```
Dim P As New Point(1, 1)
Dim S As New Size(100, 20)
box = New Rectangle(P, S)
```

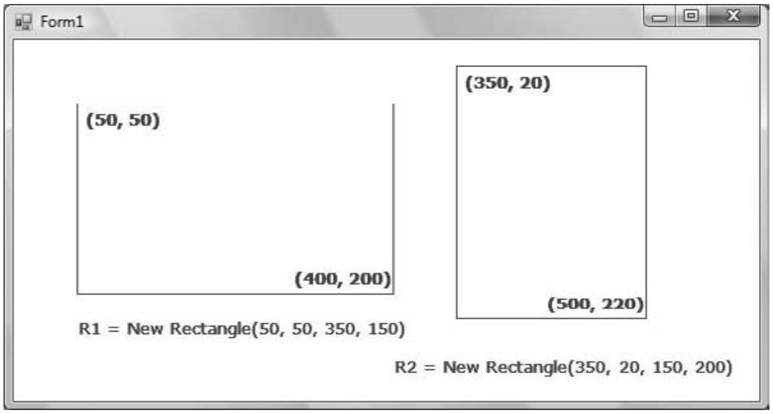

Form1
(50, 50)
(400, 200)
(350, 20)
(500, 220)
R1 = New Rectangle(50, 50, 350, 150)
R2 = New Rectangle(350, 20, 150, 200)

FIGURE 18.4 Specifying rectangles with the coordinates of their top-left corner and their dimensions

Both sets of statements create a rectangle that extends from point (1, 1) to the point (1 + 100, 1 + 20) or (101, 21), in the same manner as the ones shown in Figure 18.4. Alternatively, you can declare a Rectangle object and then set its `X`, `Y`, `Width`, and `Height` properties.

THE SIZE CLASS

The Size class represents the dimensions of a rectangle; it's similar to a Rectangle object, but it doesn't have an origin, just dimensions. To create a new Size object, use the following constructor:

```
Dim S1 As New Size(100, 400)
```

If you want to specify coordinates as fractional numbers, use the SizeF class, which is identical to the Size class except that its dimensions are nonintegers.

THE COLOR CLASS

The Color class represents colors, and there are many ways to specify a color. We'll discuss the Color class in more detail in Chapter 19, ''Manipulating Images and Bitmaps.'' In the meantime, you can specify colors by name. Declare a variable of the Color type and initialize it to one of the named colors exposed as properties of the Color class:

```
Dim myColor As Color
myColor = Color.Azure
```

The 128 color names of the Color class will appear in the IntelliSense box as soon as you enter the period following the keyword *Color*. You can also use the `FromARGB` method, which creates a new color from its basic color components (the Red, Green, and Blue components). For more information on specifying colors with this method, see the section called ''Specifying Colors'' in Chapter 19.

THE FONT CLASS

The Font class represents fonts, which are used when rendering strings via the `DrawString` method. To specify a font, you must create a new Font object; set its family name, size, and style;

and then pass it as argument to the `DrawString` method. Alternatively, you can prompt the user for a font via the Font common dialog box and use the object returned by the dialog box's `Font` property as an argument with the `DrawString` method. To create a new Font object, use a statement like the following:

```
Dim drawFont As New Font("Verdana", 12, FontStyle.Bold)
```

The Font constructor has 13 forms in all. Two of the simpler forms of the constructor, which allow you to specify the size and the style of the font, are shown in the following code lines, where *`size`* is an integer and *`style`* is a member of the `FontStyle` enumeration (Bold, Italic, Regular, Strikeout, and Underline):

```
Dim drawFont As New Font(name, size)
Dim drawFont As New Font(name, size, style)
```

To specify multiple styles, combine them with the `OR` operator:

```
FontStyle.Bold Or FontStyle.Italic
```

You can also initialize a `Font` variable to an existing font. The following statement creates a Font object and initializes it to the current font of the form:

```
Dim textFont As New Font
textFont = Me.Font
```

The Pen Class

The Pen class represents virtual pens, which you use to draw on the Graphics object's surface. To construct a new Pen object, you must specify the pen's color and width in pixels. The following statements declare three Pen objects with the same color and different widths:

```
Dim thinPen, mediumPem, thickPen As Pen
thinPen = New Pen(Color.Black, 1)
mediumPen = New Pen(Color.Black, 3)
thickPen = New Pen(Color.Black, 5)
```

If you omit the second argument, a pen with a width of a single pixel will be created by default. Another form of the Pen object's constructor allows you to specify a brush instead of a color, as follows, where *`brush`* is a Brush object (discussed later in this chapter):

```
Dim patternPen as Pen
patternPen = New Pen(brush, width)
```

The quickest method of creating a new Pen object is to use the built-in `Pens` collection, which creates a Pen with a width of 1 pixel and the color you specify. The following statement can appear anywhere a Pen object is required and will draw shapes in blue color:

```
Pens.Blue
```

The Pen object exposes these properties:

Alignment Determines the alignment of the Pen, and its value is one of the members of the `PenAlignment` enumeration: Center or Inset. When set to *`Center`*, the width of the pen is centered on the outline (half the width is inside the shape, and half is outside). When set to *`Inset`*, the entire width of the pen is inside the shape. The default value of this property is *`PenAlignment.Center`*.

LineJoin Determines how two consecutive line segments will be joined. Its value is one of the members of the `LineJoin` enumeration: *`Bevel`*, *`Miter`*, *`MiterClipped`*, and *`Round`*.

StartCap, EndCap Determines the caps at the two ends of a line segment, respectively. Their value is one of the members of the `LineCap` enumeration: *`Round`*, *`Square`*, *`Flat`*, *`Diamond`*, and so on.

DashCap Determines the caps to be used at the beginning and end of a dashed line. Its value is one of the members of the `DashCap` enumeration: *`Flat`*, *`Round`*, and *`Triangle`*.

DashStyle Determines the style of the dashed lines drawn with the specific Pen. Its value is one of the members of the `DashStyle` enumeration (*`Solid`*, *`Dash`*, *`DashDot`*, *`DashDotDot`*, *`Dot`*, and *`Custom`*).

PenType Determines the style of the Pen; its value is one of the members of the `PenType` enumeration: *`HatchFilled`*, *`LinearGradient`*, *`PathGradient`*, *`SolidColor`*, and *`TextureFill`*.

The Brush Class

The Brush class represents the instrument for filling shapes; you can create brushes that fill with a solid color, a pattern, or a bitmap. In reality, there's no Brush object. The Brush class is actually an abstract class that is inherited by all the classes that implement a brush, but you can't declare a variable of the Brush type in your code. The brush objects are shown in Table 18.2.

Table 18.2: Brush Styles

Brush	Fill Effect
SolidBrush	Fills shapes with a solid color
HatchBrush	Fills shapes with a hatched pattern
LinearGradientBrush	Fills shapes with a linear gradient
PathGradientBrush	Fills shapes with a gradient that has one starting color and many ending colors
TextureBrush	Fills shapes with a bitmap

Solid Brushes

To fill a shape with a solid color, you must create a SolidBrush object with the following constructor, where *`brushColor`* is a color value, specified with the help of the Color object:

```
Dim sBrush As SolidBrush
sBrush = New SolidBrush(brushColor)
```

Every filled object you draw with the *sBrush* object will be filled with the color of the brush.

Hatched Brushes

To fill a shape with a hatch pattern, you must create a HatchBrush object with the following constructor:

```
Dim hBrush As HatchBrush
HBrush = New HatchBrush(hatchStyle, hatchColor, backColor)
```

The first argument is the style of the hatch, and it can have one of the values shown in Table 18.3 and in the following illustration. The `HatchStyle` enumeration has 54 members, so Table 18.3 shows only a few common patterns. You can fill shapes with plaid, spheres, waves, and a lot more patterns that aren't listed here, but you will see their names in the IntelliSense box. The other two arguments are the colors to be used in the hatch. The hatch is a pattern of lines drawn on a background, and the two color arguments are the color of the hatch lines and the color of the background on which the hatch is drawn.

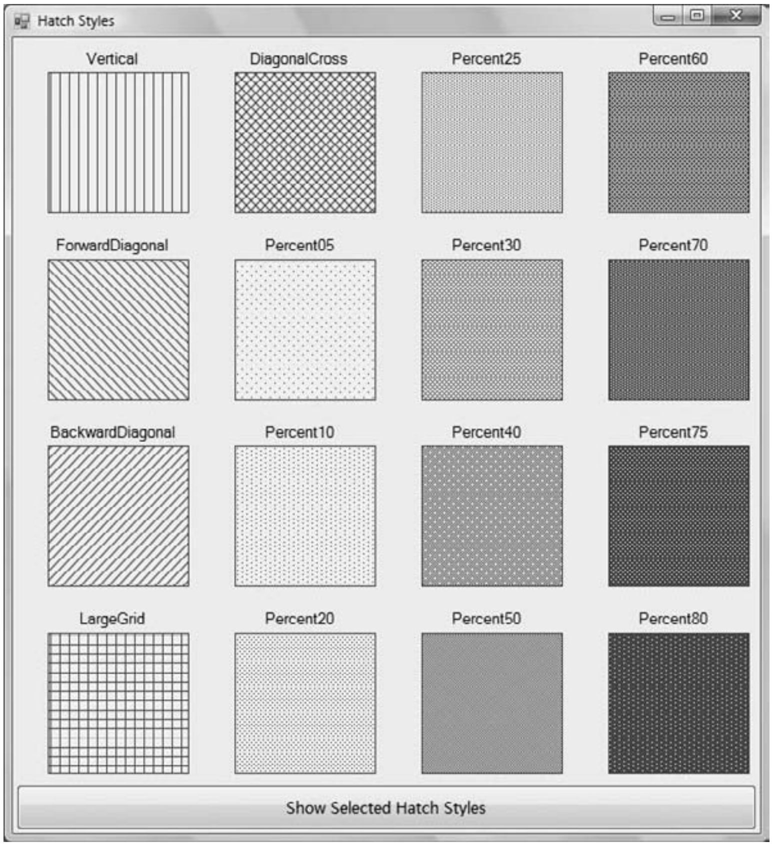


TABLE 18.3: The *HatchStyle* Enumeration

VALUE	EFFECT
BackwardDiagonal	Diagonal lines from top-right to bottom-left
Cross	Vertical and horizontal crossing lines
DiagonalCross	Diagonally crossing lines
ForwardDiagonal	Diagonal lines from top-left to bottom-right
Horizontal	Horizontal lines
Vertical	Vertical lines

Gradient Brushes

A gradient brush fills a shape with a specified gradient. The LinearGradientBrush fills a shape with a linear gradient, and the PathGradientBrush fills a shape with a gradient that has one starting color and one or more ending colors. Gradient brushes are discussed in detail in the section titled ''Gradients,'' later in this chapter.

Textured Brushes

In addition to solid and hatched shapes, you can fill a shape with a texture by using a TextureBrush object. The texture is a bitmap that is tiled as needed to fill the shape. Textured brushes are used to create rather fancy graphics, and we won't explore them in this book.

THE PATH CLASS

The Path class represents shapes made up of various drawing entities, such as lines, rectangles, and curves. You can combine as many of these drawing entities as you'd like and build a new entity, which is called a *path*. Paths are usually closed and filled with a color, a gradient, or a bitmap. You can create a path in several ways. The simplest method is to create a new Path object and then use one of the following methods to append the appropriate shape to the path:

`AddArc`	`AddEllipse`	`AddPolygon`
`AddBezier`	`AddLine`	`AddRectangle`
`AddCurve`	`AddPie`	`AddString`

These methods add to the path the same shapes you can draw on the Graphics object with the methods discussed in the following section. There's even an `AddPath` method, which adds an existing path to the current one. The syntax of the various methods that add shapes to a path is identical to the corresponding methods that draw. We simply omit the first argument (the Pen object) because all the shapes that make up a path will be rendered with the same pen. The following method draws an ellipse:

```
Me.CreateGraphics.DrawEllipse(mypen, 10, 30, 40, 50)
```

To add the same ellipse to a Path object, use the following statement:

```
Dim myPath As New Path
myPath.AddEllipse(10, 30, 40, 50)
```

To display the path, call the `DrawPath` method, passing a Pen and Path object as arguments:

```
Me.CreateGraphics.DrawPath(myPen, myPath)
```

Why combine shapes into paths instead of drawing individual shapes? After the shape has been defined, you can draw multiple instances of it, draw the same path with a different pen, or fill the path's interior with a gradient. Paths are also used to create the ultimate type of gradient, the PathGradient (as you will see in the section called ''Path Gradients,'' later in this chapter).

Drawing Shapes

Now that we've covered the auxiliary drawing objects, we can look at the drawing methods of the Graphics class. Before getting into the details of the drawing methods, however, let's write a simple application that draws a couple of simple shapes on a form. First, we must create a Graphics object with the following statements:

```
Dim G As Graphics
G = Me.CreateGraphics
```

Everything you'll draw on the surface represented by the `G` object will appear on the form. Then, we must create a Pen object to draw with. The following statement creates a Pen object that's 1 pixel wide and draws in blue:

```
Dim P As New Pen(Color.Blue)
```

We created the two basic objects for drawing: the drawing surface and the drawing instrument. Now we can draw shapes by calling the Graphics object's drawing methods. The following statement will print a rectangle with its top-left corner near the top-left corner of the form (at a point that's 10 pixels to the right and 10 pixels down from the form's corner) and is 200 pixels wide and 150 pixels tall. These are the values you must pass to the `DrawRectangle` method as arguments, along with the Pen object that will be used to render the rectangle:

```
G.DrawRectangle(P, 10, 10, 200, 150)
```

Let's add the two diagonals of the rectangle with the following statements:

```
G.DrawLine(P, 10, 10, 210, 160)
G.DrawLine(P, 210, 10, 10, 160)
```

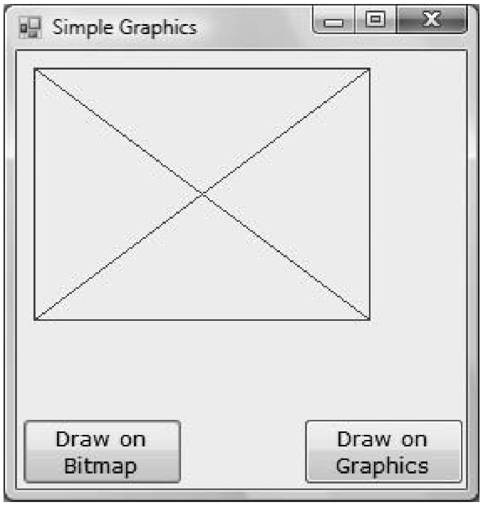
We wrote all the statements to create a shape on the form, but where do we insert them? Let's try a button. Start a new project, place a button on it, and then insert the statements of Listing 18.1 in the button's `Click` event handler.

LISTING 18.1: Drawing Simple Shapes

```
Private Sub Button1_Click(...) Handles Button1.Click
   Dim G As Graphics
   G = Me.CreateGraphics
   Dim P As New Pen(Color.Blue)
   G.DrawRectangle(P, 10, 10, 200, 150)
   G.DrawLine(P, 10, 10, 210, 160)
   G.DrawLine(P, 210, 10, 10, 160)
End Sub
```

Run the application and click the Draw On Graphics button. You will see the shape shown in Figure 18.5. This figure was created by the SimpleShapes sample application.

FIGURE 18.5
The output of Listing 18.1



PERSISTENT DRAWING

If you switch to the Visual Studio IDE or any other window, and then return to the form of the SimpleShapes application, you'll see that the drawing has disappeared! The same will happen if you minimize the window and then restore it to its normal size. Everything you draw on the Graphics object is temporary. It doesn't become part of the Graphics object and is visible only while the control, or the form, need not be redrawn. As soon as the form is redrawn, the shapes disappear.

So, how do we make the output of the various drawing methods permanent on the form? Microsoft suggests placing all the graphics statements in the `Paint` event handler, which is triggered automatically when the form is redrawn. The `Paint` event handler passes the `e` argument, which (among other properties) exposes the form's Graphics object. You can create a Graphics object in the `Paint` event handler and then draw on this object.

Listing 18.2 is the `Paint` event handler that creates the shape shown in Figure 18.5 and refreshes the form every time it's totally or partially covered by another form. Delete the code in the button's `Click` event handler and insert the statements of Listing 18.2 into the `Paint` event's handler, as shown here. (Notice that the Graphics object is a property of the *PaintEventArgs* argument of the event handler.)

LISTING 18.2: Drawing Simple Shapes in the *Paint* Event

```
Private Sub Form1_Paint(ByVal sender As Object, _
                ByVal e As System.Windows.Forms.PaintEventArgs) _
                Handles Me.Paint
   Dim G As Graphics
   G = e.Graphics
   Dim P As New Pen(Color.Blue)
   G.DrawRectangle(P, 10, 10, 200, 150)
   G.DrawLine(P, 10, 10, 210, 160)
   G.DrawLine(P, 210, 10, 10, 160)
End Sub
```

If you run the application now, it works like a charm. The shapes appear to be permanent, even though they're redrawn every time you switch to the form. This technique is fine for a few graphics elements you want to place on the form to enhance its appearance. But many applications draw something on the form in response to user actions, such as the click of a button or a menu command. Using the Form's `Paint` event in a similar application is out of the question. The drawing isn't always the same, and you must figure out from within your code which shapes you have to redraw at any given time. The solution is to make the drawing permanent on the Graphics object, so it won't have to be redrawn every time the form is hidden or resized.

FORCING REFRESHES

A caveat of drawing from within the `Paint` event is that it isn't fired when the form is resized by default. To force a refresh when the form is resized, you must insert the following statement in the form's `Load` event handler:

```
Me.SetStyle(ControlStyles.ResizeRedraw, True)
```

It is possible to make the graphics permanent by drawing not on the Graphics object, but directly on the control's (or the form's) bitmap. The Bitmap object contains the pixels that make up the image and is very similar to the Image object. As you will see in the following chapter, you can create a Bitmap object and assign it to an Image object. To create this ''permanent'' drawing surface, you must first create a Bitmap object that has the same dimensions as the form (or PictureBox control) on which you want to draw:

```
Dim bmp As Bitmap
bmp = New Bitmap(Me.Width, Me.Height)
```

The *bmp* variable represents an empty bitmap. Set the control's `Image` property to this bitmap by using the following statement:

```
Me.BackGroundImage = bmp
```

Immediately after that, you must set the bitmap to the control's background color via the `Clear` method:

```
G.Clear(Me.BackColor)
```

If you're using the PictureBox control to draw on, replace the `BackgroundImage` property with the `Image` property. After the execution of this statement, anything we draw on the *bmp* bitmap is shown on the surface of the PictureBox control and is permanent. All we need is a Graphics object that represents the bitmap, so that we can draw on the control. The following statement creates a Graphics object based on the *bmp* variable:

```
Dim G As Graphics
G = Graphics.FromImage(bmp)
```

Now, we're in business. We can call the *G* object's drawing methods to draw and create permanent graphics on the form. You can put all the statements presented so far in a function that returns a Graphics object (Listing 18.3) and use it in your applications.

Listing 18.3: Retrieving a Graphics Object from a Form's Bitmap

```
Function GetGraphicsObject(ByVal PBox As PictureBox) As Graphics
   Dim bmp As Bitmap
   bmp = New Bitmap(Me.Width, Me.Height)
   Dim G As Graphics
   Me.BackgroundImage = bmp
   G = Graphics.FromImage(bmp)
   Return G
End Function
```

To create permanent drawings on the surface of the form, you must call the `GetGraphicsObject()` function to obtain a Graphics object from the form's bitmap. Listing 18.4 is the revised `GetGraphicsObject()` function for the PictureBox control.

Listing 18.4: Retrieving a Graphics Object from a PictureBox Control's Bitmap

```
Function GetGraphicsObject() As Graphics
   Dim bmp As Bitmap
   bmp = New Bitmap(PBox.Width, PBox.Height)
   PBox.Image = bmp
   Dim G As Graphics
   G = Graphics.FromImage(bmp)
   Return G
End Function
```

Now that you know how to draw on the Graphics object and you're familiar with the basic drawing objects, we can discuss the drawing methods in detail. In the following sections, I use the `CreateGraphics` method to retrieve the drawing surface of a PictureBox or form to keep the examples short. You can modify any of the projects to draw on the Graphics object derived from a bitmap. All you have to do is replace the statements that create the *`G`* variable with a call to the `CreateGraphics()` function.

Drawing Methods

The Framework provides several drawing methods, one for each basic shape. You can create much more elaborate shapes by combining the methods described in the following sections.

All drawing methods have a few things in common. The first argument is always a Pen object, which will be used to render the shape on the Graphics object. The following arguments are the parameters of a shape: They determine the location and dimensions of the shape. The `DrawLine` method, for example, needs to know the endpoints of the line to draw, whereas the `DrawRectangle` method needs to know the origin and dimensions of the rectangle to draw. The parameters needed to render the shape are passed as arguments to each drawing method, following the Pen object.

The drawing methods can also be categorized in two major groups: the methods that draw stroked shapes (outlines) and the methods that draw filled shapes. The methods in the first group start with the `Draw` prefix (`DrawRectangle`, `DrawEllipse`, and so on). The methods of the second group start with the `Fill` prefix (`FillRectangle`, `FillEllipse`, and so on). Of course, some `Draw`*`XXX`* methods don't have an equivalent `Fill`*`XXX`* method. For example, you can't fill a line or an open curve, so there are no `FillLine` or `FillCurve` methods.

Another difference between the drawing and filling methods is that the filling methods use a Brush object to fill the shape — you can't fill a shape with a pen. So, the first argument of the methods that draw filled shapes is a Brush object, not a Pen object. The remaining arguments are the same because you must still specify the shape to be filled. In the following sections, I present in detail the shape-drawing methods but not the shape-filling methods. If you can use a drawing method, you can just as easily use its filling counterpart.

Table 18.4 shows the names of the drawing methods. The first column contains the methods for drawing stroked shapes, and the second column contains the corresponding methods for drawing filled shapes (if there's a matching method).

Some of the drawing methods allow you to draw multiple shapes of the same type, and they're properly named `DrawLines`, `DrawRectangles`, and `DrawBeziers`. We simply supply more shapes as arguments, and they're drawn one after the other with a single call to the corresponding method. The multiple shapes are stored in arrays of the same type as the individual shapes. The `DrawRectangle` method, for example, accepts as an argument the Rectangle object to be drawn. The `DrawRectangles` method accepts as an argument an array of Rectangle objects and draws all of them at once.

DrawLine

The `DrawLine` method draws a straight-line segment between two points with a pen supplied as an argument. The simplest forms of the `DrawLine` method are the following, where *`point1`* and *`point2`* are either Point or PointF objects, depending on the coordinate system in use:

```
Graphics.DrawLine(pen, X1, Y1, X2, Y2)
Graphics.DrawLine(pen, point1, point2)
```

TABLE 18.4: The Drawing Methods

DRAWING METHOD	FILLING METHOD	DESCRIPTION
`DrawArc`		Draws an arc
`DrawBezier`		Draws very smooth curves with fixed endpoints, whose exact shape is determined by two control points
`DrawBeziers`		Draws multiple Bezier curves in a single call
`DrawClosedCurve`	`FillClosedCurve`	Draws a closed curve
`DrawCurve`		Draws curves that pass through certain points
`DrawEllipse`	`FillEllipse`	Draws an ellipse
`DrawIcon`		Renders an icon on the Graphics object
`DrawImage`		Renders an image on the Graphics object
`DrawLine`		Draws a line segment
`DrawLines`		Draws multiple line segments in a single call
`DrawPath`	`FillPath`	Draws a GraphicsPath object
`DrawPie`	`FillPie`	Draws a pie section
`DrawPolygon`	`FillPolygon`	Draws a polygon (a series of line segments between points)
`DrawRectangle`	`FillRectangle`	Draws a rectangle
`DrawRectangles`	`FillRectangles`	Draws multiple rectangles in a single call
`DrawString`		Draws a string in the specified font on the drawing surface
	`FillRegion`	Fills a Region object

DRAWRECTANGLE

The `DrawRectangle` method draws a stroked rectangle and has two forms:

```
Graphics.DrawRectangle(pen, rectangle)
Graphics.DrawRectangle(pen, X1, Y1, width, height)
```

The *`rectangle`* argument is a Rectangle object that specifies the shape to be drawn. In the second form of the method, the arguments *`X1`* and *`Y1`* are the coordinates of the rectangle's top-left corner, and the other two arguments are the dimensions of the rectangle. All these arguments can be integers or singles, depending on the coordinate system in use. However, they must be all of the same type.

The following statements draw two rectangles, one inside the other. The outer rectangle is drawn with a red pen with the default width, whereas the inner rectangle is drawn with a 3-pixel-wide green pen and is centered within the outer rectangle:

```
G.DrawRectangle(Pens.Red, 100, 100, 200, 100)
G.DrawRectangle(New Pen(Color.Green, 3), _
                125, 125, 150, 50)
```

DrawEllipse

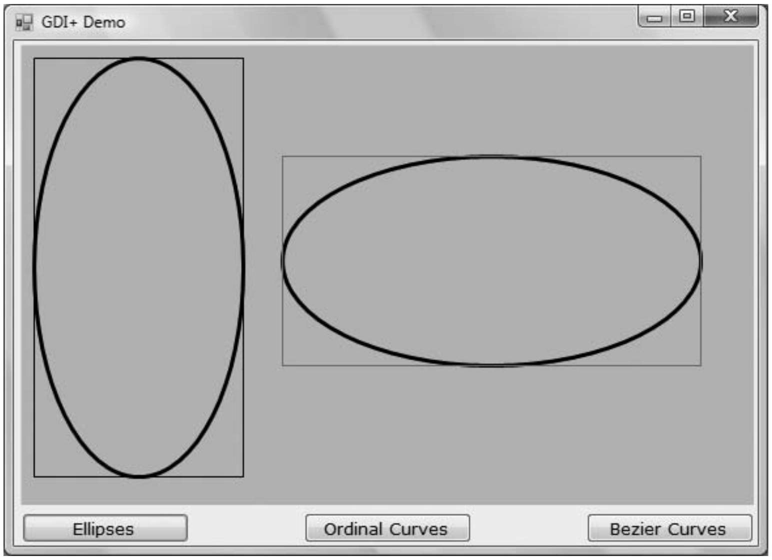
An *ellipse* is an oval or circular shape, determined by the rectangle that encloses it. The two dimensions of this rectangle are the ellipse's major and minor diameters. Instead of giving you a mathematically correct definition of an ellipse, I prepared a few ellipses with different ratios of their two diameters (these ellipses are shown in Figure 18.6). The figure was prepared with the GDIPlus sample application, which demonstrates a few more graphics operations. The ellipse is oblong along the direction of the major diameter and squashed along the direction of the minor diameter. If the two diameters are exactly equal, the ellipse becomes a circle. Indeed, the circle is just a special case of the ellipse, and there's no `DrawCircle` method.

To draw an ellipse, call the `DrawEllipse` method, which has two basic forms:

```
Graphics.DrawEllipse(pen, rectangle)
Graphics.DrawEllipse(pen, X1, Y1, width, height)
```

The arguments are the same as with the `DrawRectangle` method because an ellipse is basically a circle deformed to fit in a rectangle. The two ellipses and their enclosing rectangles shown in Figure 18.6 were generated with the statements of Listing 18.5.

FIGURE 18.6
Two ellipses with their enclosing rectangles



LISTING 18.5: Drawing Ellipses and Their Enclosing Rectangles

```
Private Sub bttnEllipses_Click(...) Handles bttnEllipses.Click
    Dim G As Graphics
    G = PictureBox1.CreateGraphics
    G.SmoothingMode = Drawing.Drawing2D.SmoothingMode.AntiAlias
    Dim R1, R2 As Rectangle
    R1 = New Rectangle(10, 10, 160, 320)
    R2 = New Rectangle(200, 85, 320, 160)
    G.DrawEllipse(New Pen(Color.Black, 3), R1)
    G.DrawRectangle(Pens.Black, R1)
    G.DrawEllipse(New Pen(Color.Black, 3), R2)
    G.DrawRectangle(Pens.Red, R2)
End Sub
```

The ellipses were drawn with a 3-pixel-wide pen. As you can see in the figure, the width of the ellipse is split to the inside and outside of the enclosing rectangle, which is drawn with a 1-pixel-wide pen.

DrawPie

A *pie* is a shape similar to a slice of pie (an arc along with the two line segments that connect its endpoints to the center of the circle or the ellipse, to which the arc belongs). The `DrawPie` method accepts as arguments the pen with which it will draw the shape, the circle to which the pie belongs, the arc's starting angle, and its sweep angle. The circle (or the ellipse) of the pie is defined with a rectangle. The starting and sweeping angles are measured clockwise. The `DrawPie` method has two forms:

```
Graphics.DrawPie(pen, rectangle, start, sweep)
Graphics.DrawPie(pen, X, Y, width, height, start, sweep)
```

The two forms of the method differ in how the rectangle is defined (a Rectangle object versus its coordinates and dimensions). The *`start`* argument is the pie's starting angle, and *`sweep`* is the angle of the pie. The ending angle is *`start`* + *`sweep`*. Angles are measured in degrees (there are 360 degrees in a circle) and increase in a clockwise direction. The 0 angle corresponds to the horizontal axis.

The statements of Listing 18.6 create a pie chart by drawing individual pie slices. Each pie starts where the previous one ends, and the sweeping angles of all pies add up to 360 degrees, which corresponds to a full rotation (a full circle). Unlike the other samples of this section, I've used the `FillPie` method, because we hardly ever draw the outlines of the pies; we fill each one with a different color instead. Figure 18.7 shows the output produced by Listing 18.6.

LISTING 18.6: Drawing a Simple Pie Chart with the *FillPie* Methods

```
Private Sub bttnPie_Click(...) Handles bttnPie.Click
   Dim G As System.Drawing.Graphics
   G = Me.CreateGraphics
```

```
        Dim brush As System.drawing.SolidBrush
        Dim rect As Rectangle
        brush = New System.Drawing.SolidBrush(Color.Green)
        Dim Angles() As Single = {0, 43, 79, 124, 169, 252, 331, 360}
        Dim Colors() As Color = {Color.Red, Color.Cornsilk, _
                                 Color.Firebrick, Color.OliveDrab, _
                                 Color.LawnGreen, Color.SandyBrown, _
                                 Color.MidnightBlue}
        G.Clear(Color.Ivory)
        rect = New Rectangle(100, 10, 300, 300)
        Dim angle As Integer
        For angle = 1 To Angles.GetUpperBound(0)
           brush.Color = Colors(angle - 1)
           G.FillPie(brush, rect, Angles(angle - 1), _
                     Angles(angle) - Angles(angle - 1))
        Next
        G.DrawEllipse(Pens.Black, rect)
     End Sub
```

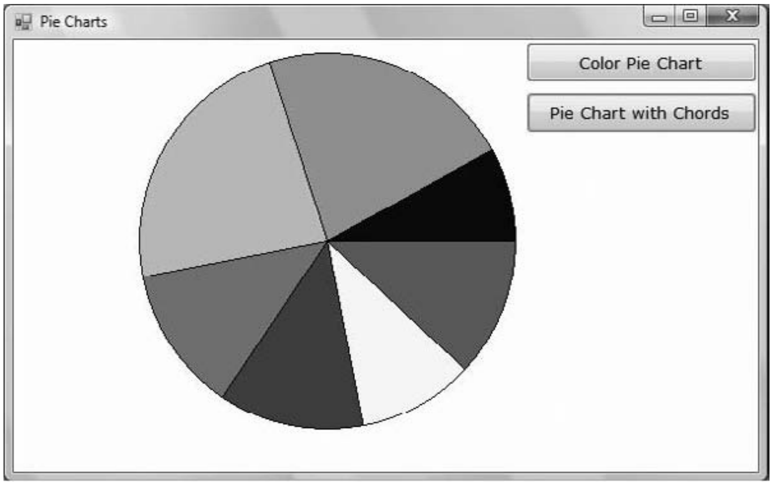


FIGURE 18.7
A simple pie chart generated with the `FillPie` method

The code sets up two arrays: one with angles and another with colors. The *`Angles`* array holds the starting angle of each pie. The sweep angle of each pie is the difference between its own starting angle and the starting angle of the following pie. The sweep angle of the first pie is `Angles(1)` − `Angles(0)`, which is 43 degrees. The loop goes through each pie and draws it with a color it picks from the *`Colors`* array, based on the angles stored in the *`Angles`* array. In your application, you must calculate the total of a quantity (such as all customers, or all units of a product sold in a territory) and then use the individual percentages to set the starting and ending angles of each pie. If there are 800 customers and 20 of them belong to a specific area, this area's sweep angle should be 1/40 of the circle, which is 9 degrees.

Notice that the `FillPie` method doesn't connect the pie's endpoints to the center of the ellipse. The second button on the PieChart project's form draws the same pie chart, but it also connects each slice's endpoints to the center of the circle. The code behind this button is identical to the code shown in Listing 18.6 — with the exception that after calling the `FillPie` method, it calls the `DrawPie` method to draw the outline of the pie.

DrawPolygon

The `DrawPolygon` method draws an arbitrary polygon. It accepts two arguments: the Pen that it will use to render the polygon and an array of points that define the polygon. The polygon has as many sides (or vertices) as there are points in the array, and it's always closed, even if the first and last points are not identical. In fact, you do not need to repeat the starting point at the end because the polygon will be automatically closed. The syntax of the `DrawPolygon` method is the following:

```
Graphics.DrawPolygon(pen, points())
```

where *`points`* is an array of points, which can be declared with a statement like the following:

```
Dim points() As Point = {New Point(x1, y1), New Point(x2, y2), ...}
```

DrawCurve

Curves are smooth lines drawn as *cardinal splines*. A real spline is a flexible object (made of soft wood) that designers used to flex on the drawing surface with spikes. The spline goes through all the fixed points and assumes the smoothest possible shape, given the restrictions imposed by the spikes. If the spline isn't flexible enough, it breaks. In modern computer graphics, there are mathematical formulas that describe the path of the spline through the fixed points and take into consideration the tension (the degree of flexibility) of the spline. A more flexible spline yields a curve that bends easily. Less-flexible splines do not bend easily around their fixed points. Computer-generated splines do not break, but they can take unexpected shapes.

To draw a curve with the `DrawCurve` method, you specify the locations of the spikes (the points that the spline must go through) and the spline's tension. If the tension is 0, the spline is totally flexible, like a rubber band: All the segments between points are straight lines. The higher the tension, the smoother the curve will be. Figure 18.8 shows four curves passing through the same points, but each curve is drawn with a different tension value. The curves shown in the figure were drawn with the GDIPlus project (using the Ordinal Curves button).

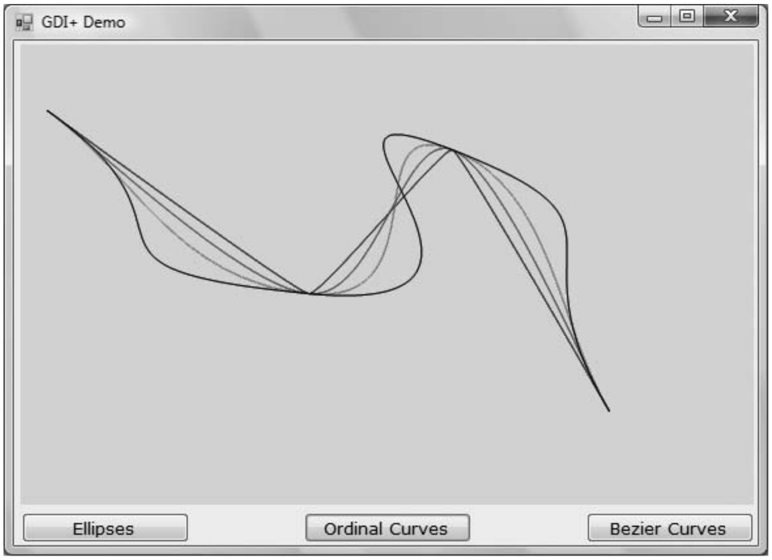
The simplest form of the `DrawCurve` method has the following syntax, where *`points`* is an array of points:

```
Graphics.DrawCurve(pen, points, tension)
```

The first and last elements of the array are the curve's endpoints, and the curve will go through the remaining points as well.

The curves shown in Figure 18.8 were produced by the code shown in Listing 18.7. Notice that a tension of 0.5 is practically the same as 0 (the spline bends around the fixed points like a rubber band). If you drew the same curve with a tension of 5, you'd get an odd curve indeed because although a physical spline would break, the mathematical spline takes an unusual shape to accommodate the fixed points.

FIGURE 18.8
These curves go through the same points, but they have different tensions.



LISTING 18.7: Curves with Common Fixed Points and Different Tensions

```
Private Sub bttnCurves_Click(...) Handles bttnCurves.Click
    Dim G As Graphics
    G = PictureBox1.CreateGraphics
    G.Clear(PictureBox1.BackColor)
    G.FillRectangle(Brushes.Silver, ClientRectangle)
    G.SmoothingMode = Drawing.Drawing2D.SmoothingMode.HighQuality
    Dim points() As Point = { _
           New Point(20, 50), New Point(220, 190), _
           New Point(330, 80), New Point(450, 280)}
    G.DrawCurve(Pens.Blue, points, 0.1)
    G.DrawCurve(Pens.Red, points, 0.5)
    G.DrawCurve(Pens.Green, points, 1)
    G.DrawCurve(Pens.Black, points, 2)
End Sub
```

DRAWBEZIER

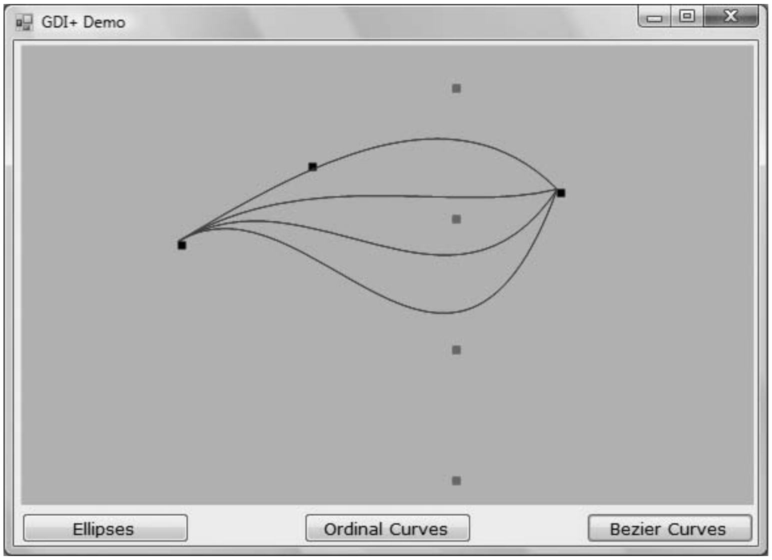
The `DrawBezier` method draws Bezier curves, which are smoother than cardinal splines. A Bezier curve is defined by two endpoints and two control points. The control points act as magnets. The curve is the trace of a point that starts at one of the endpoints and moves toward the second one. As it moves, the point is attracted by the two control points. Initially, the first control point's influence is predominant. Gradually, the curve comes into the second control point's field and it ends at the second endpoint.

The `DrawBezier` method accepts a pen and four points as arguments:

```
Graphics.DrawBexier(pen, X1, Y1, X2, Y2, X3, Y3, X4, Y4)
Graphics.DrawBezier(pen, point1, point2, point3, point4)
```

Figure 18.9 shows four Bezier curves, which differ in the y-coordinate of the third control point. All control points are marked with little squares: one each for the three points that are common to all curves, and four in a vertical column for the point that differs in each curve.

FIGURE 18.9
Bezier curves and their control points



The code of Listing 18.8 draws the four Bezier curves (I'm not showing the statements that draw the small rectangles; they simply call the `FillRectangle` method). The endpoints and one control point (P1, P2, and P4) remain the same, whereas the other control point (P3) is set to four different values. Notice how far the control point must go to have a significant effect on the curve's shape.

LISTING 18.8: Drawing Bezier Curves and Their Control Points

```
Private Sub bttnBezier_Click(...) Handles bttnBezier.Click
   Dim G As Graphics
   G = PictureBox1.CreateGraphics
   G.SmoothingMode = Drawing.Drawing2D.SmoothingMode.AntiAlias
   G.FillRectangle(Brushes.Silver, ClientRectangle)
   Dim P1 As New Point(120, 150)
   Dim P2 As New Point(220, 90)
   Dim P3 As New Point(330, 30)
   Dim P4 As New Point(410, 110)
   Dim sqrSize As New Size(6, 6)
   G.DrawBezier(Pens.Blue, P1, P2, P3, P4)
   P3 = New Point(330, 130)
```

```
        G.DrawBezier(Pens.Blue, P1, P2, P3, P4)
        P3 = New Point(330, 230)
        G.DrawBezier(Pens.Blue, P1, P2, P3, P4)
        P3 = New Point(330, 330)
        G.DrawBezier(Pens.Blue, P1, P2, P3, P4)
    End Sub
```

To draw the curve, all you need is to specify the four control points and pass them along with a Pen object to the `DrawBezier` method.

DrawPath

This method accepts a Pen object and a Path object as arguments and renders the specified path on the screen:

```
Graphics.DrawPath(pen, path)
```

To construct the Path object, use the `Add`*`XXX`* methods (`AddLine`, `AddRectangle`, and so on) — refer to the section called ''The Path Class,'' earlier in this chapter. You will find an example of how to use the Path object later in this chapter, when you'll learn how to plot functions.

DrawString, MeasureString

The `DrawString` method renders a string in a single line or multiple lines. As a reminder, the `TextRenderingHint` property of the Graphics object allows you to specify the quality of the rendered text. The simplest form of the `DrawString` method is the following:

```
Graphics.DrawString(string, font, brush, X, Y)
```

The first argument is the string to be rendered in the font specified by the second argument. The text will be rendered with the Brush object specified by the *`brush`* argument. *`X`* and *`Y`*, finally, are the coordinates of the top-left corner of a rectangle that completely encloses the string.

While working with strings, in most cases you need to know the actual dimensions of the string when rendered with the `DrawString` method in the specified font. The `MeasureString` method allows you to retrieve the metrics of a string before actually drawing it. This method returns a SizeF structure with the width and height of the string when rendered on the same Graphics object with the specified font. We'll use this method extensively in Chapter 20, ''Printing with Visual Basic 2008,'' to position text precisely on the printed page. You can also pass a Rectangle object as an argument to the `MeasureString` method to find out how many lines it will take to render the string on the rectangle.

The simplest form of the `MeasureString` method is the following, where *`string`* is the string to be rendered and *`font`* is the font in which the string will be rendered:

```
Dim textSize As SizeF
textSize = Me.Graphics.MeasureString(string, font)
```

To center a string on the form, use the x-coordinate returned by the `MeasureString` method, as in the following code segment:

```
Dim textSize As SizeF
Dim X As Integer, Y As Integer = 0
```

```
textSize = Me.Graphics.MeasureString(string, font)
X = (Me.Width - textSize.Width) / 2
G.DrawString("Centered string", font, brush, X, Y)
```

We subtract the rendered string's length from the form's width, and we split the difference in half at the two sides of the string.

Figure 18.10 shows a string printed at the center of the form and the two lines passing through the same point. Listing 18.9 shows the statements that produced the string. This listing is part of the TextEffects sample project.

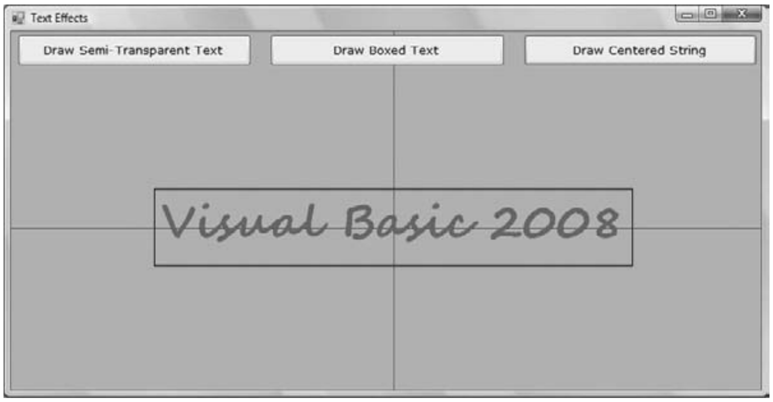


FIGURE 18.10
Centering a string on a form

LISTING 18.9: Printing a String Centered on the Form

```
Private Sub Center(...) Handles bttnCentered.Click
   Dim G As Graphics
   G = Me.CreateGraphics
   G.FillRectangle(New SolidBrush(Color.Silver), ClientRectangle)
   G.TextRenderingHint = Drawing.Text.TextRenderingHint.AntiAlias
   FontDialog1.Font = Me.Font
   FontDialog1.ShowDialog()
   Dim txtFont As Font
   txtFont = FontDialog1.Font
   G.DrawLine(New Pen(Color.Green), CInt(Me.Width / 2), CInt(0), _
               CInt(Me.Width / 2), CInt(Me.Height))
   G.DrawLine(New Pen(Color.Green), 0, CInt(Me.Height / 2), _
               CInt(Me.Width), CInt(Me.Height / 2))
   Dim txtLen, txtHeight As Integer
   Dim str As String = "Visual Basic 2008"
   Dim txtSize As SizeF
   txtSize = G.MeasureString(str, txtFont)
   Dim txtX, txtY As Integer
   txtX = (Me.Width - txtSize.Width) / 2
   txtY = (Me.Height - txtSize.Height) / 2
```

```
        G.DrawString(str, txtFont, _
                         New SolidBrush(Color.Red), txtX, txtY)
    Me.Invalidate ()
    End Sub
```

The coordinates passed to the `DrawString` method (variables *`txtX`* and *`txtY`*) are the coordinates of the top-left corner of the rectangle that encloses the first character of the string.

Another form of the `DrawString` method accepts a rectangle as an argument and draws the string in this rectangle, breaking the text into multiple lines if needed. The syntax of this form of the method is as follows:

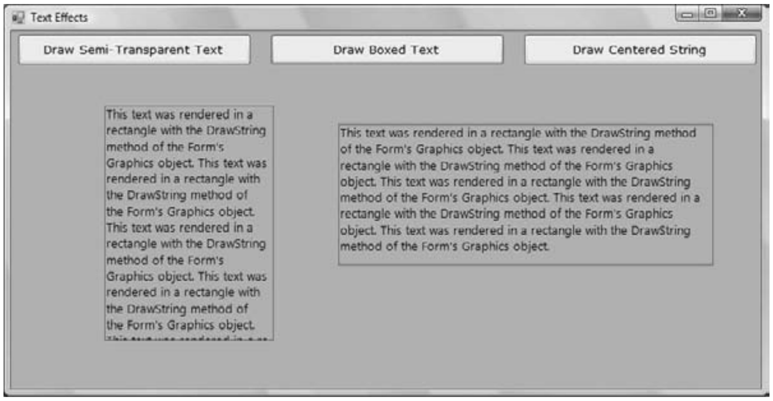
```
Graphics.DrawString(string, font, brush, rectanglef)
Graphics.DrawString(string, font, brush, rectanglef, stringFormat)
```

If you want to render text in a box, you will most likely use the equivalent form of the `MeasureString` method to retrieve the metrics of the text in the rectangle. This form of the `MeasureString` method returns the number of lines it will take to render the string in the supplied rectangle, and it has the following syntax, where *`string`* is the text to be rendered, and *`font`* is the font in which the string will be rendered:

```
e.Graphics.MeasureString(string, font, fitSize,
                              stringFormat, lines, cols)
```

The *`fitSize`* argument is a SizeF object that represents the width and height of a rectangle, where the string must fit. The *`lines`* and *`cols`* variables are passed by reference, and they are set by the `MeasureString` method to the number of lines and number of characters that will fit in the specified rectangle. The exact location of the rectangle doesn't make any difference — only its dimensions matter, and that's why the third argument is a SizeF object, not a Rectangle object.

Figure 18.11 shows a string printed in two different rectangles by the TextEffects sample project; the figure was created with the Draw Boxed Text button. The code that produced the figure is shown in Listing 18.10.



This text was rendered in a rectangle with the DrawString method of the Form's Graphics object. This text was rendered in a rectangle with the DrawString method of the Form's Graphics object. This text was rendered in a rectangle with the DrawString method of the Form's Graphics object. This text was rendered in a rectangle with the DrawString method of the Form's Graphics object.

This text was rendered in a rectangle with the DrawString method of the Form's Graphics object. This text was rendered in a rectangle with the DrawString method of the Form's Graphics object. This text was rendered in a rectangle with the DrawString method of the Form's Graphics object. This text was rendered in a rectangle with the DrawString method of the Form's Graphics object. This text was rendered in a rectangle with the DrawString method of the Form's Graphics object.

FIGURE 18.11 Printing text in a rectangle

Listing 18.10: Printing Text in a Rectangle

```
Private Sub BoxedText(...) Handles bttnBoxed.Click
    Dim G As Graphics
    G = GetGraphicsObject()
    G.FillRectangle(New SolidBrush(Color.Silver), ClientRectangle)
    FontDialog1.Font = Me.Font
    FontDialog1.ShowDialog()
    Dim txtFont As Font
    txtFont = FontDialog1.Font
    Dim txt As String = "This text was rendered in a rectangle " & _
                        "with the DrawString method of the Form's " & _
                        "Graphics object. "
    txt = txt & txt & txt & txt & txt
    G.DrawString(txt, txtFont, Brushes.Black, _
                 New RectangleF(100, 80, 180, 250))
    G.DrawRectangle(Pens.Red, 100, 80, 180, 250)
    G.DrawString(txt, txtFont, Brushes.Black, _
                 New RectangleF(350, 100, 400, 150))
    G.DrawRectangle(Pens.Red, 350, 100, 400, 150)
    Me.Invalidate()
End Sub
```

The StringFormat Object

Some of the overloaded forms of the `DrawString` method accept an argument of the `StringFormat` type. This argument determines characteristics of the text and exposes a few properties of its own, which include the following:

Alignment Determines the alignment of the text; its value is a member of the `StringAlignment` enumeration: *`Center`* (text is aligned in the center of the layout rectangle), *`Far`* (text is aligned far from the origin of the layout rectangle), and *`Near`* (text is aligned near the origin of the layout rectangle).

Trimming Determines how text will be trimmed if it doesn't fit in the layout rectangle. Its value is one of the members of the `StringTrimming` enumeration: *`Character`* (text is trimmed to the nearest character), *`EllipsisCharacter`* (text is trimmed to the nearest character and an ellipsis is inserted at the end to indicate that some of the text is missing), *`EllipsisPath`* (text at the middle of the string is removed and replaced by an ellipsis), *`EllipsisWord`* (text is trimmed to the nearest word and an ellipsis is inserted at the end), *`None`* (no trimming), and *`Word`* (text is trimmed to the nearest word).

FormatFlags Specifies layout information for the string. Its value can be one of the members of the `StringFormatFlags` enumeration. The two members of this enumeration that you might need often are *`DirectionRightToLeft`* (prints to the left of the specified point) and *`DirectionVertical`*.

To use the *stringFormat* argument of the DrawString method, instantiate a variable of this type, set the desired properties, and then pass it as an argument to the DrawString method, as shown here:

```
Dim G As Graphics = Me.CreateGraphics
Dim SF As New StringFormat()
SF.FormatFlags = StringFormatFlags.DirectionVertical
G.DrawString("Visual Basic", Me.Font, Brushes.Red, 80, 80, SF)
```

The call to the DrawString method will print the string from top to bottom. It will also rotate the characters. The *DirectionRightToLeft* setting will cause the DrawString method to print the string to the left of the specified point, but it will not mirror the characters.

You can find additional examples of the MeasureString method in Chapter 20, in which we'll use this method to fit strings on the width of the page. The third button on the form of the TextEffects project draws text with a three-dimensional look by overlaying a semitransparent string over an opaque string. This technique is explained in the ''Alpha Blending'' section in Chapter 19, in which you'll learn how to use transparency. You might also wonder why none of the DrawString methods' forms accept as an argument an angle of rotation for the text. You can draw text or any shape at any orientation as long as you set up the proper rotation transformation. This topic is discussed in the ''Applying Transformations'' section later in this chapter, as well as in Chapter 20.

DrawImage

The DrawImage method, which renders an image on the Graphics object, is a heavily overloaded and quite flexible method. The following form of the method draws the image at the specified location. Both the image and the location of its top-left corner are passed to the method as arguments (as Image and Point arguments, respectively):

```
Graphics.DrawImage(img, point)
```

Another form of the method draws the specified image within a rectangle. If the rectangle doesn't match the original dimensions of the image, the image will be stretched to fit in the rectangle. The rectangle should have the same aspect ratio as the Image object, to avoid distorting the image in the process.

```
Graphics.DrawImage(img, rectangle)
```

Another form of the method allows you to change not only the magnification of the image, but also its shape. This method accepts as an argument not a rectangle, but an array of three points that specifies a parallelogram. The image will be sheared to fit in the parallelogram, where *points* is an array of points that define a parallelogram:

```
Graphics.DrawImage(img, points())
```

The array holds three points, which are the top-left, top-right, and bottom-left corners of the parallelogram. The fourth point is determined uniquely by the other three, and you need not supply it. The ImageCube sample project, shown later in this chapter, uses this overloaded form of the `DrawImage` method to draw a cube with a different image on each face.

Another interesting form of the method allows you to set the attributes of the image:

```
Graphics.DrawImage(image, points(), srcRect, units, attributes)
```

The first two arguments are the same as in the previous forms of the method. The *`srcRect`* argument is a rectangle that specifies the portion of image to draw, and *`units`* is a constant of the `GraphicsUnit` enumeration. It determines how the units of the rectangle are measured (pixels, inches, and so on). The last argument is an ImageAttributes object that contains information about the attributes of the image you want to change (such as the gamma value, and a transparent color value or color key). The properties of the ImageAttributes class are discussed shortly.

The `DrawImage` method is quite flexible, and you can use it for many special effects, including wipes. A *wipe* is the gradual appearance of an image on a form or PictureBox control. You can use this method to draw stripes of the original image, or start with a small rectangle in the middle that grows gradually until it covers the entire image.

You can also correct the color of the image by specifying the *`attributes`* argument. To specify the *`attributes`* argument, create an ImageAttributes object with a statement like the following:

```
Dim attr As New System.Drawing.Imaging.ImageAttributes
```

Then call one or more of the ImageAttributes class's methods:

SetWrapMode Specifies the wrap mode that is used to decide how to tile a texture across a shape. This attribute is used with textured brushes (a topic that isn't discussed in this book).

SetGamma This method sets the gamma value for the image's colors and accepts a Single value, which is the gamma value to be applied. A gamma value of 1 doesn't affect the colors of the image. A smaller value darkens the colors, whereas a larger value makes the image colors brighter. Notice that the gamma correction isn't the same as manipulating the brightness of the colors. The gamma correction takes into consideration the entire range of values in the image; it doesn't apply equally to all the colors. In effect, it takes into consideration both the brightness and the contrast and corrects them in tandem with a fairly complicated algorithm. The syntax of the `SetGamma` method is as follows:

```
ImageAttributes.SetGamma(gamma)
```

The following statements render the image stored in the `img` Image object on the *`G`* Graphics object, and they gamma-correct the image in the process by a factor of 1.25:

```
Dim attrs As New System.Drawing.Imaging.ImageAttributes()
attrs.SetGamma(1.25)
Dim dest As New Rectangle(0, 0, PictureBox1.Width, PictureBox1.Height)
G.DrawImage(img, dest, 0, 0, img.Width, img.Height, _
                                         GraphicsUnit.Pixel,   attrs)
```

Gradients

In this section, you'll look at the tools for creating gradients. The techniques for gradients can get quite complicated, but I will limit the discussion to the types of gradients you'll need for business or simple graphics applications.

LINEAR GRADIENTS

Let's start with linear gradients. Like all other gradients, they're part of the System.Drawing class and are implemented as brushes. To draw a linear gradient, you must create an instance of the LinearGradientBrush class with a statement like the following:

```
Dim lgBrush As LinearGradientBrush
lgBrush = New LinearGradientBrush(rect, startColor, endColor, gradientMode)
```

To understand how to use the arguments, you must understand how the linear gradient works. This method creates a gradient that fills a rectangle, specified by the *rect* object passed as the first argument. This rectangle isn't filled with any gradient; it simply tells the method how long (or how tall) the gradient should be. The gradient starts with the *startColor* at the left side of the rectangle and ends with the *endColor* at the opposite side. The gradient changes color slowly as it moves from one end to the other. The last argument, *gradientMode*, specifies the direction of the gradient and can have one of the values shown in Table 18.5.

TABLE 18.5: The *LinearGradientMode* Enumeration

VALUE	EFFECT
BackwardDiagonal	The gradient fills the rectangle diagonally from the top-right corner (*startColor*) to the bottom-left corner (*endColor*).
ForwardDiagonal	The gradient fills the rectangle diagonally from the top-left corner (*startColor*) to the bottom-right corner (*endColor*).
Horizontal	The gradient fills the rectangle from left (*startColor*) to right (*endColor*).
Vertical	The gradient fills the rectangle from top (*startColor*) to bottom (*endColor*).

Notice that in the descriptions of the various modes in the table, I state that the gradient fills the rectangle, not the shape. The gradient is calculated according to the dimensions of the rectangle specified with the first argument. If the actual shape is smaller than this rectangle, only a section of the gradient will be used to fill the shape. If the shape is larger than this rectangle, the gradient will repeat as many times as necessary to fill the shape. We usually fill a shape that's as wide (or as tall) as the rectangle used to specify the gradient.

Let's say you want to use the same gradient that extends 300 pixels horizontally to fill two rectangles: one that's 200 pixels wide and another that's 600 pixels wide. The first rectangle, which is 200 pixels wide, will be filled with two thirds of the gradient; the second rectangle, which is 600 pixels wide, will be filled with a gradient that's repeated twice. The code in Listing 18.11 corresponds to the Linear Gradient button of the Gradients project.

LISTING 18.11: Filling Rectangles with a Linear Gradient

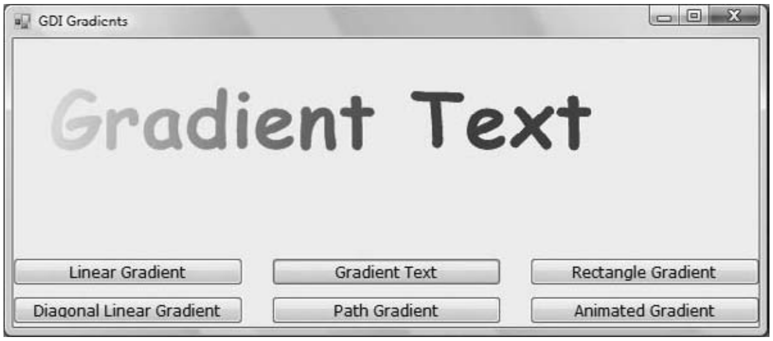
```
Private Sub LinearGradient_Click(...) Handles bttnLinearGradient.Click
   Dim G As Graphics
   G = Me.CreateGraphics
   Dim R As New RectangleF(20, 20, 300, 100)
   Dim startColor As Color = Color.BlueViolet
   Dim EndColor As Color = Color.LightYellow
   Dim LGBrush As New System.Drawing.Drawing2D.LinearGradientBrush _
        (R, startColor, EndColor, LinearGradientMode.Horizontal)
   G.FillRectangle(LGBrush, New Rectangle(20, 20, 200, 100))
   G.FillRectangle(LGBrush, New Rectangle(20, 150, 600, 100))
End Sub
```

For a horizontal gradient, only the width of the rectangle is used; the height is irrelevant. For a vertical gradient, only the height of the rectangle matters. When you draw a diagonal gradient, both dimensions are taken into consideration.

You can create gradients at various directions by setting the `gradientMode` argument of the LinearGradientBrush object's constructor. The Diagonal Linear Gradient button on the Gradients project does exactly that.

The button Gradient Text on the form of the Gradients project renders some text filled with a linear gradient. As you recall from our discussion of the `DrawString` method, strings are rendered with a Brush object, not a Pen object. If you specify a LinearGradientBrush object, the text will be rendered with a linear gradient. The text shown in Figure 18.12 was produced by the Gradient Text button, whose code is shown in Listing 18.12.

FIGURE 18.12
Drawing a string filled with a gradient



LISTING 18.12: Rendering Strings with a Linear Gradient

```
Private Sub bttnGradientText_Click(...) Handles bttnGradientText.Click
   Dim G As Graphics
   G = Me.CreateGraphics
   G.Clear (me.BackColor)
   G.TextRenderingHint = System.Drawing.Text.TextRenderingHint.AntiAlias
```

```
        Dim largeFont As New Font( _
              "Comic Sans MS", 48, FontStyle.Bold, GraphicsUnit.Point)
        Dim gradientStart As New PointF(0, 0)
        Dim txt As String = "Gradient Text"
        Dim txtSize As New SizeF()
        txtSize = G.MeasureString(txt, largeFont)
        Dim gradientEnd As New PointF()
        gradientEnd.X = txtSize.Width
        gradientEnd.Y = txtSize.Height
        Dim grBrush As New LinearGradientBrush(gradientStart, gradientEnd, _
                            Color.Yellow, Color.Blue)
        G.DrawString(txt, largeFont, grBrush, 20, 20)
    End Sub
```

The code of Listing 18.12 is a little longer than it could be (or than you might expect). Because linear gradients have a fixed size and don't expand or shrink to fill the shape, you must call the `MeasureString` method to calculate the width of the string and then create a linear gradient with the exact same width. This way, the gradient's extent matches that of the string.

Path Gradients

This is the ultimate gradient tool. Using a PathGradientBrush, you can create a gradient that starts at a single point and fades into multiple different colors in different directions. You can fill a rectangle starting from a point in the interior of the rectangle, which is colored, say, black. Each corner of the rectangle might have a different ending color. The PathGradientBrush will change color in the interior of the shape and will generate a gradient that's smooth in all directions. Figure 18.13 shows a rectangle filled with a path gradient, although the gray shades on the printed page won't show the full impact of the gradient. Open the Gradients project to see the same figure in color (use the Path Gradient button).

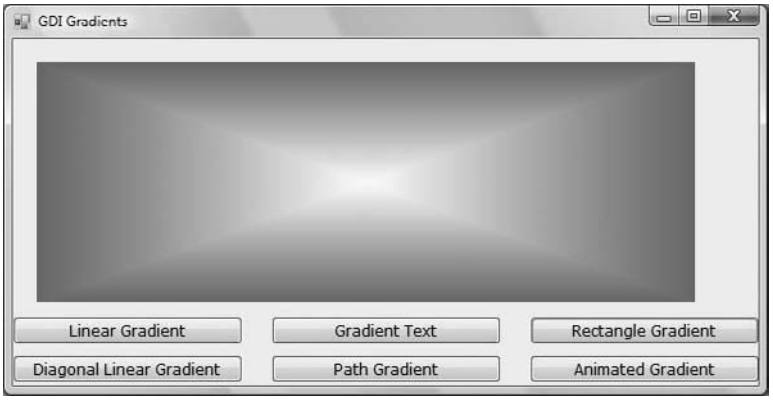


Figure 18.13
A path gradient starting at the middle of the rectangle

To fill a shape with a path gradient, you must first create a Path object. The PathGradientBrush will be created for the specific path and can be used to fill this path — but not any other shape. Actually, you can fill any other shape with the PathGradientBrush created for a specific path, but the gradient won't fit the new shape. To create a PathGradientBrush, use the following syntax, where *path* is a properly initialized Path object:

```
Dim pgBrush As New PathGradientBrush(path)
```

The *pgBrush* object provides properties that determine the exact coloring of the gradient. First, you must specify the color of the gradient at the center of the shape by using the `CenterColor` property. The `SurroundColors` property is an array with as many elements as there are vertices (corners) in the Path object. Each element of the `SurroundColors` array must be set to a color value, and the resulting gradient will have the color of the equivalent element of the `Surround-Colors` array.

The following declaration creates an array of three different colors and assigns the colors to the `SurroundColors` property of a PathGradientBrush object:

```
Dim Colors() As Color = {Color.Yellow, Color.Green, Color.Blue}
pgBrush.SurroundColors = Colors
```

After setting the PathGradientBrush, you can fill the corresponding Path object by calling the `FillPath` method. The Path Gradient button on the Gradient application's main form creates a rectangle filled with a gradient that's red in the middle of the rectangle and has a different color at each corner. Listing 18.13 shows the code behind the Path Gradient button.

LISTING 18.13: Filling a Rectangle with a Path Gradient

```
Private Sub bttnPathGradient_Click(...) Handles bttnPathGradient.Click
   Dim G As Graphics
   G = Me.CreateGraphics
   Dim path As New System.Drawing.Drawing2D.GraphicsPath()
   path.AddLine(New Point(10, 10), New Point(400, 10))
   path.AddLine(New Point(400, 10), New Point(400, 250))
   path.AddLine(New Point(400, 250), New Point(10, 250))
   Dim pathBrush As New System.Drawing.Drawing2D.PathGradientBrush(path)
   pathBrush.CenterColor = Color.Red
   Dim surroundColors() As Color = _
            {Color.Yellow, Color.Green, Color.Blue, Color.Cyan}
   pathBrush.SurroundColors = surroundColors
   G.FillPath(pathBrush, path)
End Sub
```

The gradient's center point is, by default, the center of the shape. You can also specify the center of the gradient (the point that will be colored according to the `CenterColor` property). You can place the center point of the gradient anywhere by setting its `CenterPoint` property to a Point or PointF value.

The Gradients application has a few more buttons that create interesting gradients, which you can examine on your own. The Rectangle Gradient button fills a rectangle with a gradient that has a single ending color all around. All the elements of the `SurroundColors` property are set to the same color. The Animated Gradient animates the same gradient by changing the coordinates of the PathGradientBrush object's `CenterPoint` property slowly over time.

Clipping

Anyone who has used drawing or image-processing applications already knows that many of the application's tools use masks. A *mask* is any shape that limits the area in which you can draw. If you want to place a star or heart on an image and print something in it, you create the shape in which you want to limit your drawing tools and then you convert this shape into a mask. When you draw with the mask, you can start and end your strokes anywhere on the image. Your actions will have no effect outside of the mask, however.

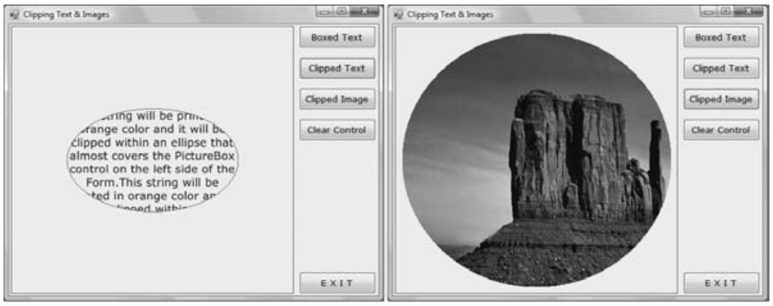
The mask of the various image-processing applications is a *clipping region*, which can be anything, as long as it's a closed shape. While the clipping region is activated, drawing takes place in the area of the clipping region. To specify a clipping region, you must call the `SetClip` method of the Graphics object. The `SetClip` method accepts the clipping area as an argument, and the clipping area can be the Graphics object itself (no clipping), a Rectangle, a Path, or a Region. A *region* is a structure made up of simple shapes, just like a path. There are many methods for creating a Region object — you can combine and intersect shapes, or exclude shapes from a region — but we aren't going to discuss the Region object in this chapter because it's not among the common objects we use to generate the type of graphics discussed in the context of this book.

The `SetClip` method has the following forms:

```
Graphics.SetClip(Graphics)
Graphics.SetClip(Rectangle)
Graphics.SetClip(GraphicsPath)
Graphics.SetClip(Region)
```

All methods accept a second optional argument, which determines how the new clipping area will be combined with the existing one. The `combineMode` argument's value is one of the members of the `CombineMode` enumeration: *`Complement`*, *`Exclude`*, *`Intersect`*, *`Replace`*, *`Union`*, and *`XOR`*.

After a clipping area has been set for the Graphics object, drawing is limited to that area. You can specify any coordinates, but only the part of the drawing that falls inside the clipping area is visible. The Clipping project demonstrates how to clip text and images within an elliptical area (see Figure 18.14). The Boxed Text button draws a string in a rectangle. The Clipped Text button draws the same text but first applies a clipping area, which is an ellipse. The Clipped Image button uses the same ellipse to clip an image. Because there's no form of the `SetClip` method that accepts an ellipse as an argument, we must construct a Path object, add the ellipse to the path, and then create a clipping area based on the path.



```
string will be prin
orange color and it will be
clipped within an ellipse that
almost covers the PictureBox
control on the left side of the
Form.This string will be
ted in orange color an
lipped within
```

FIGURE 18.14
Clipping text (left) and images (right) in an ellipse

The following statements create the clipping area for the text, which is an ellipse. The path is created by calling the `AddEllipse` method of the GraphicsPath object. This path is then passed as an argument to the Graphics object's `SetClip` method:

```
Dim P As New System.Drawing.Drawing2D.GraphicsPath()
Dim clipRect As New RectangleF(30, 30, 250, 150)
P.AddEllipse(clipRect)
Dim G As Graphics
G = PictureBox1.CreateGraphics
G.SetClip(P)
```

Listing 18.14 shows the code behind the Boxed Text and Clipped Text buttons. The Boxed Text button prints some text in a rectangular area that is centered over the clipping area. The Clipped Text button shows how the text is printed within the rectangle. Both the rectangle and the ellipse are based on the same Rectangle object.

Listing 18.14: The Boxed Text and Clipped Text Buttons

```
Private Sub bttnBoxedText_Click(...) Handles bttnBoxedText.Click
    Dim G As Graphics
    G = GetGraphicsObject()
    Dim Rect As New Rectangle( _
              Convert.ToInt32((PictureBox1.Width - 250) / 2), _
              Convert.ToInt32((PictureBox1.Height - 150) / 2), 250, 150)
    G.ResetTransform()
    G.ResetClip()
    Dim format As StringFormat = New StringFormat()
    format.Alignment = StringAlignment.Center
    G.TextRenderingHint = Drawing.Text.TextRenderingHint.AntiAlias
    G.DrawString(txt & txt, _
          New Font("Verdana", 12, FontStyle.Regular), _
          Brushes.DarkGreen, Rect, format)
    G.DrawRectangle(Pens.Yellow, Rect)
    PictureBox1.Invalidate()
End Sub

Private Sub bttnClippedText_Click(...) Handles bttnClippedText.Click
    Dim G As Graphics
    G = GetGraphicsObject()
    Dim P As New System.Drawing.Drawing2D.GraphicsPath()
    Dim clipRect As New RectangleF( _
              Convert.ToSingle((PictureBox1.Width - 250) / 2), _
              Convert.ToSingle((PictureBox1.Height - 150) / 2), 250, 150)
    P.AddEllipse(clipRect)
    G.ResetTransform()
    G.DrawEllipse(Pens.Red, clipRect)
    G.SetClip(P)
    Dim format As StringFormat = New StringFormat()
```

```
        format.Alignment = StringAlignment.Center
        G.TextRenderingHint = Drawing.Text.TextRenderingHint.AntiAlias
        G.DrawString(txt & txt, _
              New Font("Verdana", 12, FontStyle.Regular), _
              Brushes.DarkBlue, clipRect, format)
        PictureBox1.Invalidate()
    End Sub
```

The difference between the two subroutines is that the second sets an ellipse as the clipping area; anything we draw on it is automatically clipped.

The Clipped Image button sets up a similar clipping area and then draws an image centered behind the clipping ellipse. As you can see in Figure 18.14, only the segment of the image that's inside the clipping area is visible. The code behind the Clipped Image button is shown in Listing 18.15.

LISTING 18.15: The Clipped Image Button

```
    Private Sub bttnClippedImage_Click(...) Handles bttnClippedImage.Click
       Dim G As Graphics
       G = CreateGraphicsObject
       G.ResetClip()
       Dim P As New System.Drawing.Drawing2D.GraphicsPath()
       Dim clipRect As New RectangleF(10, 10,  _
                    PictureBox1.Width - 20), PictureBox1.Height - 20)
       P.AddEllipse(clipRect)
       G.SetClip(P)
       G.DrawImage(Image.FromFile(fileName), -150, -150)
       PictureBox1.Invlidate()
    End Sub
```

An easy and interesting technique for creating paths is to use the `AddString` method of the GraphicsPath object. Then you can draw an image over this path. The net effect is seeing sections of the image through the string's characters. You can open the StringPath sample project to see how you can clip an image with text; just be sure you select an interesting image to show through the string's characters.

Applying Transformations

In computer graphics, there are three types of transformations: scaling, translation, and rotation:

- The *scaling transformation* changes the dimensions of a shape but not its basic form. If you scale an ellipse by 0.5, you'll get another ellipse that's half as wide and half as tall as the original one.
- The *translation transformation* moves a shape by a specified distance. If you translate a rectangle by 30 pixels along the x-axis and 90 pixels along the y-axis, the new origin will be 30 pixels to the right and 90 pixels down from the original rectangle's top-left corner.

- The *rotation transformation* rotates a shape by a specified angle, expressed in degrees; 360 degrees correspond to a full rotation, and the shape appears the same. A rotation by 180 degrees is equivalent to flipping the shape vertically and horizontally.

Transformations are stored in a 5 × 5 matrix, but you need not set it up yourself. The Graphics object provides the `ScaleTransform`, `TranslateTransform`, and `RotateTransform` methods, and you can specify the transformation to be applied to the shape by calling one or more of these methods and passing the appropriate argument(s). The `ScaleTransform` method accepts as arguments scaling factors for the horizontal and vertical directions:

```
Graphics.ScaleTransformation(Sx, Sy)
```

If an argument is smaller than one, the shape will be reduced in the corresponding direction; if it's larger than one, the shape will be enlarged in the corresponding direction. We usually scale both directions by the same factor to retain the shape's aspect ratio. If you scale a circle by different factors in the two dimensions, the result will be an ellipse, and not a smaller or larger circle.

The `TranslateTransform` method accepts two arguments, which are the displacements along the horizontal and vertical directions:

```
Graphics.TranslateTransform(Tx, Ty)
```

The *`Tx`* and *`Ty`* arguments are expressed in the coordinates of the current coordinate system. The shape is moved to the right by *`Tx`* units and down by *`Ty`* units. If one of the arguments is negative, the shape is moved in the opposite direction (to the left or up).

The `RotateTransform` method accepts a single argument, which is the angle of rotation expressed in degrees:

```
Graphics.RotateTransform(rotation)
```

The rotation takes place about the origin. As you will see, the final position and orientation of a shape is different if two identical rotation and translation transformations are applied in a different order.

Every time you call one of these methods, the elements of the transformation matrix are set accordingly. All transformations are stored in this matrix, and they have a cumulative effect. If you specify two translation transformations, for example, the shape will be translated by the sum of the corresponding arguments in either direction. These two transformations:

```
Graphics.TranslateTransform(10, 40)
Graphics.TranslateTransform(20, 20)
```

are equivalent to the following one:

```
Graphics.TranslateTransform(30, 60)
```

To start a new transformation after drawing some shapes on the Graphics object, call the `Reset-Transform` method, which clears the transformation matrix.

The effect of multiple transformations might be cumulative, but the order in which transformations are performed makes a big difference. You will find some real-world examples of transformations in Chapter 20, where I discuss printing with Visual Basic. In specific, you'll see how to apply transformations to print rotated strings on a page. I've also included the Transformations sample project in this chapter. This project allows you to apply transformations to an entity that consists of a rectangle that contains a string and a small bitmap, as shown in Figure 18.15. Each button on the right performs a different transformation or combination of transformations. The code is quite short, and you can easily insert additional transformations or change their order, and see how the shape is transformed. Keep in mind that some transformations might bring the shape entirely outside the form. In this case, just apply a translation transformation in the opposite direction.

The code behind the Translate Shape, Rotate Shape, and Scale Shape buttons is shown in Listing 18.16. The code in the `Click` event handlers of the buttons sets the appropriate transformations and then calls the `DrawShape()` subroutine, passing the current Graphics object as an argument. The `DrawShape()` subroutine draws the same shape, but its actual output (the position and size of the shape) is affected by the transformation matrix in effect.

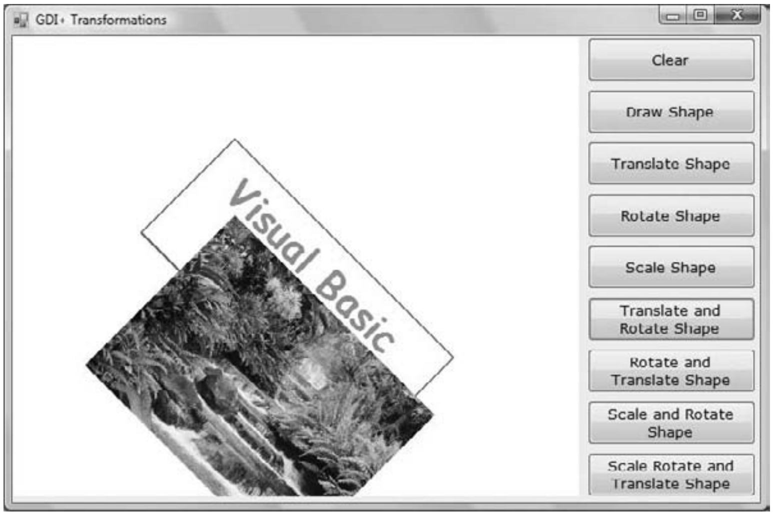


FIGURE 18.15 The Transformations project

LISTING 18.16: The Buttons of the GDIPlusTransformations Project

```
Private Sub bttnTranslate_Click(...) Handles bttnTranslate.Click
   Dim G As Graphics = PictureBox1.CreateGraphics
   G.TranslateTransform(200, 90)
   DrawShape(G)
End Sub

Private Sub bttnRotate_Click(...) Handles bttnRotate.Click
   Dim G As Graphics = PictureBox1.CreateGraphics
   G.RotateTransform(45)
   DrawShape(G)
End Sub
```

```
Private Sub bttnTranslateRotate_Click(...) _
            Handles bttnTranslateRotate.Click
   Dim G As Graphics = PictureBox1.CreateGraphics
   G.TranslateTransform(200, 90)
   G.RotateTransform(45)
   DrawShape(G)
End Sub
```

VB 2008 at Work: The ImageCube Project

As discussed earlier in this chapter, the `DrawImage` method can render images on any parallelogram, not just a rectangle, with the necessary distortion. A way to look at these images is not as distorted images, but as *perspective* images. Looking at a printout from an unusual angle is equivalent to rendering an image within a parallelogram. Imagine a cube with a different image glued on each side. To display such a cube on your monitor, you must calculate the coordinates of the cube's edges and then use these coordinates to define the parallelograms on which each image will be displayed. Figure 18.16 shows a cube with a different image on each side.

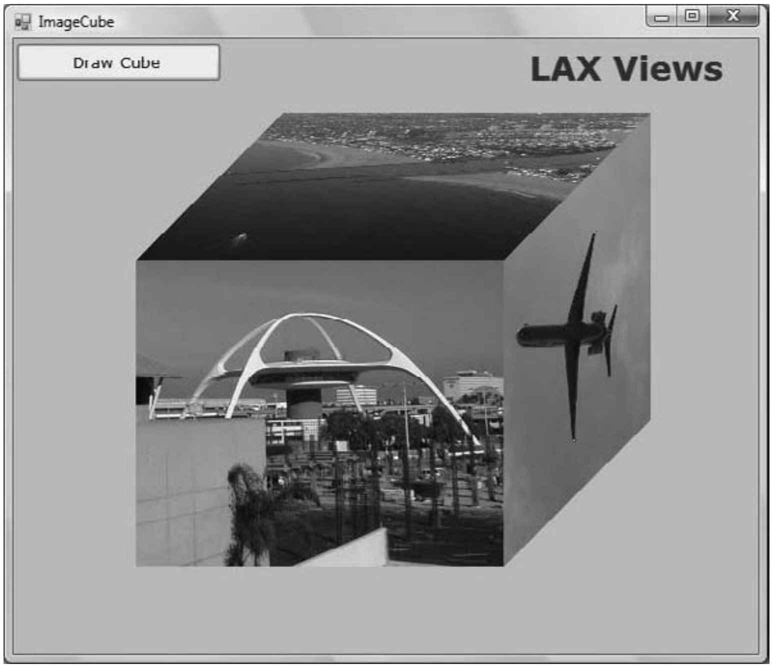


FIGURE 18.16 This cube was created with a call to the `DrawImage` method for each visible face of the cube.

If you're good at math, you can rotate a cube around its vertical and horizontal axes and then map the rotated cube on the drawing surface. You can even apply a perspective transformation, which will make the image look more like the rendering of a three-dimensional cube. This process is more involved than the topics discussed in this book. Instead of doing all the calculations, I came up with a set of coordinates for the parallelogram that represents each vertex (corner) of the cube. For a different orientation, you can draw a perspective view of a cube on paper and measure the coordinates of its vertices. After you define the parallelogram that corresponds to each visible side, you can draw an image on each face by using the `DrawImage` method. The `DrawImage` method will shear the image as necessary to fill the specified area. The result is a 3D-looking cube covered with images. You can open the sample project and examine its code, which contains comments to help you understand how it works.

VB 2008 at Work: Plotting Functions

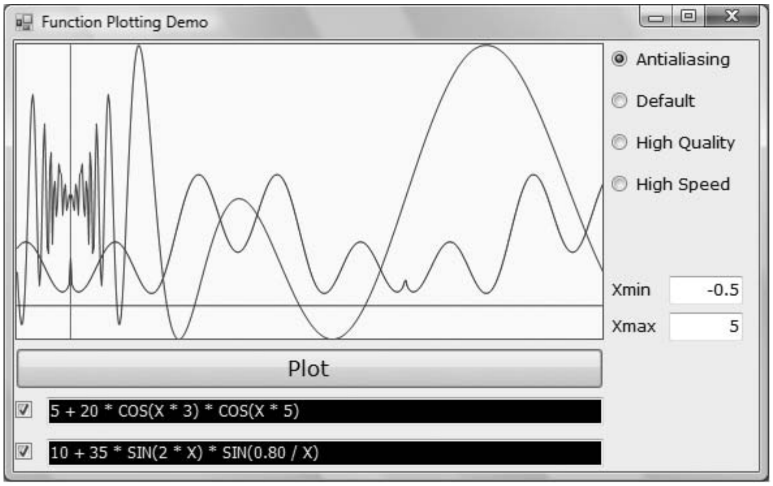
In this last section of this chapter, I address a fairly common task in scientific programming: the plotting of functions or user-supplied data sets. If you have no use for such an application, you can skip this section. I decided to include this application because many readers (especially college students) might use it as a starting point for developing a custom plotting application.

A *plot* is a visual representation of a function's values over a range of an independent variable. Figure 18.17 shows the following function plotted against time in the range from −0.5 to 5:

```
10 + 35 * Sin(2 * X) * Sin(0.80 / X)
```

The plot of Figure 18.17 was created with the FunctionPlotting project. The variable `x` represents time and goes from −0.5 to 5. The time is mapped to the horizontal axis, and the vertical axis is the magnitude of the function. For each pixel along the horizontal axis, we calculate the value of the function and turn on the pixel that corresponds to the calculated value.

FIGURE 18.17
Plotting math functions with Visual Basic



Xmin -0.5
Xmax 5

5 + 20 * COS(X * 3) * COS(X * 5)
10 + 35 * SIN(2 * X) * SIN(0.80 / X)

The application's code is fairly lengthy, and I will not show it in the printed version of this book. You will find a detailed discussion of the FunctionPlotting application in the `Readme.doc` file in the application's folder. The application's code is also well documented, and you should be able to easily follow it and adjust it to suit your custom requirements.

The Bottom Line

Display and size images. The most appropriate control for displaying images is the PictureBox control. You can assign an image to the control through its `Image` property, either at design time or at runtime. To display a user-supplied image at runtime, call the `DrawImage` method of the control's Graphics object.

> **Master It** How would you implement a form that displays a large image and allows users to scroll the image to bring any segment of it into view?

Generate graphics by using the drawing methods. Every object you draw on, such as forms and PictureBox controls, exposes the `CreateGraphics` method, which returns a Graphics object. The `Paint` event's `e` argument also exposes the Graphics object of the control or form. To draw something on a control, retrieve its Graphics object and then call the Graphics object's drawing methods.

> **Master It** Show how to draw a circle on a form from within the form's `Paint` event handler.

Display text in various ways, including gradient fills. The Graphics object provides the `DrawString` method, which prints a user-supplied string on a control. You can also specify the coordinates of the string's upper-left corner and its font. To position the string, you need to know its dimensions. You can use the `MeasureString` method to retrieve the dimensions of the image when rendered on the Graphics object in a specific font. Text is drawn with a Brush object, and you can use a SolidBrush object to draw the string in a solid color, the Linear-GradientBrush object to fill the text with a linear gradient, the PathGradientBrush object to fill the text with an arbitrary gradient defined by a path, or the TextureBush object to fill the text with
a texture.

> **Master It** How will you print a string centered on a PictureBox control?

Chapter 19

Manipulating Images and Bitmaps

The graphics you explored in Chapter 18, ''Drawing and Painting with Visual Basic 2008,'' are called *vector graphics* because they're based on geometric descriptions and they can be scaled to any extent. Because they're based on mathematical equations, you can draw any details of the picture without losing any accuracy. You can zoom into a tiny section of an ellipse, for example, and never lose any detail because the ellipse is redrawn every time.

Vector graphics, however, can't be used to describe the type of images you capture with your digital camera. These images belong to a different category of graphics: *bitmap graphics* or *raster graphics*. A bitmap is a collection of colored pixels arranged in rows and columns. As you will see, a *bitmap* is nothing more than a two-dimensional array of integers that represent colors, and you can achieve interesting effects with simple arithmetic operations on the pixels of an image.

In this chapter, you'll learn how to do the following:

- Specify colors
- Manipulate images and bitmaps
- Process images

Specifying Colors

You're already familiar with the Color common dialog box, which lets you specify colors by manipulating their basic components. To specify a Color value through this dialog box, you'll see three boxes — Red, Green, and Blue (RGB) — whose values change as you move the cross-shaped pointer over the color spectrum. These are the values of the three basic components that computers use to specify colors. Any color that can be represented on a computer monitor is specified by means of these three colors. By mixing percentages of these basic colors, you can design almost any color in the spectrum.

The model of designing colors based on the intensities of their *RGB* components is called the RGB model, and it's a fundamental concept in computer graphics. If you aren't familiar with this model, this section is well worth reading. Nearly every color you can imagine can be constructed by mixing the appropriate percentages of the three basic colors. Each color, therefore, is represented by a triplet of byte values that represent the basic color components of red, green, and blue. The smallest value, 0, indicates the absence of the corresponding color. The largest value, 255, indicates full intensity, or saturation. The triplet (0, 0, 0) is black because all colors are missing, and the triplet (255, 255, 255) is white — it contains all three basic colors in full intensity. Other colors have various combinations: (255, 0, 0) is a pure red tone, (0, 255, 255) is a pure cyan tone (what you get when you mix green and blue), and (0, 128, 128) is a mid-cyan tone (a mix of mid-green and mid-blue tones). The possible combinations of the three basic color components are 256 × 256 × 256, or 16,777,216 colors. Graphics cards that can display all 16 million colors are said to have a color depth of 24 bits (3 bytes). Most graphics cards today support a color depth of 32 bits: 24 bits for color and 8 bits for a transparency layer. (The topic of transparency is discussed in the ''Alpha Blending'' section later in this chapter.)

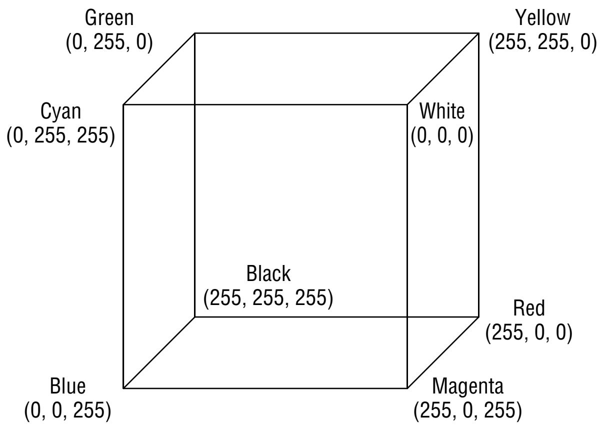
Notice that we use the term *basic colors*, not *primary colors*, which are the three colors used in designing colors with paint. The concept is the same: You mix these colors until you get the desired result. The primary colors used in painting, however, are different. They are the colors red, yellow, and blue. Painters can create any shade imaginable by mixing the appropriate percentages of red, yellow, and blue paint. On a computer monitor, you can design any color by mixing the appropriate percentages of red, green, and blue.

There are other color specification models besides the RGB model. Modern color printers use four primary colors: cyan, magenta, yellow, and black (the CMYK model). The color specification model used by computers is called *additive* (you must add all three basic colors to get white on your monitor). The color specification model used by printers, on the other hand, is called *subtractive* (absence of all colors gives white, which is the color of the paper on which an image is printed). For more information on the various color-specification models, see the *additive color* and *subtractive color* entries in Wikipedia. In this chapter, we'll focus on the RGB color model, which is used to specify colors in all graphics applications.

The RGB Color Cube

The process of generating colors with three basic components is based on the RGB color cube, which is shown in Figure 19.1. The three dimensions of the color cube correspond to the three basic colors. The cube's corners are assigned each of the three primary colors, their complements, and the colors black and white. Complementary colors are easily calculated by subtracting their basic colors from 255. For example, the color (0, 0, 255) is a pure blue tone. Its complementary color is (255 – 0, 255 – 0, 255 – 255) or (255, 255, 0), which is a pure yellow tone. Blue and yellow thus are mapped to opposite corners of the cube. The same is true for red and cyan, green and magenta, and black and white. If you add a color to its complement, you get white.

FIGURE 19.1
Color specification of the RGB color cube



Notice that the components of the colors at the corners of the cube have either zero or full intensity. As you move from one corner to another along the same edge of the cube, only one of its components changes value. For example, as you move from the green to the yellow corner, the red component changes from 0 to 255. The other two components remain the same.

As you move between these two corners, you get all the available tones from green to yellow (256 in all). Similarly, as you move from the yellow to the red corner, the only component that changes is the green, and you get all the available shades from yellow to red. As you move from one corner to another on the same face of the cube along the diagonal, two components change value. To go from the yellow corner (255, 255, 0) to the cyan corner (0, 255, 255), you must change the red component from 255 to 0 and the blue component from 0 to 255. Finally, as you move along a diagonal of the cube — when you go blue to yellow, for instance — all three components must change value. As you can guess, this is how GDI + calculates the gradients: It draws an (imaginary) line between the two points that represent the starting and ending colors of the gradient in the RGB cube and picks the colors along this line. These colors are then used to generate the gradient.

To see an interactive animated color cube with the actual colors on its surface, visit this URL: `http://processing.org/learning/examples/rgbcube.html`. In addition to the colors on the surface of the color cube, its inside is also filled with colors. At the very center of the cube, you'll find a mid-gray shade, and you'll run into this shade as you move from black to white, from green to magenta, from red to cyan, and so on.

Defining Colors

To manipulate colors, use the Color class of the Framework. This is a shared class, and you need not create new Color objects; just call the appropriate property or method of the Color class. The Color class exposes 128 predefined colors as properties, which you can access by name, and additional members for specifying custom colors. For example, you can define colors by using the `FromARGB` method of the Color class. This method accepts three arguments, which are the components of the primary colors in the desired color:

```
Color.FromARGB(Red, Green, Blue)
```

The method returns a Color value, which you can assign to a variable of the same type, or use it directly as the value of a `Color` property. To change the form's background color to yellow, you can assign the value returned by the `FromARGB` method to the `BackColor` property of a form or control:

```
Form1.BackColor = FromARGB(255, 128, 128)
```

There's another form of the `FromARGB` method that accepts four arguments. The first argument in the method is the *transparency* of the color, or the alpha channel (which explains the *A* in the method's name). This component is similar to the other three color components, in the sense that it can be a value from 0 (totally transparent) to 255 (totally opaque). The other three arguments are the usual red, green, and blue color components. For more information on transparent colors, see the following section, ''Alpha Blending.''

You can also retrieve the three basic components of a Color value with the `R`, `G`, and `B` methods. (Yes, they're single-letter method names!) The following statements print the values of the three components of one of the named colors in the Output window:

```
Dim clr As Color = Color.Beige
Debug.WriteLine "Red Component = " & clr.R.ToString
Debug.WriteLine "Green Component = " & clr.G.ToString
Debug.WriteLine "Blue Component = " & clr.B.ToString
```

Image-processing applications read the image's pixel values, isolate their basic color components, and then process them separately. Then they combine the processed components to produce the new pixel value.

Alpha Blending

Besides the red, green, and blue components, a Color value might also contain a transparency component. This value determines whether the color is opaque (255) or transparent (0). In the case of transparent colors, you can specify the degree of transparency. This component is the *alpha component*. The following statement creates a new color value, which is yellow and 25 percent transparent:

```
Dim trYellow As Color
trYellow = Color.FromARGB(192, Color.Yellow)
```

If you want to ''wash out'' the colors of an image on a form, draw a white rectangle with a transparency of 50 percent or more on top of the image. The size of the rectangle must be the same as the size of the form, so you can use the `ClientRectangle` form's property to retrieve the area taken by the form. Then create a solid brush with a semitransparent color by using the `Color.FromARGB` method. The following code segment does exactly that:

```
Dim brush As New SolidBrush(Color.FromARGB(128, Color.White))
Me.CreateGraphics.FillRectangle(brush, Me.ClientRectangle)
```

If you execute these statements repeatedly, the form will eventually become white. Another use of transparent drawing is to place watermarks on images that you'll publish on the Web. A *watermark* is a string or logo that's drawn transparently on the image. It doesn't really disturb the viewers, but it makes the image unusable on another site. It's a crude but effective way to protect your images on the Web. (If all images have your site's URL or your company name on them, they're useless to anyone else.)

The following statements place a watermark with the string `MySite.Com` on top of the image of a PictureBox control. The font is fairly large and bold, and the code assumes that the text fits in the width of the image.

```
Private Sub Button1_Click(...) Handles Button1.Click
    Dim WMFont As New Font("Impact", 36, FontStyle.Bold)
    Dim WMBrush As New SolidBrush(Color.FromArgb(64, 192, 255, 255))
    PictureBox1.CreateGraphics.DrawString( _
              "MySite.com", WMFont, WMBrush, 240, 0)
    WMBrush.Color = Color.FromArgb(128, 0, 0, 0)
    PictureBox1.CreateGraphics.DrawString( _
              "MySite.com", WMFont, WMBrush, 10, 320)
End Sub
```

The preceding statements print the logo at two locations on the image of the *`PictureBox1`* control with different colors, as shown in Figure 19.2. Run the ImageWatermarks project and run it to see the watermarked image, or open the `Watermark.tif` image in the project's folder.

You can also experiment with the watermark's size, color, and transparency. You can combine these statements with a simple program that scans all the images in a folder, to write an application that watermarks a large number of files en masse (you learned how to iterate through all the files in a folder in Chapter 15, ''Accessing Folders and Files'') Notice that if you click the Watermark Image button repeatedly, the watermark will become more and more evident, because the same string will be printed on top of the previous one, in effect reducing the transparency effect.

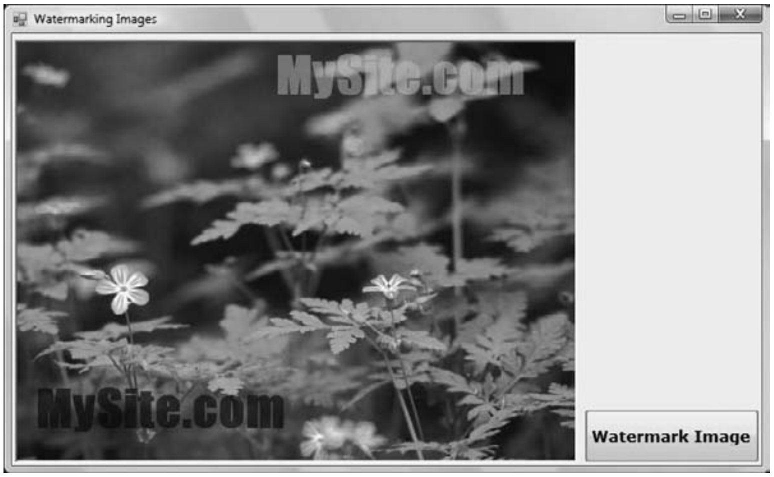


FIGURE 19.2
Watermarking an image with a semitransparent string

Another interesting application of transparency is to superimpose a semitransparent drawing over an opaque one. Figure 19.3 shows some text with a 3D look. To achieve this effect, you render a string by using a totally opaque brush. Then you superimpose the same string drawn with a partially transparent brush. The superimposed string is displaced by a few pixels in relation to the first one. The amount of displacement, its direction, and the colors you use determine the type of 3D effect (raised or depressed). The second brush can have any color, as long as the color combination produces a pleasant effect. The strings shown in Figure 19.3 were generated with the TextEffects project (via the Draw Semi-Transparent Text button), which was discussed in the preceding chapter. If you run the application and look at the rendered strings carefully, you'll see that they're made up of three colors. The two original colors appear around the edges. The inner area of each character is what the transparency of the second color allows us to see.

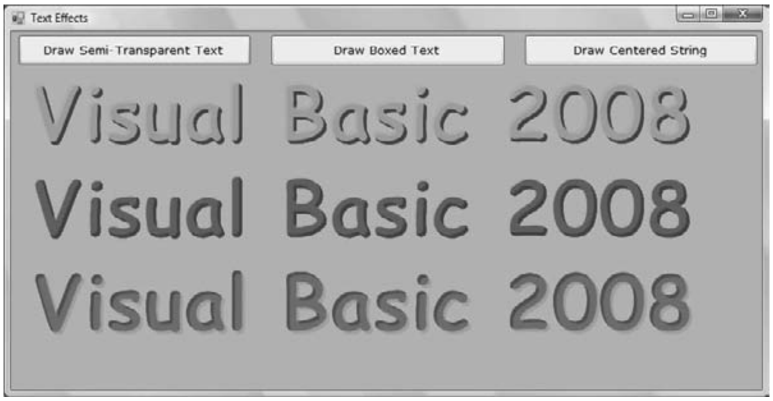


FIGURE 19.3
Creating a 3D effect by superimposing transparency on an opaque and a semitransparent string

The code behind the Draw Semi-Transparent Text button is quite simple, really. First it draws the string with the solid blue brush:

```
brush = New SolidBrush(Color.FromARGB(255, 0, 0, 255))
```

Then another instance of the same string is drawn, this time with a different brush:

```
brush.Color = Color.FromARGB(192, 0, 255, 255)
```

This is a semitransparent shade of cyan. The two superimposed strings are displaced a little with respect to one another. The statements in Listing 19.1 produced the strings of Figure 19.3.

Listing 19.1: Simple Text Effects with Transparent Brushes

```
Dim G As Graphics
Dim brush As SolidBrush

G = GetGraphicsObject()
G.TextRenderingHint = Drawing.Text.TextRenderingHint.AntiAlias
G.FillRectangle(New SolidBrush(Color.Silver), ClientRectangle)
Dim drawFont As Font
Dim drawString As String = "Visual Basic 2008"
' Draw string
brush = New SolidBrush(Color.FromArgb(255, 0, 0, 255))
drawFont = New Font("Comic Sans MS", 60, Drawing.FontStyle.Bold)
' Draw string
G.DrawString(drawString, drawFont, brush, 10, 30)
brush.Color = Color.FromArgb(192, 0, 255, 255)
' Draw same string with a displacement (-3, -3) pixels
G.DrawString(drawString, drawFont, brush, 7, 27)
brush.Color = Color.FromArgb(255, 0, 0, 255)
G.DrawString(drawString, drawFont, brush, 10, 130)
brush.Color = Color.FromArgb(128, 0, 255, 255)
G.DrawString(drawString, drawFont, brush, 7, 127)
' Draw same string with a displacement (3, 3) pixels
brush.Color = Color.FromArgb(255, 128, 64, 255)
G.DrawString(drawString, drawFont, brush, 10, 230)
brush.Color = Color.FromArgb(128, 255, 128, 64)
G.DrawString(drawString, drawFont, brush, 13, 233)
Me.Invalidate()
```

The Image Object

Images are two-dimensional arrays that hold the color values of the pixels making up the image. This isn't how images are stored in their respective files: JPG or JPEG (Joint Photographic Experts Group), GIF (Graphics Interchange Format), TIFF (Tagged Image File Format), and so on, but it's a convenient abstraction for the developer. To access a specific pixel of an image, you need to specify only the horizontal and vertical coordinates of the desired pixel. Let's turn our attention to images, starting with the discussion of the Image object. Each pixel is a Long value; the first byte is the pixel's alpha value and the other three bytes are the red, green, and blue components.

The `Image` property of the PictureBox or Form control is an Image object, and there are several ways to create such an object. You can declare a variable of the Image type and then assign the `Image` property of the PictureBox control or the Form object to the variable:

```
Dim img As Image
img = PictureBox1.Image
```

The `img` Image variable holds the bitmap of the *`PictureBox1`* control. This code segment assumes that an image was assigned to the control at design time. As you will see shortly, you can call the `Save` method of the Image class to save the image to a disk file.

You can also create a new Image object from an image file by using the Image class's `FromFile` method:

```
Dim img As Image
img = Image.FromFile("Butterfly.jpg")
```

After the *`img`* variable has been set up, you can assign it to the `Image` property of a PictureBox control:

```
PictureBox1.Image = img
```

Properties

The Image class exposes several members, some of which are discussed next. Let's start with a few simple properties and then we'll examine the methods of the Image class.

HorizontalResolution, VerticalResolution

These are read-only properties that return the horizontal and vertical resolution of the image, respectively, in pixels per inch.

Width, Height

These are read-only properties that return the width and height of the image, respectively, in pixels. If you divide the dimensions of the image (properties `Width` and `Height`) by the corresponding resolutions (properties `HorizontalResolution` and `VerticalResolution`), you'll get the actual size of the image in inches — the dimensions of the image when printed, for instance.

PixelFormat

This is another read-only property that returns the image's pixel format, which determines the quality of the image. There are many pixel formats and they're all members of the `PixelFormat` enumeration. For now, I assume that you're using a color display with a depth of 24 bits per pixel. Images with 24-bit color are of the `Format24bppRgb` type. *Rgb* stands for *red, green, blue*, and *24bpp* stands for *24 bits per pixel*. Each of the basic colors in this format is represented by 1 byte (8 bits).

Methods

In addition to the basic properties, the Image class exposes methods for manipulating images, which are discussed next.

RotateFlip

This method rotates and/or flips an image, and its syntax is the following, where the `type` argument determines how the image will be rotated:

```
Image.RotateFlip(type)
```

This argument can have one of the values of the `RotateFlipType` enumeration, shown in Table 19.1.

Table 19.1: The *RotateFlipType* Enumeration

Member	Description
Rotate180FlipNone	Rotates image by 180 degrees
Rotate180FlipX	Rotates image by 180 degrees and then flips it horizontally
Rotate180FlipXY	Rotates image by 180 degrees and then flips it vertically and horizontally
Rotate180FlipY	Rotates image by 180 degrees and then flips it vertically
Rotate270FlipNone	Rotates image by 270 degrees (which is equivalent to rotating it by −90 degrees)
Rotate270FlipX	Rotates image by 270 degrees (which is equivalent to rotating it by −90 degrees) and then flips it horizontally
Rotate270FlipXY	Rotates image by 270 degrees (which is equivalent to rotating it by −90 degrees) and then flips it vertically and horizontally
Rotate270FlipY	Rotates image by 270 degrees (which is equivalent to rotating it by −90 degrees) and then flips it vertically
Rotate90FlipNone	Rotates image by 90 degrees
Rotate90FlipX	Rotates image by 90 degrees and then flips it horizontally
Rotate90FlipXY	Rotates image by 90 degrees and then flips it horizontally and vertically
Rotate90FlipY	Rotates image by 90 degrees and then flips it vertically
RotateNoneFlipNone	No rotation and no flipping
RotateNoneFlipX	Flips image horizontally
RotateNoneFlipXY	Flips image vertically and horizontally
RotateNoneFlipY	Flips image vertically

To vertically flip the image displayed on a PictureBox control, use the following statement:

```
PictureBox1.Image.RotateFlip(RotateFlipType.RotateNoneFlipY)
PictureBox1.Invalidate()
```

The `Invalidate` method redraws the control, and you must call it to display the new (flipped) image on the control. The first statement rotates the image, but it doesn't refresh the control. If you omit the call to the `Refresh` method, you won't see the effect of the `RotateFlip` method. If you switch to another window and then back to the application's window, the image will be flipped because Windows refreshes the form automatically when it brings it to the top of the z-order (in front of all other windows).

GetThumbnailImage

This method returns the thumbnail of the specified image. The *thumbnail* is a miniature version of the image, whose exact dimensions you specify as arguments. Thumbnail images are used as visual enhancements for selecting images in dialog boxes. The thumbnail takes a small fraction of the space taken by the actual image, and we can display many thumbnails on a form to let the user select the desired one(s) instead of selecting filenames. Thumbnails are usually stored in the same folder as the images and are updated as needed. You can also create a hidden subfolder under each folder of images and populate the subfolder with the thumbnails. Of course, you must also maintain the list of thumbnails as images are added, modified, and removed. Alternatively, you can create the thumbnails as needed and display them to the user, without having to worry about maintaining the thumbnail version of each image. This approach works quite well on fast systems, unless you run into a folder with thousands of large images.

The syntax of the `GetThumbnailImage` method is as follows:

```
Image.GetThumbnailImage(width, height, Abort, Data)
```

The first two arguments are the dimensions of the thumbnail. The other two arguments are callbacks, which are used when the process is aborted. Because thumbnails don't take long to generate, we'll ignore these two arguments for the purposes of this book and we'll set them both to Nothing. These two arguments enable you to request the generation of a large number of thumbnails, without waiting for each thumbnail to be generated. As soon as each thumbnail is generated, a user-supplied procedure (the callback procedure) is called.

The following statements create a thumbnail of the image selected by the user and display it on a PictureBox control. To test these statements, place a PictureBox and a Button on the form. Then add an instance of the Open dialog box to the form and insert the following statements in the Button's `Click` event handler:

```
' Display the FileOpen dialog box
Dim img As Image
img = Image.FromFile(OpenFileDialog1.FileName)
PictureBox1.Image = img.GetThumbnailImage(32, 32, Nothing, Nothing)
```

Save

If your application processes the displayed image during the course of its execution and you want to save the image, you can use the `Save` method of the Image object. The simplest syntax of the `Save` method accepts a single argument, which is the path of the file in which the image will be saved:

```
Image.Save(path)
```

To save the contents of the *`PictureBox1`* control to a file, you must use a statement like the following:

```
PictureBox1.Image.Save("c:\tmpImage.bmp")
```

The image will be saved in BMP format, not because of the specified file extension (which could be anything), but because this is the default format of the `Save` method. Another form of the `Save` method allows you to specify the format in which the image will be saved, where the *`format`* argument's value can be one of the members of the `ImageFormat` enumeration:

```
PictureBox1.Image.Save("c:\tmpImage.bmp", format)
```

The fully qualified name of the enumeration is System.Drawing.Imaging.ImageFormat, so you should import the library System.Drawing.Imaging into any project that uses the enumerations mentioned in this chapter. Thus, you won't have to fully qualify the name of the enumeration.

The `ImageFormat` enumeration contains members for all common image formats (see Table 19.2). After you import the System.Drawing.Imaging namespace to your project, use the following statement to save the image on the *`PictureBox1`* control in GIF format:

```
PictureBox1.Image.Save("c:\tmpImage.gif", ImageFormat.Gif)
```

TABLE 19.2: The *ImageFormat* Enumeration

MEMBER	DESCRIPTION	EXTENSION
Bmp	Bitmap image	BMP
Emf	Enhanced Metafile Format	EMF
Exif	Exchangeable Image Format	EXIF
Gif	Graphics Interchange Format	GIF
Icon	Windows icon	ICO
Jpeg	Joint Photographic Experts Group format	JPEG, JPG
MemoryBmp	Saves the image to a memory bitmap	
Png	W3C Portable Network Graphics format	PNG
Tiff	Tagged Image File Format	TIF
Wmf	Windows metafile	WMF

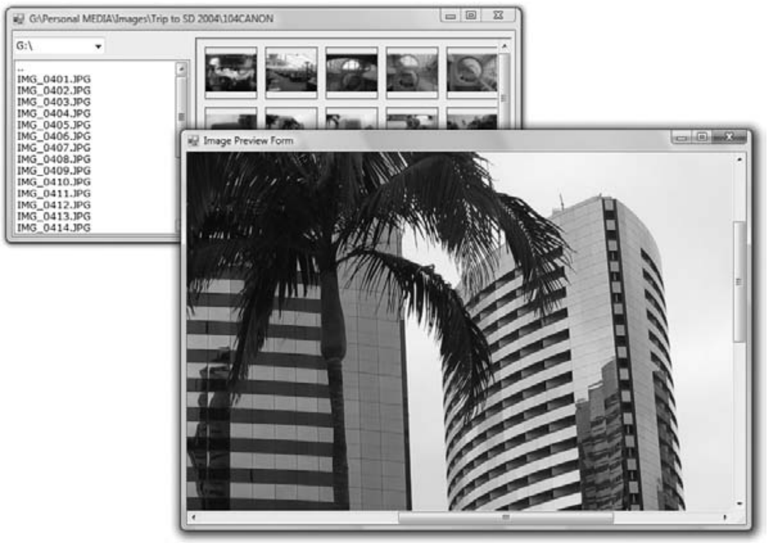
VB 2008 at Work: The Thumbnails Project

You can combine the `GetThumbnailImage` method of the Image object with the techniques described in Chapter 15 to scan a folder, retrieve all the image files, and create a thumbnail for each. As for displaying them, I suggest that you create as many PictureBox controls as there are images in the folder and then arrange them horizontally and vertically on a form. Chapter 7, ''Working with Forms,'' describes how to create instances of Windows controls at runtime and position them on the form from within your code.

Because this isn't a trivial project, I have included a sample project that demonstrates how to display thumbnails on a form. The project is called Thumbnails, and you will find it in this chapter's folder. I copied the CustomExplorer project of Chapter 15, renamed the main form to

`ThumbnailsForm`, and removed the ListView control in which the names of the files in the selected folder were displayed. In its place, the program displays the PictureBox controls with the thumbnails. When the user clicks an image, the program loads the selected image on the PictureBox control of another form and displays it. Figure 19.4 shows the main form of the Thumbnails application. You can double-click a thumbnail to preview the corresponding image on another form.

FIGURE 19.4
The Thumbnails application displays the images in a folder as thumbnails.



Then I adjusted the code to accommodate the display of thumbnails instead of filenames. The `ShowFilesInFolder()` subroutine of the original application displayed the names of the files in the current folder on a ListBox control. This subroutine was replaced by the `ShowImagesInFolder()` subroutine, which is shown in Listing 19.2.

LISTING 19.2: The *ShowImagesInFolder* Subroutine

```
Sub ShowImagesInFolder()
  Dim file As String
  Dim FI As FileInfo
  Dim PBox As PictureBox, img As Image
  Dim thmbLeft As Integer = 10
  Dim thmbTop As Integer = 10
  Dim PictureWidth As Integer = 64
  Panel1.Controls.Clear()
  Me.Invalidate()
  Dim pnlWidth As Integer = Panel1.Width
  For Each file In Directory.GetFiles(Directory.GetCurrentDirectory)
      FI = New FileInfo(file)
      If FI.Extension = ".GIF" Or _
         FI.Extension = ".JPG" Or _
         FI.Extension = ".TIFF" Then
          PBox = New PictureBox()
```

```
                PBox.SizeMode = PictureBoxSizeMode.Zoom
                PBox.BorderStyle = BorderStyle.FixedSingle
                img = Image.FromFile(FI.FullName)
                PBox.Image = img.GetThumbnailImage( _
                        PictureWidth, _
                        PictureWidth * (img.Height / img.Width), _
                        Nothing, Nothing)
                If thmbLeft + PictureWidth > pnlWidth Then
                  thmbLeft = 10
                  thmbTop += PictureWidth + 10
                End If
                PBox.Left = thmbLeft
                PBox.Top = thmbTop
                PBox.Width = PictureWidth
                PBox.Height = PictureWidth
                PBox.Visible = True
                PBox.Tag = FI.FullName
                Me.Controls.Item("Panel1"). _
                         Controls.Add(PBox)
                AddHandler PBox.Click, _
                               New System.EventHandler( _
                               AddressOf OpenImage)
                thmbLeft += PictureWidth + 10
                Application.DoEvents()
            End If
            FoldersList.Items.Add(FI.Name)
        Next
    End Sub
```

The PictureBox controls with the thumbnails are not added directly on the form; instead, they're added to a Panel control with its `AutoScroll` property set to True, so that users can scroll them independently of the other controls on the form. The subroutine starts by removing all controls from the *`Panel1`* control. Then the code goes through each file in the selected folder and examines its extension. If it's `JPG`, `GIF`, or `TIFF` (you can add more file extensions if you want), it creates a new PictureBox control, sets its size and location, loads the thumbnail of the image, and then adds it to the `Controls` collection of the *`Panel1`* control. Each image's path is stored in the PictureBox control's `Tag` property, and it is retrieved later to load the image on the second form, where it can be previewed.

Notice that the code adds a handler for the `Click` event of each PictureBox control. All the PictureBox controls share a common handler for their `Click` event, the `OpenImage()` subroutine. This subroutine reads the selected image's path from the `Tag` property of the control that fired the `Click` event and displays the corresponding image on the auxiliary form. The implementation of the `OpenImage()` subroutine is shown here:

```
    Sub OpenImage(ByVal sender As Object, ByVal e As System.EventArgs)
       Dim imgForm As New previewForm()
       imgForm.PictureBox1.Image = Image.FromFile(sender.tag)
       imgForm.Show()
    End Sub
```

previewForm is the name of the second form of the application, where the selected image is previewed. If you need more information about this project, please review the material in Chapter 7, which explains how to create instances of controls at runtime. This application is more of an exercise on designing dynamic forms instead of a demo of the `GetThumbnailImage` method, but it's an interesting application, and some readers might have a good use for the techniques demonstrated here. Notice that all the thumbnail bitmaps have a width of 64 pixels. To avoid possible distortion, I set the height of the thumbnail to a value that's close to 64 pixels, but proportionate to the image's width-to-height ratio.

Exchanging Images through the Clipboard

Whether you use bitmap images or create graphics from scratch with the Framework's drawing methods, sooner or later you'll want to exchange them with other Windows applications. To do so, you use the Clipboard and its `GetImage` and `SetImage` methods. The `SetImage` method accepts an image object as an argument and places it on the Clipboard. To copy the bitmap displayed on the *PictureBox1* control to the Clipboard, use the following statement:

```
Clipboard.SetImage(PictureBox1.Image)
```

The `GetImage` method returns the image on the Clipboard as an Image object. To read the bitmap stored in the Clipboard and display it on the *PictureBox1* control, you must use a statement like the following:

```
PictureBox1.Image = Clipboard.GetImage
```

Another interesting method of the Clipboard object is the `GetDataObject` method, which allows you to find out whether the Clipboard contains data of a specific type. This method returns an object of the IDataObject type, which in turns exposes the `GetData` and `GetDataPresent` methods. The `GetData` method returns the data on the Clipboard in the format specified by its argument, which is a member of the `DataFormats` enumeration (*Bitmap*, *WaveAudio*, *RTF*, and so on). The `GetDataPresent` method also accepts as an argument a member of the `DataFormats` enumeration and returns a True/False value indicating whether the Clipboard's contents are of the specific type.

The `GetImage` method will attempt to read the Clipboard's data as an Image object. To read the bitmap stored in the Clipboard and display it on the *PictureBox1* control, you should make sure that the Clipboard contains an image before calling the `GetImage` method:

```
If Clipboard.GetDataObject.GetDataPresent(DataFormats.Bitmap) Then
    PictureBox1.Image = Clipboard.GetImage
Else
    MsgBox("The Clipboard doesn't contain a bitmap!")
End If
```

Accessing the Clipboard with the My Object

The Clipboard access techniques discussed in the preceding section are those exposed by the Framework's Clipboard class. The My object offers a simpler alternative through its My.Computer.Clipboard component. This component exposes the `Contains`*`XXX`* properties, which return True if the Clipboard contains a specific format (`ContainsText` for text, `ContainsImage` for images, and so on). To retrieve the corresponding item from the Clipboard, use the `Get`*`XXX`* method (`GetText` for text, `GetImage` for images, and so on). The following `If` statement retrieves the image stored in the Clipboard, but only if the Clipboard contains image data:

```
Dim img As Image = Nothing
If My.Computer.Clipboard.ContainsImage Then
    img = My.Computer.Clipboard.GetImage
End If
```

If the Clipboard doesn't contain an image, the *`img`* variable will have a value of Nothing.

One of this chapter's sample projects is the ImageClipboard project, which demonstrates how to exchange images between a VB application and any other image-aware application running under Windows through the Clipboard. You can copy the image displayed in a PictureBox control on the application's main form to the Clipboard and then paste it to another application, or copy an image in any image-processing application and paste it on the same PictureBox. The application is straightforward, and you can open it with Visual Studio to examine its code. This example concludes our discussion of the Image object. In the following section, you'll explore the Bitmap object, and you'll learn how to access and manipulate individual pixels in an image.

The Bitmap Object

The Image class doesn't provide any methods for manipulating a bitmap; it's a class for storing bitmaps only. The Bitmap class, on the other hand, provides methods that allow you to read and set its pixels. In the last section of this chapter, you're going to build an image-processing application. The Bitmap object's constructor is heavily overloaded, and you can create empty bitmaps with specific properties or import images from files and streams. Two of the simplest forms of the Bitmap object's constructors are shown here:

```
Dim bmp As New Bitmap(filename)
Dim bmp As New Bitmap(stream)
```

They both create a new Bitmap object and initialize it to the bitmap of an image. This image is read from a file (with the first form of the constructor) or from a Stream object (with the second form of the constructor). The properties of the Bitmap object (its dimensions and color depth) are determined by the image assigned to it. You can also create empty bitmaps with specific properties by using the following form of the constructor:

```
Dim bmp As New Bitmap(width, height, pixelformat)
```

The first two arguments are the bitmap's dimensions in pixels, and the last argument is the same as the `PixelFormat` property of the Image object. An image's pixel format is fixed; you can't change it. When you create an empty Bitmap object, however, you can specify the format of its pixels. You can also retrieve an image's bitmap by initializing the Bitmap object with an Image object, where *`img`* is a properly initialized Image object:

```
Dim bmp As New Bitmap(img)
```

The Bitmap object doesn't expose a `Graphics` property, and therefore you can't draw any shapes directly on a bitmap. The Bitmap object, however, exposes two methods for accessing its pixels: the `GetPixel` and `SetPixel` methods. The syntax of these two methods is as follows, where *`X`* and *`Y`* are the coordinates of the pixel whose value you're reading, or setting:

```
color = Bitmap.GetPixel(X, Y)
Bitmap.SetPixel(X, Y, color)
```

The `GetPixel` method returns the color of the specified pixel, whereas the `SetPixel` method sets the pixel's color to the specified value. (You'll see how these two methods are used in the following section.) To rotate the bitmap, you can use the `RotateFlip` method, whose syntax is identical to the syntax of the `RotateFlip` method of the Image object.

It's possible to place graphics on a bitmap by using the drawing methods, but you must first create a Graphics object that represents the Bitmap object's surface. One way to obtain a Graphics object for the bitmap is to call the Graphics class's `FromImage` method, passing the Bitmap object as an argument:

```
Dim G As Graphics
G = Graphics.FromImage(bmp)
```

After you obtain the Graphics object, you can draw on it by using all the drawing methods discussed in the preceding chapter. Eventually, you'll have to display the bitmap on a control to see what it looks like. The following statements create an Image object based on the bitmap and display it on a PictureBox control:

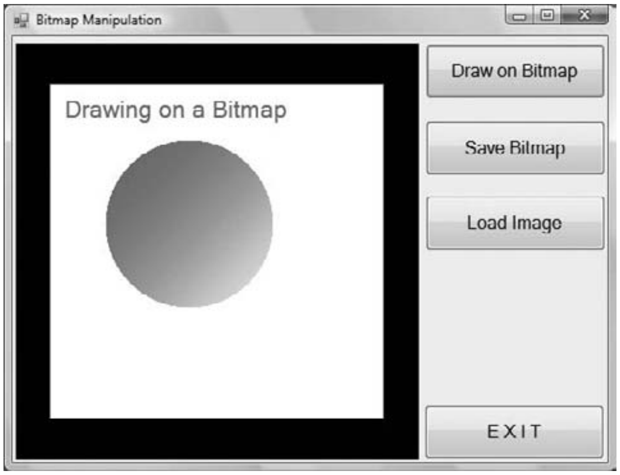
```
Dim img As Image
img = CType(bmp, Image)
PictureBox1.Image = img
```

When you're finished editing the bitmap, you can save it as an image file via the `Save` method of the Bitmap object. The `Save` method accepts as arguments the path of the file in which the image will be stored and the type of the image:

```
Bitmap.Save(path, imageType)
```

The second argument is a member of the `ImageFormat` enumeration (its members were shown in Table 19.2). The project BitmapManipulation, whose main form is shown in Figure 19.5, demonstrates how to create a bitmap in memory from within your code, save it to a file, and load the image. I hard-coded the file's path for the purposes of a simple demo, and it's always `BitmapImage.jpg`.

FIGURE 19.5
The BitmapManipulation application demonstrates how to draw on a bitmap from within your code.



The JPG format compresses the image at the expense of its quality. If you carefully examine the image saved and reloaded to the PictureBox control, you'll notice artifacts. Change the image format to TIFF, and you'll see that this format doesn't sacrifice image quality for size compression.

An interesting application of this technique of generating bitmaps in memory is to create graphics on-the-fly. For example, you can create a stack of boxes, books, and so on to indicate the sales volume in a period (the higher the volume, the taller the stack). The exact image depends on some live data and is different every time. Instead of storing dozens of images and selecting the proper one every time, create the image on-the-fly. The same technique can be used to create fancy counter images on web pages.

One last interesting method of the Bitmap object is the `MakeTransparent` method, which accepts a color as an argument and treats it as transparent. Any areas of the bitmap you want to treat as transparent (for example, as ''holes'' in the bitmap) should be filled with this color. When this image is placed on a form or another image, the transparent areas allow the underlying colors to show through.

Processing Bitmaps

A bitmap is a two-dimensional array of color values. These values are stored in disk files, and when an image is displayed on a PictureBox or Form control, each of its color values is mapped to a pixel on the PictureBox or form. This is true when the image isn't resized, of course. When the image is resized (when displayed on a PictureBox control with its `SizeMode` property set to *`Stretch`* or *`Zoom`*, for example), the mapping between the monitor pixels and image pixels is no longer one to one. The image is resized via some interpolation technique. Yet the Image object returned by the control's `Image` property isn't affected; the control ''sees'' the original image.

As you'll see, image processing is nothing more than simple arithmetic operations on the values of the image's pixels. The ImageProcessing application we'll build to demonstrate the various image-processing techniques doesn't have the features of a professional application, but it demonstrates the principles of these techniques and can be used as a starting point for custom applications.

We'll build a simple image-processing application that can read all the image types that the Framework can handle (BMP, GIF, TIFF, JPG, and so on), process them, and then display the processed images. There are simpler ways to demonstrate the pixel-manipulation methods, but image processing is an intriguing topic, and I hope you'll experiment with the techniques presented in this section.

Let's look at a simple technique: the inversion of an image's colors. To invert an image, you must change all pixels to their complementary colors — black to white, green to magenta, and so on. (The complementary colors are on opposite corners of the RGB cube, which was shown in Figure 19.1.)

To calculate complementary colors, you subtract each of the three color components from 255. For example, a pure green pixel whose value is (0, 255, 0) will be converted to (255 – 0, 255 – 255, 255 – 0) or (255, 0, 255), which is magenta. Similarly, a mid-yellow tone (0, 128, 128) will be converted to (255 – 0, 255 – 128, 255 – 128) or (255, 127, 127), which is a mid-brown tone. If you apply this color transformation to all the pixels of an image, the result will be the negative of the original image (what you'd see if you looked at the negative, back in the days of film cameras).

Other image-processing techniques aren't as simple, but image processing is generally as straightforward as arithmetic operations on the image's pixels. After we go through the Image-Processing application, you'll probably come up with your own techniques and be able to implement them.

Refreshing the Image

When you draw on a bitmap, which is associated with the `Image` property of a PictureBox control, the image on the control isn't refreshed every time the bitmap is modified. Instead, the image is modified when the `Paint` event has a chance to be serviced. The processing is implemented with two nested loops that iterate through the bitmap's rows and columns, as in the following code:

```
For pxlCol As Integer = 0 To PictureBox1.Image.Height - 1
    For pxlRow As Integer = 0 To PictureBox1.Image.Width - 1
        ' statements to process current pixel:
        ' (pxlRow, pxlCol)
    Next
Next
```

The image on the control won't be refreshed until the outer loop has finished. As a result, users can't see the progress of the operation; they will see the new image after all its pixels have been processed.

To force the PictureBox control to refresh its image, you must call the `Refresh` method. This method, however, isn't instant. If you insert it in the inner loop, it will make the processing time simply unacceptable. You can insert this statement between the two `Next` statements, so that users will see each new column of pixels as they're processed. Even so, the `Refresh` method introduces a substantial delay. On my computer, it took less than 2 seconds to copy the pixels of a 1,024 × 768 image from one PictureBox control to another. When I introduced a call to the `Refresh` method after processing an entire column of pixels, the time jumped to 8 seconds.

It's actually much faster to update a ProgressBar control from within your code than to update an image. The obvious solution is to avoid refreshing the PictureBox control too often, but then again you're giving up the immediate feedback.

VB 2008 at Work: The ImageProcessing Project

The application we'll develop in this section is called ImageProcessing, and its main form is shown in Figure 19.6. It's not a professional tool, but it can be easily implemented in Visual Basic 2008 and it will give you the opportunity to explore various image-processing techniques on your own.

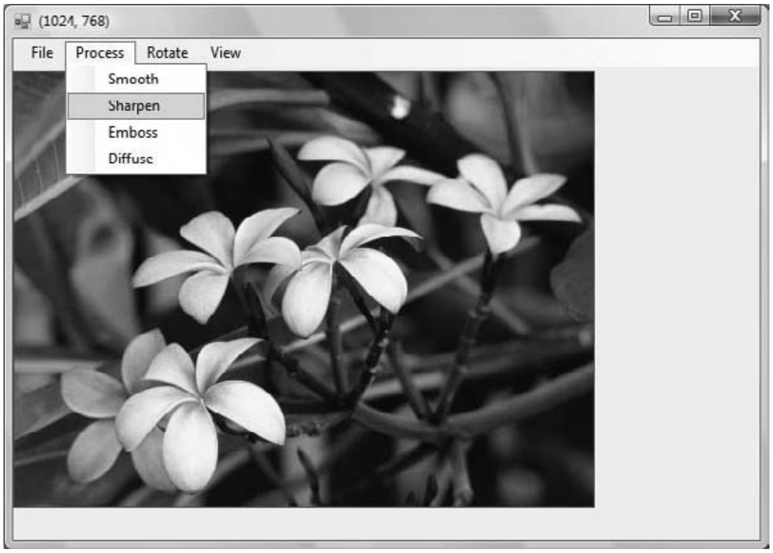


FIGURE 19.6 The Image-Processing application demonstrates several image-processing techniques, all implemented with VB 2008.

To process an image with the application, choose File ➢ Open to load it to the PictureBox control and then select the type of action from the Process menu. You can also zoom in or out by using the commands of the View menu, and you can rotate the image by using the commands of the Rotate menu.

The sample application implements the following image-processing techniques:

Smooth Reduces the amount of detail in the image by smoothing areas with abrupt changes in color and/or intensity. Smoothing blurs the image, and extreme smoothing results in total loss of detail.

Sharpen Brings out the detail in the image by amplifying the differences between similarly colored pixels.

Emboss Adds a raised (embossed) look to the image.

Diffuse Gives the image a ''painterly'' look.

Next, let's look at how each algorithm works and how it's implemented in Visual Basic.

How the Application Works

Let's start with a general discussion of the application's operation before looking at the actual code. After the image is loaded on a PictureBox control, you can access the values of its pixels with the `GetPixel` method of the Bitmap object that holds the image. The `GetPixel` method returns a Color value; you can use the `R`, `G`, and `B` methods of the Color object to extract the basic color components. This is a time-consuming step, and for most algorithms it must be performed more than once for each pixel.

All image-processing algorithms read a few pixel values and process them to calculate the new value of a pixel. This value is then written into the new bitmap via the `SetPixel` method.

To process an image, we set up two nested loops: an outer loop scans the rows of pixels, and an inner loop scans the pixels in each row. In the inner loop's body, we calculate the current pixel's new value, taking into consideration the values of the surrounding pixels. Because of this, we can't save the new pixel values to the original bitmap. When processing the next pixel, some of the surrounding pixels will have their original values, whereas others will have the new values. As a result, we must create a copy of the original bitmap and use this bitmap to retrieve the original values of the pixels. The processed values are displayed on the bitmap of the PictureBox control, so that you can watch the progress of the processing. The following is the outline of all the algorithms that we'll implement shortly:

```
bmp = New Bitmap(PictureBox1.Image)
PictureBox1.Image = bmp
Dim tempbmp As New Bitmap(PictureBox1.Image)
Dim pixRow, pixCol As Integer
With tempbmp
   For pixRow = DX To .Height - DX - 1
      For pixCol = DY To .Width - DY - 1
         { calculate new pixel value }
         bmp.SetPixel(pixRow, pixCol, new_pixel_value)
      Next
      If i Mod 10 = 0 Then
         PictureBox1.Invalidate()
      End If
   Next
End With
```

Here's how it works. First, we create a Bitmap object from the image on the PictureBox control. This is the *bmp* variable, which is then assigned back to the `Image` property of the control. Everything you draw on the *bmp* object will appear on the control's surface. We then create another identical Bitmap object, the *tempbmp* variable. This object holds the original values of all the pixels of the image.

The two nested loops go through every pixel in the image. In the inner loop's body, we calculate the new value of the current pixel and then write this value to the matching location of the *bmp* object. The new pixel will appear on the control when we refresh it by calling the control's `Invalidate` method. This method isn't called every time we display a new pixel. It would introduce a significant delay, so we invalidate the control after processing 10 rows of pixels. This is a good balance between performance and a visual feedback of the process's progress. We could have displayed a dialog box with a progress bar to indicate the progress of the operation. If a simple indication will do, you can simply display the percentage of the completed work on the form's title bar.

Applying Effects

In this section, you'll find a short description of the algorithm that implements each effect. You can open the ImageProcessing project with Visual Studio and examine the code, which contains a lot of comments explaining the various operations.

Smoothing Images

One of the simplest and most common operations in all image-processing programs is the smoothing (or blurring) operation. The smoothing operation is equivalent to low-pass filtering: Just as you can cut off a stereo's high-frequency sounds with the help of an equalizer, you can cut off the high frequencies of an image. If you're wondering what the high frequencies of an image are, think of them as the areas with abrupt changes in the image's intensity. These are the areas that are mainly affected by the blurring filter.

The smoothed image contains fewer abrupt changes than the original and looks a lot like the original image seen through a semitransparent glass. Figure 19.7 shows a smoothed image obtained with the ImageProcessing application. This image is a good candidate for smoothing, given that it has a lot of detail, especially in the area of the water drops.

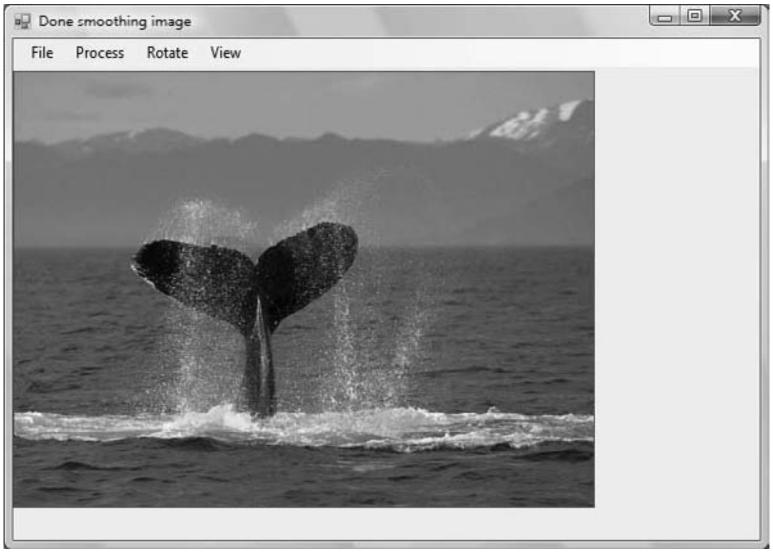


FIGURE 19.7
Smoothing an image reduces its detail, but can make the image less ''noisy'' and ''busy.''

To smooth an image, you must reduce the large differences between adjacent pixels. Let's take a block of 9 pixels, centered on the pixel we want to blur. This block contains the pixel to be blurred and its eight immediate neighbors. Let's assume that all the pixels in this block are green except for the middle one, which is red. This pixel is drastically different from its neighbors, and for it to be blurred, it must be pulled toward the average value of the other pixels. Taking the average of a block of pixels is, therefore, a good choice for a blurring operation. If the current pixel's value is similar to the values of its neighbors, the average won't significantly affect its value. If its value is drastically different, the remaining pixels will ''pull'' the current pixel's value toward them. In other words, if the middle pixel were green, the average wouldn't affect it. Because it's the only red pixel in the block, however, it will come closer to the average value of the remaining pixels. It will assume a green tone.

The intensity of blurring depends on the size of the block over which the average is calculated. We used a 3 × 3 block in our example, which yields an average blur. To blur the image even more, use a 5 × 5 block. Even larger blocks will blur the image to the point that useful information is lost. The actual code of the Smooth operation scans all the pixels of the image (excluding the edge pixels that don't have neighbors all around them) and takes the average of their RGB components (one value per component). It then combines the three values by using the method `Color.FromARGB` to produce the new value of the pixel.

Sharpening Images

Because the basic operation for smoothing an image is addition, the opposite operation will result in sharpening the image. The sharpening effect is more subtle than smoothing, but is also more common and more useful. Nearly every image published, especially in monochrome (''one-color'') publications, must be sharpened to some extent. Sharpening an image consists of highlighting the edges of the objects in it, which are the very same pixels blurred by the previous algorithm. *Edges* are areas of an image with sharp changes in intensity between adjacent pixels.

In a smooth area of an image, the difference between two adjacent pixels will be zero or a very small number. If the difference is zero, the two pixels are nearly identical, which means that there's nothing to sharpen. This is called a *flat* area of the image. If the pixels are on an edge, the difference between two adjacent pixels will be a large value (perhaps negative). This is an area of the image with some degree of detail that can be sharpened. The difference between adjacent pixels isolates the areas with detail and completely flattens out the smooth areas. The question now is how to bring out the detail without leveling the rest of the image. How about adding the difference to the original pixel? Where the image is flat, the difference is negligible, and the processed pixel will be practically the same as the original one. If the difference is significant, the processed pixel will be the original plus a value that's proportional to the magnitude of the detail. The sharpening algorithm can be expressed as follows:

```
new_value = original_value + 0.5 * difference
```

If you simply add the difference to the original pixel, the algorithm brings out too much detail. You usually add a fraction of the difference; a 50 percent factor is common. You can also use different factors for different components (use a 60 percent factor for the green component and a 40 percent factor for the red component, for example).

Embossing Images

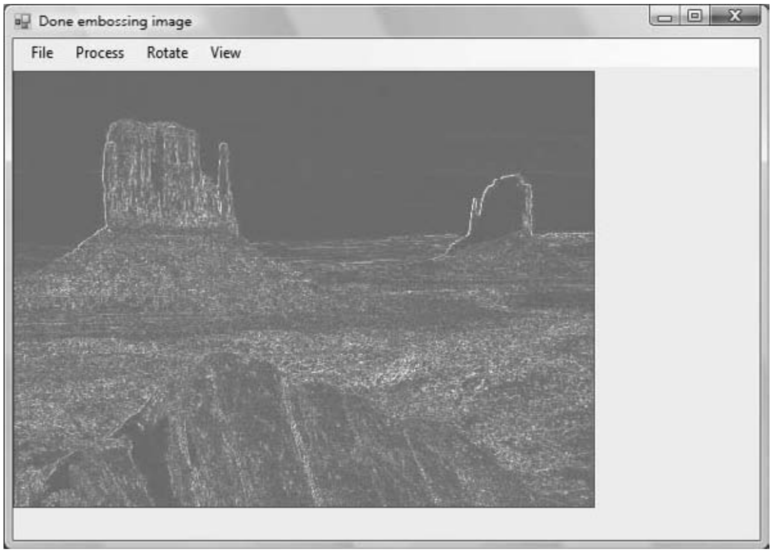
To sharpen an image, we add the difference between adjacent pixels to the pixel value. What do you think would happen to a processed image if you took the difference between adjacent pixels only? The flat areas of the image would be totally leveled, and only the edges would remain visible. The result would be an image like the one shown in Figure 19.8. This effect clearly sharpens the edges and flattens the smooth areas of the image. By doing so, it gives depth to the image. The processed image looks as if it's raised and illuminated from the right side. This effect is known as *embossing* or *bas relief*.

The actual algorithm is based on the difference between adjacent pixels. For most of the image, however, the difference between adjacent pixels is a small number, and the image will turn black. The Emboss algorithm adds a constant to the difference to bring some brightness to areas of the image that would otherwise be dark. The algorithm can be expressed as follows:

```
new_value = difference + 128
```

As usual, you can take the difference between adjacent pixels in the same row, adjacent pixels in the same column, or diagonally adjacent pixels. The code that implements the Emboss filter in the ImageProcessing application uses differences in the x and y directions. (Set one of the variables *`DispX`* or *`DispY`* to 0 to take the difference in one direction only.)

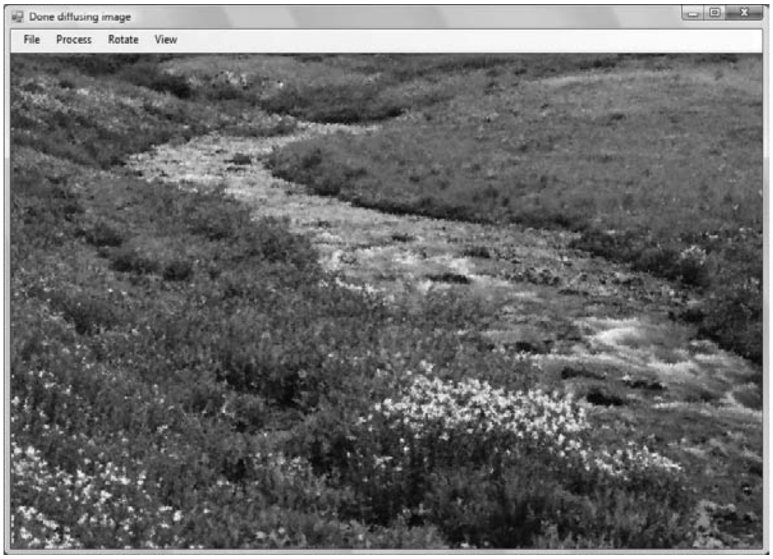
FIGURE 19.8
The Emboss special effect



Diffusing Images

The Diffuse special effect is different from the previous ones, in the sense that it's not based on the sums or the differences of pixel values. This effect uses the Random class to introduce some randomness to the image and give it a ''painterly'' look, as demonstrated in Figure 19.9. You can control the intensity of the effect by applying the same type of processing repeatedly to the image.

FIGURE 19.9
The Diffuse special effect gives the image a painterly look.



This time, we won't manipulate the values of the pixels. Instead, the current pixel will assume the value of another one, selected randomly in its 5 × 5 neighborhood with the help of the Random class.

The Diffuse algorithm is the simplest one. It generates two random variables, `DX` and `DY`, in the range −3 to 3. These two variables are added to the coordinates of the current pixel to yield the coordinates of another pixel in the neighborhood. The original pixel is replaced by the value of the pixel that is (`DX`, `DY`) pixels away.

Open the ImageProcessing application to explore the code that implements the various effects. Change the parameters of the various algorithms and see how they affect the processed image. You can easily implement new algorithms by inserting the appropriate code in the inner loop's body. The rest of the code remains the same. Some simple ideas include clipping one or more colors (force the red color component of each pixel to remain within a range of values), substituting one component for another (replace the red component of each pixel with the green or blue component of the same pixel), inverting the colors of the image (subtract all three color components of each pixel from 255), and so on. With a little imagination, you can create interesting effects for your images.

The Bottom Line

Specify colors. Color values are based on the RGB cube. Each color is a point in the RGB code and is expressed as a triplet of integer values that represent the intensity of the red, green, and blue components of the color. You can also use the named colors of the Color class.

Master It How do you draw with semitransparent colors?

Manipulate images and bitmaps. The Framework provides two classes for representing images: the Image and Bitmap classes. The Image class represents images. You use Image objects to read images from files or streams and to store them in memory. You can't use the Image object to create a new image. The Bitmap class also represents images, but you can use the Bitmap object to create new images from within your code.

Master It When will you use an Image object versus a Bitmap object in a graphics application?

Process images. Images are two-dimensional arrays of color values, one value per pixel, arranged in rows and columns. To process an image's pixels, start by reading the image into a Bitmap object. Then set up two nested loops that iterate through each row and each column of pixels. Use the `GetPixel` method to read pixel values, and the `SetPixel` method to change a pixel's value.

Master It Outline the code that processes the pixels of an image.

Chapter 20

Printing with Visual Basic 2008

The topic of printing with Visual Basic is a not trivial, and many developers use third-party tools to add print capabilities to their applications. As you already know, there's no control with built-in printing capabilities. It would be nice if certain controls, such as the TextBox or the ListView control, would print their contents, but this is not the case. Even to print a few text paragraphs entered by the user on a TextBox control, you must provide your own code.

Printing with VB isn't complicated, but it requires a lot of code — most of it calling graphics methods. You must carefully calculate the coordinates of each graphic element placed on the paper, take into consideration the settings of the printer and the current page, and start a new page when the current one is filled. It's like generating graphics for the monitor, so you need a basic understanding of the graphics methods, even if you're only going to develop business applications. If you need to generate elaborate printouts, I suggest that you look into third-party controls with built-in printing capabilities, because the controls that come with Visual Studio have no built-in printing capabilities.

The examples of this chapter will address many of your day-to-day needs, and I'm including examples that will serve as your starting point for some of the most typical printing needs, from printing tabular data to bitmaps.

In this chapter, you'll learn how to do the following:

- Use the printing controls and dialog boxes
- Print plain text and images
- Print tabular data

The Printing Components

We'll start our exploration of Visual Basic's printing capabilities with an overview of the printing process, which is the same no matter what you print. In this section, you'll find a quick overview of the printing controls (you'll find more information on them, as well as examples, in the following sections). You don't need to use all these components in your project. Only the PrintDocument component is required, and you will have to master the members of this control.

The PrintDocument Control

This object represents your printer, and you must add a PrintDocument control to any project that generates printouts. In effect, everything you draw on the PrintDocument object is sent to the printer. The PrintDocument object represents the printing device, and it exposes a Graphics object that represents the printing surface, just like the `Graphics` property of all Windows controls. You can program against the Graphics object by using all the graphics methods discussed in Chapter 18, ''Drawing and Painting with Visual Basic 2008.'' If you can create drawings on a form, you can just as easily print them on your printer. To print text, for example, you must call the `DrawString` method. You can also print frames around the text with the `DrawLine` or `DrawRectangle` method. In general, you can use all the methods of the Graphics object to prepare the printout.

The PrintDocument control is invisible at runtime, and its icon will appear in the Components tray at design time. When you're ready to print, call the PrintDocument object's `Print` method. This method doesn't produce any output, but it does raise the control's `BeginPrint` and `PrintPage` events. The `BeginPrint` event is fired as soon as you call the `Print` method, and this is where you insert the printout's initialization code. The `PrintPage` event is fired once for every page of the printout, and this is where you must insert the code that generates output for the printer. Finally, the `EndPrint` event is fired when the printout ends, and this is where you insert the code to reset any global variables.

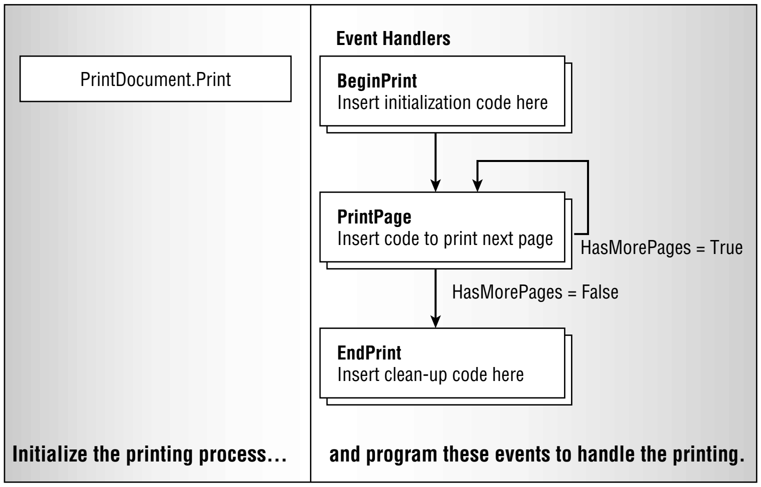
The following statement initiates the printing:

```
PrintDocument1.Print
```

This statement is usually placed in a button's or a menu item's `Click` event handler. To experiment with simple printouts, create a new project, place a button on the form, add an instance of the PrintDocument object to the form, and enter the preceding statement in the button's `Click` event handler.

After the execution of this statement, the `PrintDocument1_PrintPage` event handler takes over. This event is fired for each page, so you insert the code to print the first page in this event's handler. The `PrintPage` event exposes the `e` argument, which gives you access to the `Graphics` property of the current Printer object. This is the same object we used in the two preceding chapters to generate all kinds of graphics. The printer has its own Graphics object, which represents the page you print on. If you need to print additional pages, you set the `e.HasMorePages` property to True just before you exit the event handler. This will fire another `PrintPage` event. The same process will repeat until you've printed everything. After you finish, you set the `e.HasMorePages` property to False, and no more `PrintPage` events will be fired. Instead, the `EndPrint` event will be fired and the printing process will come to an end. Figure 20.1 outlines the printing process.

FIGURE 20.1
All printing takes place in the `PrintPage` event handler of the PrintDocument object.


PrintDocument.Print
Event Handlers
BeginPrint
Insert initialization code here
PrintPage
Insert code to print next page
HasMorePages = True
HasMorePages = False
EndPrint
Insert clean-up code here
Initialize the printing process...
and program these events to handle the printing.

The code in Listing 20.1 shows the structure of a typical `PrintPage` event handler. The `PrintPage` event handler prints three pages with the same text but a different page number on each page.

LISTING 20.1: A Simple *PrintPage* Event Handler

```
Private Sub PrintDocument1_PrintPage( _
         ByVal sender As Object, _
         ByVal e As System.Drawing.Printing.PrintPageEventArgs) _
         Handles PrintDocument1.PrintPage
   Static pageNum As Integer
   Dim prFont As New Font("Verdana", 24, GraphicsUnit.Point)
   e.Graphics.DrawString( _
            "PAGE " & pageNum + 1, prFont, _
            Brushes.Black, 700, 1050)
   e.Graphics.DrawRectangle(Pens.Blue, 0, 0, 300, 100)
   e.Graphics.DrawString( _
            "Printing with VB 2005", prFont, _
            Brushes.Black, 10, 10)
   ' Add more printing statements here
   ' Following is the logic that determines whether we're done printing
   pageNum = pageNum + 1
   If pageNum <= 3 Then
      e.HasMorePages = True
   Else
      e.HasMorePages = False
      pageNum = 0
   End If
End Sub
```

Notice that the page number is printed at the bottom of the page, but the corresponding statement is the first one in the subroutine. I assume that you're using a letter-size page, so I hard-coded the coordinates of the various elements in the code. Later in this chapter, you'll learn how to take into consideration not only the dimensions of the physical page, but also its orientation.

The *`pageNum`* variable is declared as Static, so it retains its value between invocations of the event handler and isn't reset automatically. The last statement resets the *`pageNum`* variable in anticipation of another printout. Without this statement, the first page of the second printout (if you clicked the button again) would be page 4, and so on. Moreover, the printout would never come to an end because the *`pageNum`* variable would never become less than 3. Every time you repeat a printout, you must reset the global and static variables. This is a common task in printing with the PrintDocument control, and is a common source of many bugs.

INITIALIZATION OF STATIC VARIABLES

You can also declare variables such as the *`pageNum`* variable at the form's level, so that they'll retain their value between successive invocations of the `PrintPage` event handler. These variables can be reset in the PrintDocument's `BeginPrint` event handler, which is fired every time you start a new printout by calling the `PrintDocument.Print` method.

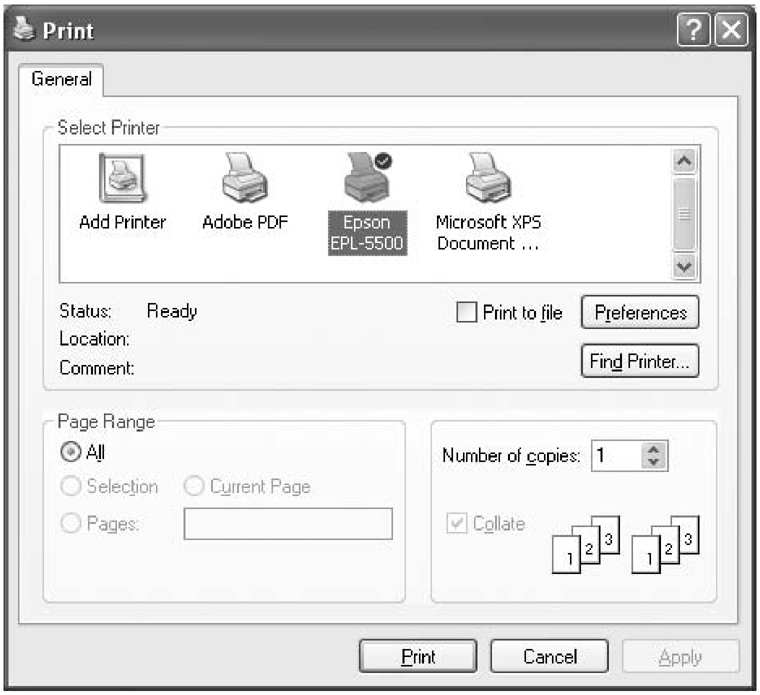
The code of Listing 20.1 uses the methods of the `e.Graphics` object to generate the printout. After printing something and incrementing the page number, the code sets the `e.HasMorePages` property to True, to fire the `PrintPage` event again, this time to print the next page. As long as there are more pages to be printed, the program sets the `e.HasMorePages` property to True. After printing the last page, it sets the same argument to False to prevent further invocations of the `PrintPage` event. If you want to print a single page, you can ignore everything in this listing, except for the drawing methods that produce the output.

The entire printout is generated by the same subroutine, one page at a time. Because pages are not totally independent of one another, we need to keep some information in variables that are not initialized every time the `PrintPage` event handler is executed. The page number, for example, must be stored in a variable that will maintain its value between successive invocations of the `PrintPage` event handler, and it must be increased every time a new page is printed. If you're printing a text file, you must keep track of the current text line, so that each page will pick up where the previous one ended, not from the beginning of the document. You can use static variables or declare variables on the form's level, whatever suits you best. This is a recurring theme in programming the `PrintPage` event, and you'll see many more examples of this technique in the following sections. I can't stress enough the importance of resetting these variables at the end of a printout (or initializing them at the beginning of the printout).

The PrintDialog Control

The PrintDialog control displays the standard Print dialog box, shown in Figure 20.2, which allows users to select a printer and set its properties. If you don't display this dialog box, the output will be sent automatically to the default printer and will use the default settings of the printer.

FIGURE 20.2
The Print dialog box



Status: Ready
Location:
Comment:
Number of copies: 1

To display the Print dialog box, call the PrintDialog control's `ShowDialog` method. However, you must set the control's `PrinterSettings` property first; if not, a runtime exception will be thrown. We usually display the Print dialog box via the following statements:

```
PrintDialog1.PrinterSettings = PrintDocument1.PrinterSettings
If PrintDialog1.ShowDialog() = Windows.Forms.DialogResult.OK Then
    PrintDocument1.PrinterSettings = PrintDialog1.PrinterSettings
End If
```

Among other settings, the Print dialog box allows you to specify the range of pages to be printed. Before allowing users to select a range, be sure that you have a way to skip any number of pages. If the user specifies pages 10 through 19, your code must calculate the section of the document that would normally be printed on the first nine pages, skip it, and start printing after that. If the printout is a report with a fixed number of rows per page, skipping pages is trivial. If the printout contains formatted text, you must execute all the calculations to generate the first nine pages and ignore them (skip the statements that actually print the graphics). Starting a printout at a page other than the first one can be a challenge, so make sure that your code will work before enabling the Print Range zone in the Print dialog box.

When users select a printer in this dialog box, it automatically becomes the active printer. Any printout generated after the printer selection will be sent to that printer; you don't have to insert any code to switch printers. The actual printer to which you will send the output of your application is almost transparent to the printing code. The same commands will generate the same output on any printer. It is also possible to set the printer from within your code by using a statement like the following, where *`printer`* is the name of one of the installed printers:

```
PrintDocument1.PrinterSettings.PrinterName = printer
```

For more information on selecting a printer from within your code, see the section called ''Printer and Page Properties,'' later in this chapter. There are times when you want to set a printer from within your code and not give users a chance to change it. An application that prints invoices and reports, for example, must use a different printer for each type of printout.

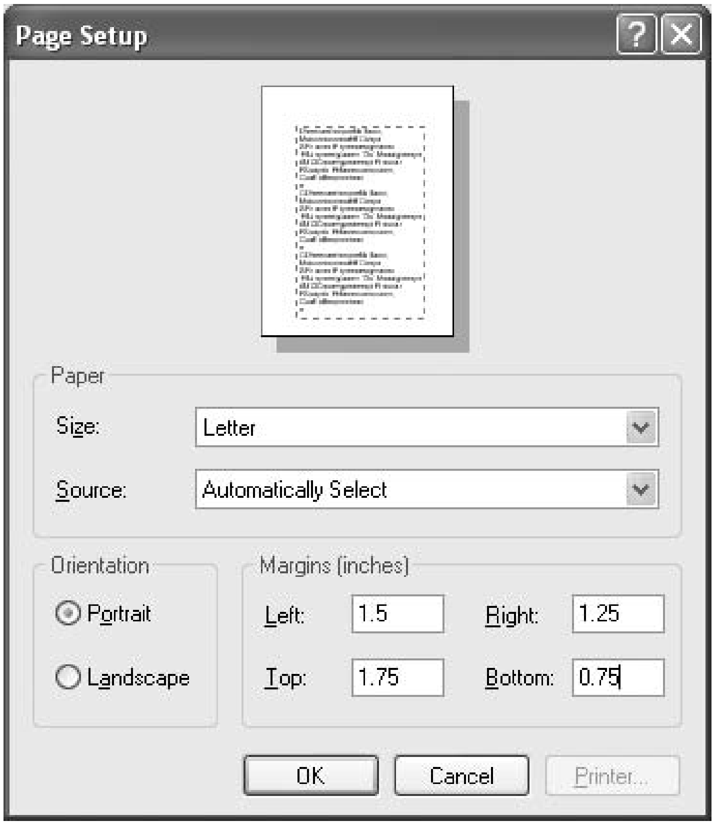
The PageSetupDialog Control

The PageSetupDialog control displays the Page Setup dialog box, which allows users to set up the page (its orientation and margins). The dialog box, shown in Figure 20.3, returns the current page settings in a PageSettings object, which exposes the user-specified settings as properties. These settings don't take effect on their own; you simply read their values and take them into consideration as you prepare the output for the printer from within your code. As you can see, there aren't many parameters to set in this dialog box, but you should display it and take into account the settings specified by the user.

To use this dialog box in your application, drop the PageSetupDialog control on the form and call its `ShowDialog` method from within the application's code. The single property of this control that you'll be using exclusively in your projects is the `PageSettings` property. PageSettings is an object that exposes a number of properties reflecting the current settings of the page (margins and orientation). These settings apply to the entire document. The PrintDocument object has an analogous property: the `DefaultPageSettings` property. After the user closes the Page Setup dialog box, we assign its `PageSettings` property to the `DefaultPageSettings` property of the PrintDocument object to make the user-specified settings available to our code. Here's how we usually display the Page Setup dialog box from within our application and retrieve its `Page-Settings` property:

```
PageSetupDialog1.PageSettings = PrintDocument1.DefaultPageSettings
If PageSetupDialog1.ShowDialog() = DialogResult.OK Then
    PrintDocument1.DefaultPageSettings = PageSetupDialog1.PageSettings
End If
```

FIGURE 20.3
The Page Setup dialog box



Size: Letter
Source: Automatically Select
Orientation: Portrait
Left: 1.5 Right: 1.25
Top: 1.75 Bottom: 0.75

Notice that the first line that initializes the dialog box is mandatory. If you attempt to display the dialog box without initializing its `PageSettings` property, an exception will be thrown. You will learn the properties of the PageSettings object later in this chapter, and we'll use it in most of the examples of this chapter. You can also create a new PageSettings object, set its properties, and then use it to initialize the Page Setup dialog box.

The statements that manipulate the printing objects can get fairly lengthy. It's common to use the `With` structure to make the statements shorter. The preceding code segment can also be coded as follows:

```
With PageSetupDialog1
   .PageSettings = PrintDocument1.DefaultPageSettings
   If .ShowDialog() = DialogResult.OK Then _
               PrintDocument1.DefaultPageSettings = .PageSettings
End With
```

To change the default margins in the Page Setup dialog box before displaying it, you can create a new PageSettings object and set its `Margins` property as shown in the following code segment. The margins are specified in the default coordinate system, and they correspond to 1.25, 1.75, 1, and 2 inches because the default coordinate system of the page is 1/100 of an inch.

```
Dim PS As New System.Drawing.Printing.PageSettings
PS.Margins.Left = 125
PS.Margins.Right = 175
PS.Margins.Top = 100
PS.Margins.Bottom = 200
PageSetupDialog1.PageSettings = PS
If PageSetupDialog1.ShowDialog() = Windows.Forms.DialogResult.OK Then
    PrintDocument1.DefaultPageSettings = PageSetupDialog1.PageSettings
    PrintDocument1.Print()
End If
```

Different Locales Use Different Units

If the application is running on a computer with a European locale, the margins will be converted to tenths of a millimeter (or hundredths of a centimeter). The values of the previous example will be mapped to 12.5, 17.5, 10, and 20 millimeters. The default coordinates of the page, however, are always expressed in hundredths of an inch. If you request the values of the `Margins.Left` and `Margins.Right` properties of the PrintDocument1.DefaultPageSettings object, you'll get back the values 49 and 69. 49/100 of an inch corresponds (practically) to half an inch, which is the same as 12.5 millimeters (there are 25.4 millimeters in an inch). The value 125 corresponds to one and a quarter inches if the target computer uses the American locale, but only half an inch if the computer is using a European locale. The PageSetupDialog control, however, will display the appropriate units in the Margins section (inches or millimeters).

The PrintPreviewDialog Control

Print Preview is another dialog box that displays a preview of the printed document. It exposes a lot of functionality and allows users to examine the output and, optionally, to send it to the printer. The Print Preview dialog box, shown in Figure 20.4, is made up of a preview pane, where you can display one or more pages at the same time at various magnifications, and a toolbar.

The buttons on the toolbar allow users to select the magnification, set the number of pages that will be displayed on the preview pane, move to any page of a multipage printout, and send the preview document to the printer.

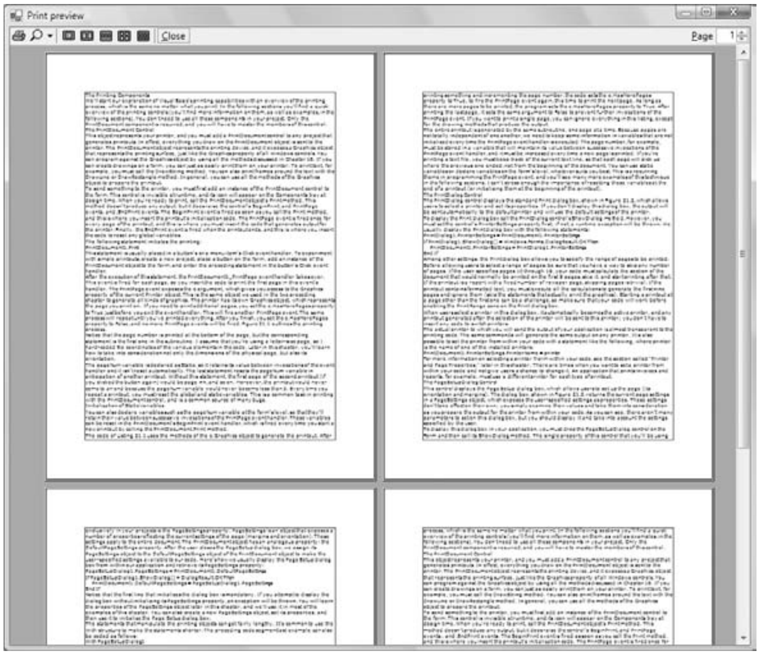


FIGURE 20.4
The Print Preview dialog box

After you write the code to generate the printout, you can direct it to the PrintPreviewDialog control. You don't have to write any additional code; just place an instance of the control on the form and set its `Document` property to the PrintDocument control on the form. Then show the control instead of calling the `Print` method of the PrintDocument object:

```
PrintPreviewDialog1.Document = PrintDocument1
PrintPreviewDialog1.ShowDialog
```

After the execution of these two lines, the PrintDocument control takes over. It fires the `Print-Page` event as usual, but it sends its output to the Print Preview dialog box, not to the printer. The dialog box contains a Print button, which the user can click to send the document being previewed to the printer. The exact same code that generated the preview document will print the document on the printer.

The PrintPreviewDialog control can save you a lot of paper and toner when you test your printing code, because you don't have to print every page to see what it looks like. Because the same code generates both the preview and the actual printed document, and the Print Preview option adds a professional touch to your application, there's no reason why you shouldn't add this feature to your projects.

YOU CAN'T USE THE PRINTPREVIEWDIALOG CONTROL WITHOUT A PRINTER

The PrintPreviewDialog control generates output that would normally be printed by the default printer (or the printer selected in the Print dialog box). If this printer is a networked printer that your computer can't access at the time, the PrintPreview dialog box will not be displayed. Instead, an exception will be thrown, which you must catch from within your code.

Of course, this control is no substitute for actual printing tests. You should also try to generate physical printouts (on several types of printers, if possible) to uncover any problems with your printing code before your customers do. For example, most printers can't print near their page edges, but this isn't a problem for the PrintPreviewDialog control. If you print near the edges, the printout will appear fine on the preview pane, but some unexpected cropping might occur on the hard copy. Some black-
and-white printers might translate colors to gray shades poorly, and what appears light gray on the monitor during a preview might show as black on a printout.

I mentioned earlier that the PageSettings class exposes the `Margins` property, which returns the margins specified by the user on the PageSetupDialog control. The PageSettings class also exposes the `HardMarginX` and `HardMarginY` properties, which return the width and height of the unprintable area of the page, respectively. For my ink-jet printer, the two values are 25 and 11 (in hundredths of an inch). Use these two properties in your code to make sure that the margins specified by the user are at least equal to the page's hard margins.

The first example of this chapter (refer to Listing 20.1) prints three simple pages to the printer. To redirect the output of the program to the PrintPreview control, add an instance of the Print-Preview control to the form and replace the statement that calls the `PrintDocument1.Print` method in the button's `Click` event handler with the following statements:

```
PrintPreviewDialog1.Document = PrintDocument1
PrintPreviewDialog1.ShowDialog
```

Run the project, and this time you preview the document on your monitor. If you're satisfied with its appearance, you can click the Print button to send the document to the printer.

To avoid runtime errors, you can use the following exception handler, whether you print directly to the printer or you're displaying a printout preview:

```
Try
   PrintPreviewDialog1.Document = PrintDocument1
   PrintPreviewDialog1.ShowDialog
Catch exc As Exception
   MsgBox "The printing operation failed" & vbCrLf & exc.Message
End Try
```

Printer and Page Properties

Before you can generate a printout, you must retrieve the settings of the current printer and page, and this is a good place to present the members of these two objects because we'll use them extensively in the examples of the following sections. The properties of these two items are reported to your application through the PrinterSettings and the PageSettings objects. The PageSettings object is a property of the PrintPageEventArgs class, and you can access it through the e argument of the PrintPage event handler. The DefaultPageSettings property of the PrintDocument component exposes the current page's settings.

The PrinterSettings object is a property of the PrintDocument object, as well as a property of the PageSetupDialog and PrintDialog controls. Finally, one of the properties exposed by the PageSettings object is the PrinterSettings object. These two objects provide all the information you might need about the selected printer and the current page through the properties listed in Tables 20.1 and 20.2.

TABLE 20.1: The Properties of the PageSettings Object

PROPERTY	DESCRIPTION
Bounds	Returns the bounds of the page (Bounds.Width and Bounds.Height). If the current orientation is landscape, the width is larger than the height.
Color	Returns, or sets, a True/False value that indicates whether the current page should be printed in color. On a monochrome printer, this property is always False.
Landscape	A True/False value that indicates whether the page is printed in landscape or portrait orientation.
Margins	The margins for the current page (Margins.Left, Margins.Right, Margins.Bottom, and Margins.Top).
PaperSize	The size of the current page (PaperSize.Width and PaperSize.Height).
PaperSource	The page's paper tray.
PrinterResolution	The printer's resolution for the current page.
PrinterSettings	This property returns, or sets, the printer settings associated with the page. For more information on the PrinterSettings object and the properties it exposes, see Table 20.2.

Retrieving the Printer Names

To retrieve the names of the installed printers, use the InstalledPrinters collection of the PrinterSettings object. This collection contains the names of the printers as strings, and you can access them with the following loop:

```
Dim i As Integer
With PrintDocument1.PrinterSettings.InstalledPrinters
   For i = 0 To .Count - 1
      Debug.WriteLine(.Item(i))
   Next
End With
```

TABLE 20.2: The Members of the PrinterSettings Object

MEMBER	**DESCRIPTION**
`InstalledPrinters`	This method retrieves the names of all printers installed on the computer. The same printer names also appear in the Print dialog box, in which the user can select any one of them.
`CanDuplex`	A read-only property that returns a True/False value indicating whether the printer supports double-sided printing.
`Collate`	Another read-only property that returns a True/False value indicating whether the printout should be collated.
`Copies`	This property returns the requested number of copies of the printout.
`DefaultPageSettings`	This property is the PageSettings object that returns, or sets, the default page settings for the current printer.
`Duplex`	This property returns, or sets, the current setting for double-sided printing.
`FromPage, ToPage`	The printout's starting and ending pages, as specified in the Print dialog box by the user.
`IsDefaultPrinter`	Returns a True/False value that indicates whether the selected printer (the one identified by the `PrinterName` property) is the default printer. Note that selecting a printer other than the default one in the Print dialog box doesn't change the default printer.
`IsPlotter`	Returns a True/False value that indicates whether the printer is a plotter.
`IsValid`	Returns a True/False value that indicates whether the `PrinterName` corresponds to a valid printer.
`LandscapeAngle`	Returns an angle, in degrees, by which the portrait orientation must be rotated to produce the landscape orientation.
`MaximumCopies`	Returns the maximum number of copies that the printer allows you to print at a time.
`MaximumPage`	Returns, or sets, the largest value that the `FromPage` and `ToPage` properties can have.
`MinimumPage`	Returns, or sets, the smallest value that the `FromPage` and `ToPage` properties can have.
`PaperSizes`	Returns all the paper sizes that are supported by this printer.

(CONTINUED)

Table 20.2: The Members of the PrinterSettings Object *(CONTINUED)*

Member	Description
`PaperSources`	Returns all the paper source trays on the selected printer.
`PrinterName`	Returns, or sets, the name of the printer to use.
`PrinterResolutions`	Returns all the resolutions that are supported by this printer.
`PrintRange`	Returns, or sets, the numbers of the pages to be printed, as specified by the user. When you set this property, the value becomes the default setting when the Print dialog box is opened.
`SupportsColor`	Returns a True/False value that indicates whether this printer supports color printing.
`CreateMeasurement-Graphics`	Returns a Graphics object that contains printer information you can use in the `PrintDocument.Print` event handler.

These statements will produce output such as the following when executed:

```
Fax
HPLaser
\\TOOLKIT\XEROX
```

The first two printers are local (Fax isn't even a printer; it's a driver for the fax and it's installed by Windows). The last printer's name is XEROX, and it's a network printer connected to the TOOLKIT workstation.

You can also change the current printer by setting the `PrinterName` property of the `PrinterSettings` property with either of the following statements:

```
PrintDocument1.PrinterSettings.PrinterName = "HPLaser"
PrintDocument1.PrinterSettings.PrinterName = _
    PrintDocument1.PrinterSettings.InstalledPrinters(1)
```

Another property that needs additional explanation is the `PrinterResolution` property. The `PrinterResolution` property is an object that exposed provides the `Kind` property, which returns, or sets, the current resolution of the printer, and its value is one of the `PrinterResolutionKind` enumeration's members: *`Custom`*, *`Draft`*, *`High`*, *`Low`*, and *`Medium`*. To find out the exact horizontal and vertical resolutions, read the `X` and `Y` properties of the `PrinterResolution` property. When you set the `PrinterResolutionKind` property to *`Custom`*, you must specify the `X` and `Y` properties.

Page Geometry

Printing on a page is similar to generating graphics onscreen. Like the drawing surface on the monitor (the client area), the page on which you're printing has a fixed size and resolution. The most challenging aspect of printing is the calculation of the coordinates and dimensions of each graphic element on the page. In business applications, the most common elements are strings (rendered in various fonts, styles, and sizes), lines, and rectangles, which are used as borders for tabular data.

Although you can print anywhere on the page, we usually print one element at a time, calculate the space it takes on the page, and then print the next element next to or below it. Printing code makes heavy use of the `MeasureString` method, and nearly all the examples in this chapter use this method.

The printable area is determined by the size of the paper you're using, and in most cases it's 8.5 × 11 inches (keep in mind that most printers can't print near the edge of the page). Printed pages have a margin on all four sides, and users can set a different margin on each side through the Page Setup dialog box. Your program should confine its printing within the specified margins.

To access the current page's margins, use the `Margins` property of the PrintDocument1.DefaultPageSettings object. This object exposes the `Left`, `Right`, `Top`, and `Bottom` properties, which are the values of the four margins. The margins, as well as the page coordinates, are expressed in hundredths of an inch. The width of a standard letter-sized page, for example, is 8,500 units, and its height is 11,000 units. Of course, you can use noninteger values for even greater granularity, but you won't see two straight lines printed at less than one-hundredth of an inch apart. You can use other units, which are all members of the `PageUnit` enumeration (refer to Chapter 18). In the examples of this chapter, I'm using the default units (1/100 of an inch).

Another property exposed by the DefaultSettings object is the `PageSize` property, which represents the dimensions of the page. The width and height of the page are given by the following expressions:

```
PrintDocument1.DefaultPageSettings.PaperSize.Width
PrintDocument1.DefaultPageSettings.PaperSize.Height
```

The top of the page is at coordinates (0, 0), which correspond to the top-left corner of the page. We never actually print at this corner. The coordinates of the top-left corner of the printable area of the page are given by the following expressions:

```
PrintDocument1.DefaultPageSettings.Margins.Top
PrintDocument1.DefaultPageSettings.Margins.Left
```

Now that you have seen how to use the printing components, their basic properties, and the page's geometry, you can look at some examples that demonstrate how to generate practical printouts.

VB 2008 at Work: The SimplePrintout Project

Let's put the information of the preceding paragraphs together to build a simple application that prints a string at the top-left corner of the page (the origin of the page) and a rectangle that delimits the page's printable area. To print something, start by dropping the PrintDocument object on your form. Then place a button on the form and enter the following statement in its `Click` event handler:

```
PrintDocument1.Print()
```

This statement tells the PrintDocument object that you're ready to print. The PrintDocument object will fire the `BeginPrint` event, in which you can place any initialization code (reset the variables that must maintain their value between consecutive invocations of the PrintPage event handler, for example). Then, it will fire the PrintPage event, whose definition is the following:

```
Private Sub PrintDocument1_PrintPage( _
         ByVal sender As Object, _
         ByVal e As System.Drawing.Printing.PrintPageEventArgs) _
         Handles PrintDocument1.PrintPage
End Sub
```

As implied by its name, the PrintPage event is fired once for each page. You must place the VB code required to produce the desired output in this event's handler. To access the page in the printer from within the PrintPage event's handler, use the e.Graphics property, which is a Graphics object. Anything you draw on this object is printed on paper.

To print a string at the page's top-left corner, call the Graphics object's DrawString method, as shown here:

```
Dim pFont As Font
pFont = New Font("Comic Sans MS", 20)
e.Graphics.DrawString("ORIGIN", pFont, Brushes.Black, 0, 0)
```

The last two arguments of the DrawString method are the coordinates of a point where the string will be printed. The string is printed right below the origin, so that it's visible. If you attempt to print a string at the bottom-right corner of the page, the entire string will fall just outside the page, and no visible output will be produced. The coordinates passed to the DrawString method are the coordinates of the upper-left corner of a box that encloses the specified string.

No matter what your default printer is, it's highly unlikely that it's been set to no margins. The page's margins aren't enforced by the PrintDocument object; you must respect them from within your code because it is possible to print anywhere on the page. To take into consideration the page's margins, change the coordinates from (0, 0) to the left and top margins.

You can also use the other members of the Graphics object to generate graphics. The following statement will render the text on the page by using an anti-alias technique (anti-aliased text looks much smoother than text rendered with the default method):

```
e.Graphics.TextRenderingHint = Drawing.Text.TextRenderingHint.AntiAlias
```

Next, we'll print a rectangle around the area of the page in which we're allowed to print — a rectangle delimited by the margins of the page. To draw this rectangle, we need to know the size of all four margins and the size of the page (obviously). To read (or set) the page's margins, use the PrintDocument1.DefaultPageSettings.Margin object, which provides the Left, Right, Top, and Bottom properties. We're also going to need the dimensions of the page, which we can read through the Width and Height properties of the PrintDocument1.DefaultPageSettings.PaperSize object. The four margins are calculated and stored in four variables via the following statements:

```
Dim Lmargin, Rmargin, Tmargin, Bmargin As Integer
With PrintDocument1.DefaultPageSettings.Margins
   Lmargin = .Left
   Rmargin = .Right
   Tmargin = .Top
   Bmargin = .Bottom
End With
```

The rectangle we want to draw should start at the point (`Lmargin`, `Tmargin`) and extend *`PrintWidth`* units to the right and *`PrintHeight`* units down. These two variables are the width and height of the page minus the respective margins, and they're calculated with the following statements:

```
Dim PrintWidth, PrintHeight As Integer
With PrintDocument1.DefaultPageSettings.PaperSize
   PrintWidth = .Width - Lmargin - Rmargin
   PrintHeight = .Height - Tmargin - Bmargin
End With
```

Then insert the following statements in the `PrintPage` event handler to draw the rectangle:

```
Dim R As Rectangle
R = New Rectangle(Lmargin, Tmargin, PrintWidth, PrintHeight)
e.Graphics.DrawRectangle(Pens.Black, R)
```

The printing takes place from within the `PrintPage` event handler, which is shown in Listing 20.2. The event handler contains all the statements presented in the previous paragraphs and a few comments.

Listing 20.2: Generating a Simple Printout

```
Private Sub PrintDocument1_PrintPage(ByVal sender As Object,_
                ByVal e As System.Drawing.Printing.PrintPageEventArgs) _
                Handles PrintDocument1.PrintPage
' Turn on antialias for text
   e.Graphics.TextRenderingHint = _
Drawing.Text.TextRenderingHint.AntiAlias
' Print a string at the origin
   Dim pFont As Font
   pFont = New Font("Comic Sans MS", 20)
   e.Graphics.DrawString("ORIGIN", pFont, Brushes.Black, 0, 0)
' Read margins into local variables
   Dim Lmargin, Rmargin, Tmargin, Bmargin As Integer
   With PrintDocument1.DefaultPageSettings.Margins
      Lmargin = .Left
      Rmargin = .Right
      Tmargin = .Top
      Bmargin = .Bottom
   End With
' Calculate the dimensions of the printable area
   Dim PrintWidth, PrintHeight As Integer
   With PrintDocument1.DefaultPageSettings.PaperSize
      PrintWidth = .Width - Lmargin - Rmargin
      PrintHeight = .Height - Tmargin - Bmargin
   End With
```

```
    ' Now print the rectangle
      Dim R As Rectangle
      R = New Rectangle(Lmargin, Tmargin, PrintWidth, PrintHeight)
      e.Graphics.DrawRectangle(Pens.Black, R)
    End Sub
```

VB 2008 at Work: The PageSettings Project

In this section, we'll write a more elaborate application to print a rectangle bounded by the margins of the page as before. In addition to printing the rectangle, the application also prints four strings, one in each margin, with different orientations (as seen in Figure 20.5). The project that generated the output is called PageSettings, and it also demonstrates how to display the Page Setup dialog box from within your code and then generate a printout according to the settings on this dialog box.

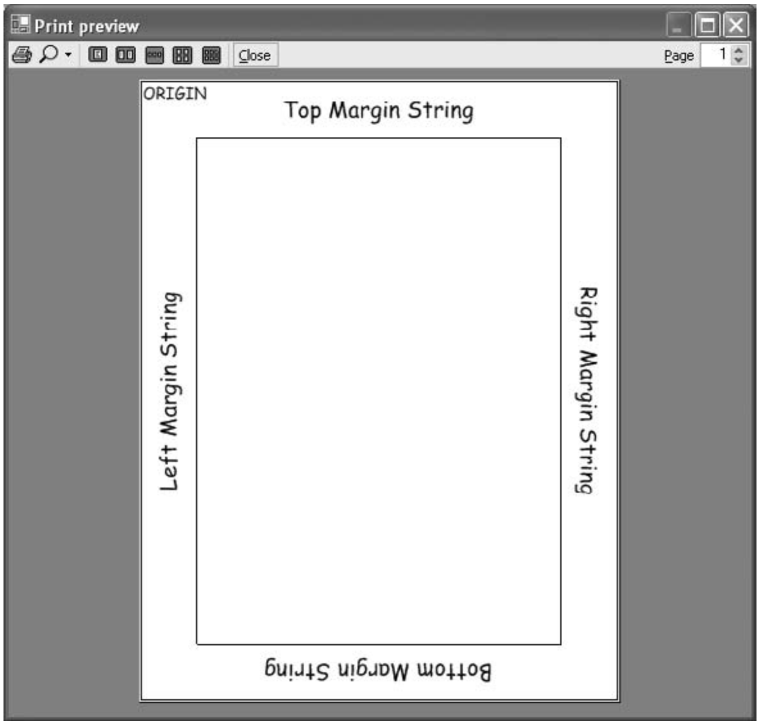


FIGURE 20.5
The output of the PageSettings project

You saw the statements that print a rectangle enclosing the printable area of the page. Printing the labels is a bit involved. Because the four strings appear in all four orientations, some rotation transformation is involved. We'll discuss the code for printing the captions later. For now, let's examine the PageSetupDialog control and how you take into consideration the settings in this dialog box from within your code.

Setting Up the Page

To display the Page Setup dialog box, first place an instance of the PageSetupDialog control on your form. Then set its `PageSettings` property to a PageSettings object that contains the default settings for the printer. We usually set this property to the `DefaultPageSettings` property of the PrintDocument object, although you can create a new PageSettings object and set its properties from within your code. Finally, display the dialog box by calling its `ShowDialog` method:

```
PageSetupDialog1.PageSettings = PrintDocument1.DefaultPageSettings
If PageSetupDialog1.ShowDialog() = Windows.Forms.DialogResult.OK Then
    PrintDocument1.DefaultPageSettings = PageSetupDialog1.PageSettings
End If
```

Upon return, we assign the `PageSettings` property of the control to the `DefaultPageSettings` property of the *`PrintDocument1`* control. Now, we must take into consideration the settings specified in the dialog box from within the `PrintPage` event's code. The area on the page in which we must restrict our output is a rectangle with its top-left corner at the left and top margins, and its dimensions being the width and height of the page (less the corresponding margins). The following statements set up a few variables to hold the page's dimensions:

```
Dim PrintWidth, PrintHeight As Single
Dim PageWidth, PageHeight As Single
With PrintDocument1.DefaultPageSettings.PaperSize
    PrintWidth = .Width - LMargin - RMargin
    PrintHeight = .Height - TMargin - BMargin
    PageWidth = .Width
    PageHeight = .Height
End With
```

A few additional statements are required if the user changes the orientation of the page. When you're printing in landscape mode, the size of the paper doesn't change. If you examine the `Width` and `Height` properties of the PaperSize object, you'll realize that the page is always taller than it is wide. This means that we must swap the width and height from within our code. The margins, however, remain the same. Notice that as you change the orientation of the page in the Page Setup dialog box, the margins are swapped automatically (the left and right margins become top and bottom, respectively).

To find out whether the user has changed the page's orientation, examine the `Landscape` property of the DefaultPageSettings object. If this property is True, it means that the user wants to print in landscape mode, and you must swap the page's width and height. The following statements calculate the dimensions of the page area within the margins when the orientation is set to landscape:

```
If PrintDocument1.DefaultPageSettings.Landscape Then
   With PrintDocument1.DefaultPageSettings.PaperSize
      PrintWidth = .Height - Tmargin - Bmargin
      PrintHeight = .Width - Rmargin - Lmargin
```

```
            PageWidth = .Height
            PageHeight = .Width
         End With
      End If
```

Printing the Labels

Now we can focus on the code that prints the captions in the space of the four margins, which is considerably more elaborate. The top margin's caption isn't rotated; it's printed at the default orientation. The caption in the right margin is rotated by 90 degrees, and the caption in the bottom margin is rotated by 180 degrees. The caption in the left margin is rotated by −90 degrees. These rotations take place around the origin, so the labels must also be moved to their places with a translation transformation. Let's look at the code that prints the Right Margin String caption, shown in Listing 20.3.

Listing 20.3: Printing a Caption in the Right Margin

```
strWidth = e.Graphics.MeasureString(RMarginCaption, pFont).Width
strHeight = e.Graphics.MeasureString(RMarginCaption, pFont).Height
X = PageWidth - (Rmargin - strHeight) / 2
Y = TMargin + (PrintHeight - strWidth) / 2
e.Graphics.ResetTransform()
e.Graphics.TranslateTransform(X, Y)
e.Graphics.RotateTransform(90)
e.Graphics.DrawString(RMarginCaption, pFont, Brushes.Black, 0, 0)
```

First, we calculate the string's width and height by using the `MeasureString` method and store them in the *`strWidth`* and *`strHeight`* variables. The string will be rotated by 90 degrees before being printed. The rotation alone would place the string just outside the left margin, so we must translate it to the right. The amount of the translation is the page's width minus half the difference between the string's height and the right margin. Translating the caption by the width of the page would bring it to the very right edge of the paper. To center it in the right margin, we must split the difference of the string's height from the right margin on either side of the string. We're using the string's height in calculating the x-coordinate and the string's width in calculating the y-coordinate because after the string is rotated by 90 degrees, the width and height will be swapped. *`X`* and *`Y`* are the amounts by which the string must be moved along the horizontal and vertical axes. The rotation of the string will be performed by a rotation transformation (for more information on transformations, see Chapter 18). Because transformations are cumulative, the code resets any existing transformations and applies two new ones.

Then, the `DrawString` method is called to print the string. The `DrawString` method draws the string at the point (0, 0), but the two transformations will place it at the proper location. This is the simplest method for printing transformed strings (or any other graphic element): Set up the appropriate transformation(s) and then draw the string at the origin.

The code for placing the other three captions is quite analogous. It uses the proper translation and rotation transformations, and the only complication is the calculation of the coordinates of the translation transformation. The listing of the `PrintPage` event handler of the PageSettings project is fairly lengthy. Listing 20.4 shows the code that prints the caption in the right margin.

Listing 20.4: Printing the Rectangle and the Margin Captions

```
Private Sub PrintDocument1_PrintPage(...) _
                 Handles PrintDocument1.PrintPage
   Dim R As Rectangle
   Dim strWidth, strHeight As Integer
   Dim pFont As Font
   pFont = New Font("Comic Sans MS", 20)
   e.Graphics.DrawString("ORIGIN", pFont, Brushes.Black, 0, 0)
   pFont = New Font("Comic Sans MS", 40)
   Dim X, Y As Integer
   Dim TMarginCaption As String = "Top Margin String"
   Dim LMarginCaption As String = "Left Margin String"
   Dim RMarginCaption As String = "Right Margin String"
   Dim BMarginCaption As String = "Bottom Margin String"
   Dim LMargin, RMargin, TMargin, BMargin As Integer
   With PrintDocument1.DefaultPageSettings.Margins
      LMargin = .Left
      RMargin = .Right
      TMargin = .Top
      BMargin = .Bottom
   End With
   Dim PrintWidth, PrintHeight, PageWidth, PageHeight As Integer
   With PrintDocument1.DefaultPageSettings.PaperSize
      PrintWidth = .Width - LMargin - RMargin
      PrintHeight = .Height - TMargin - BMargin
      PageWidth  = .Width
      PageHeight = .Height
   End With
   If PrintDocument1.DefaultPageSettings.Landscape Then
      With PrintDocument1.DefaultPageSettings.PaperSize
         PrintWidth = .Height - TMargin - BMargin
         PrintHeight = .Width - RMargin - LMargin
         PageWidth = .Height
         PageHeight = .Width
      End With
   End If
' Draw rectangle
   R = New Rectangle(LMargin, TMargin, PageWidth - LMargin - RMargin, _
                    PageHeight - BMargin - TMargin)
   e.Graphics.DrawRectangle(Pens.Black, R)
   strWidth = e.Graphics.MeasureString(RMarginCaption, pFont).Width
   strHeight = e.Graphics.MeasureString(RMarginCaption, pFont).Height
   X = PageWidth - (RMargin - strHeight) / 2
   Y = TMargin + (PrintHeight - strWidth) / 2
   e.Graphics.ResetTransform()
```

```
        e.Graphics.TranslateTransform(X, Y)
        e.Graphics.RotateTransform(90)
        e.Graphics.DrawString(RMarginCaption, pFont, Brushes.Black, 0, 0)
```

As always, you must call the PrintDocument object's `Print` method for this event handler to be activated. You can use the `Print` method of the PrintDocument object, but the sample project uses the PrintPreviewDocument object to display a preview of the printout. Listing 20.5 shows the code behind the button on the form.

Listing 20.5: The Print Button

```
Private Sub Button1_Click(...) _
            Handles Button1.Click
    Try
        PrintPreviewDialog1.Document = PrintDocument1
        PageSetupDialog1.PageSettings = _
                   PrintDocument1.DefaultPageSettings
        If PageSetupDialog1.ShowDialog() = _
                   Windows.Forms.DialogResult.OK Then
            PrintDocument1.DefaultPageSettings = _
            PageSetupDialog1.PageSettings
            PrintPreviewDialog1.ShowDialog()
        End If
    Catch exc As Exception
        MsgBox("Printing Operation Failed" & vbCrLf & _
                exc.Message)
    End Try
End Sub
```

The code uses an exception handler to prevent the program from crashing with a runtime exception if there's a problem with the printer. The application should work if there's a default printer; it will fail to generate a preview only if the default printer is a network printer and you have no access to it at the time.

The first statement sets up the PrintPreview control by setting its `Document` property to the PrintDocument object. The second statement assigns the default page settings to the PageSetup-Dialog control, and the following statement displays the Page Setup dialog box. After the user has specified the desired settings and closed the dialog box, the new settings are assigned to the Print-Document object's `DefaultPageSettings` property. The last statement displays the Print Preview dialog box. This statement initiates the printing process, which sends its output to the preview pane instead of the printer. That's all it takes to add a preview feature to your application.

If you feel uncomfortable with the transformations, especially the rotation transformation, Figure 20.6 shows what happens to a string when it's rotated in all four directions around the origin. The origin — the point with coordinates (0, 0) — is where the two axes meet.

The statements in the `PrintPage` event handler rotate the string *GDI + Graphics* around the origin by 90, 180, and 270 degrees. The numbers in parentheses indicate the angle of rotation for each string. Of course, I couldn't print to the left of the origin or above the origin, so I had to rotate the translated string by 50 percent of the page's width to the right and 50 percent of the page's height down, to appear at the middle of the page. The two axes were also translated by the same amounts in the two directions. This illustration's purpose is to help you visualize how the string is rotated around the origin. Besides the string itself, the enclosing rectangle is also printed. This is the rectangle returned by the `MeasureString` method, subject to the same transformations as the string it encloses. To examine the code that produced Figure 20.6, open the RotatedStrings sample project in Visual Studio; the printing code is well documented in the code, and you should be able to understand and experiment with it.

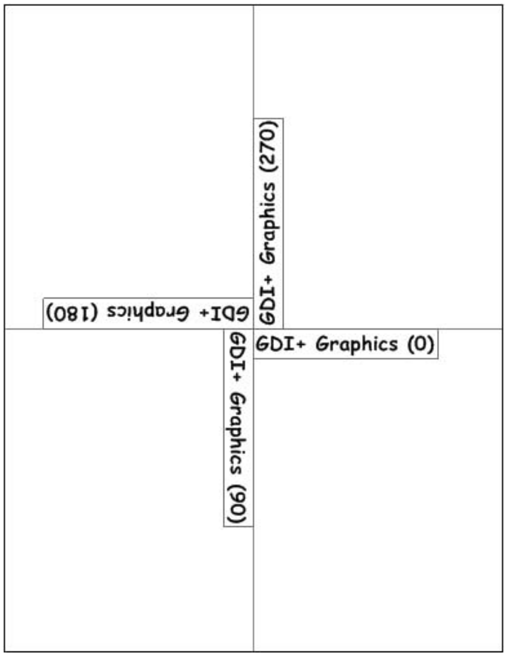


FIGURE 20.6
Rotating a string around the origin

Practical Printing Examples

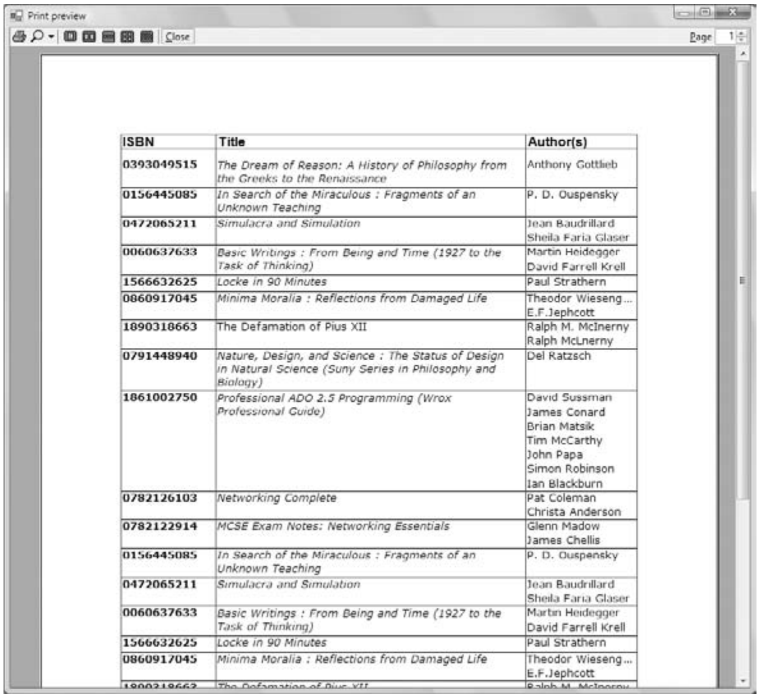
In principle, using the Framework's printing components is straightforward. Depending on the type of printout you want to generate, however, the code of the `PrintPage` event handler can get quite complicated. Because there are no techniques that you can apply to all situations, I included a few typical examples to demonstrate how to use the same objects to perform very different tasks. The first example demonstrates how to print tabular reports, which is the most common report type for business applications. A tabular report has the form of a grid, with columns of different widths and rows of different heights.

The second example is the printing of text, and even if this is the least exciting type of printout, you should be able to send text to the printer. As it turns out, it's not a trivial operation. The last example prints bitmaps, probably the simplest type of printout. The only challenge with printing bitmaps is that you might have to reduce the size of the bitmap to make it fit in the width or the height of the page, or a rectangular area within the page.

Printing Tabular Data

The printing operation you'll be using most often in typical business applications that require custom printing is that of tabular data. Figure 20.7 shows an example of a printout with tabular data. This printout was generated by the PrintTable project.

FIGURE 20.7
Using the PrintTable application to print data in a tabular arrangement



ISBN	Title	Author(s)
0393049515	*The Dream of Reason: A History of Philosophy from the Greeks to the Renaissance*	Anthony Gottlieb
0156445085	*In Search of the Miraculous : Fragments of an Unknown Teaching*	P. D. Ouspensky
0472065211	*Simulacra and Simulation*	Jean Baudrillard Sheila Faria Glaser
0060637633	*Basic Writings : From Being and Time (1927 to the Task of Thinking)*	Martin Heidegger David Farrell Krell
1566632625	*Locke in 90 Minutes*	Paul Strathern
0860917045	*Minima Moralia : Reflections from Damaged Life*	Theodor Wieseng... E.F.Jephcott
1890318663	The Defamation of Pius XII	Ralph M. McInerny Ralph McLnerny
0791448940	*Nature, Design, and Science : The Status of Design in Natural Science (Suny Series in Philosophy and Biology)*	Del Ratzsch
1861002750	*Professional ADO 2.5 Programming (Wrox Professional Guide)*	David Sussman James Conard Brian Matsik Tim McCarthy John Papa Simon Robinson Ian Blackburn
0782126103	*Networking Complete*	Pat Coleman Christa Anderson
0782122914	*MCSE Exam Notes: Networking Essentials*	Glenn Madow James Chellis
0156445085	*In Search of the Miraculous : Fragments of an Unknown Teaching*	P. D. Ouspensky
0472065211	*Simulacra and Simulation*	Jean Baudrillard Sheila Faria Glaser
0060637633	*Basic Writings : From Being and Time (1927 to the Task of Thinking)*	Martin Heidegger David Farrell Krell
1566632625	*Locke in 90 Minutes*	Paul Strathern
0860917045	*Minima Moralia : Reflections from Damaged Life*	Theodor Wieseng... E.F.Jephcott

The ISBN column contains a 10-character string, and it's quite simple to handle. All you have to do is make sure that the ISBN will fit in the corresponding column. If you allow the user to select the font at runtime and you can't set a fixed width for this column, you should print only as many characters as will fit in the reserved width. In this example, we won't do anything special with the ISBN column. You can also retrieve the width of a 10-character string in the specific font and use this value (plus a small margin) as the column's width.

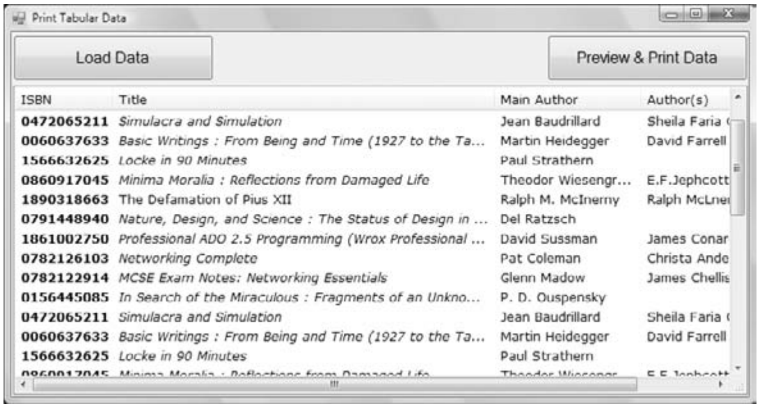
The Title column has a variable length, and you might have to break long titles into two or more printed lines — this is the real challenge of the application. As you recall from Chapter 18, the `DrawString` method can print a string in a rectangle you pass as an argument. The width of this rectangle must be the same as the width of the Title column. The height of the rectangle should be enough for the entire text to fit in it. In our code, we'll use a rectangle with the appropriate width and adequate height to make sure that the entire title will be printed. Alternatively, you can trim the title if it's too long, but there's no point in trimming substantial information.

The last intricacy of this application is the Author(s) column. Each book might have no authors, one author, or more, and we'll print each author on a separate line. The total height of each row depends on the height of the Title or Author(s) cell: Note in Figure 20.7 that the height of some lines is determined by the height of the Title cell, while the height of others is determined by the height of the Author(s) cell. We must keep track of the height of these two cells and move down accordingly before printing the following row. Where the height of the Author(s) cells is determined by the number of authors (we'll print each author on a single line and assume that the name does not exceed the width of the page), we must provide the code to break the title into multiple lines. If an author's name does not fit in a single line, it will be truncated and an ellipsis will appear in the place of the missing characters.

So, where does the data come from? It could come from a text file, an XML document, or a database. It doesn't really make a difference, as long as you can access one row at a time and extract its fields. For the purposes of this example, and because we haven't discussed databases yet, I'm using a ListView control to store the data. The ListView control is populated with the Load Data button on the form of Figure 20.8 (the project's main form). Each book is a different item in the ListView, and the various fields are subitems. Each item's `Text` property is the book's ISBN, and the remaining fields are stored as subitems. I took sample data from an online bookstore and, in some cases, edited their titles to make them long or added fictitious authors. Here are the statements that populate the ListView control with the first two items:

```
Dim BookItem As New ListViewItem
BookItem.Text = "0393049515"
BookItem.SubItems.Add( _
                  "The Dream of Reason:
                  A History of Philosophy from
                  the Greeks to the Renaissance")
BookItem.SubItems.Add("Anthony Gottlieb")
ListView1.Items.Add(BookItem)

BookItem = New ListViewItem
BookItem.Text = "0156445085"
BookItem.SubItems.Add("In Search of the Miraculous :
                       Fragments of an Unknown Teaching ")
BookItem.SubItems.Add("P. D. Ouspensky")
ListView1.Items.Add(BookItem)
```

FIGURE 20.8
The PrintTable project's main form



ISBN	Title	Main Author	Author(s)
0472065211	*Simulacra and Simulation*	Jean Baudrillard	Sheila Faria (
0060637633	*Basic Writings : From Being and Time (1927 to the Ta...*	Martin Heidegger	David Farrell
1566632625	*Locke in 90 Minutes*	Paul Strathern	
0860917045	*Minima Moralia : Reflections from Damaged Life*	Theodor Wiesengr...	E.F.Jephcott
1890318663	The Defamation of Pius XII	Ralph M. McInerny	Ralph McLne
0791448940	*Nature, Design, and Science : The Status of Design in ...*	Del Ratzsch	
1861002750	*Professional ADO 2.5 Programming (Wrox Professional ...*	David Sussman	James Conar
0782126103	*Networking Complete*	Pat Coleman	Christa Ande
0782122914	*MCSE Exam Notes: Networking Essentials*	Glenn Madow	James Chellis
0156445085	*In Search of the Miraculous : Fragments of an Unkno...*	P. D. Ouspensky	
0472065211	*Simulacra and Simulation*	Jean Baudrillard	Sheila Faria (
0060637633	*Basic Writings : From Being and Time (1927 to the Ta...*	Martin Heidegger	David Farrell
1566632625	*Locke in 90 Minutes*	Paul Strathern	

Notice that the ListView control has four columns (one for the ISBN, one for the title, and two for author names), but you can add as many authors to each title as you wish. Subitems beyond the fourth one are invisible on the ListView control, but they're there. After the list has been populated, you can click the Preview & Print Data button to generate the preview and print the report. The main form of the PrintTable project, populated with the data shown in the sample printout, is shown in Figure 20.8.

Formatting the Cells

The report is generated one row at a time. The vertical coordinate of the current row is stored in the variable *y*, which is incremented accordingly for each new row. This coordinate applies to all the cells of the current row, and if a cell contains multiple lines, the y-coordinate is adjusted accordingly for the following row. The x-coordinate of each column is the same for all rows. These coordinates are calculated at the beginning and don't change from row to row.

Breaking a string into multiple lines isn't trivial. You should include as many words as you can on each line without exceeding the available width. Fortunately, the Graphics object's `MeasureString` method can break a string into the required number of lines to fit the string into a rectangle and report the number of lines. This form of the `MeasureString` method is as follows:

```
Graphics.MeasureString(string, font, size, format, cols, lines)
```

The first argument is the string to be printed, and it will be rendered in the font specified by the second argument. The *size* argument is the width and height of the rectangle in which the string must fit. In our case, the width is that of the cell in which the string must fit. The *format* argument is a StringFormat object that lets you specify various options for printing text (its orientation, for example). You will find more information about this argument in the following section. For the purposes of this example, we'll use the default FormatString object. The last two arguments are the number of characters that will fit across the rectangle and the number of lines the string must be broken into, and they're set by the `MeasureString` method. Even if we don't know the height of the rectangle in advance, we can use an absurdly large value. The `MeasureString` method will tell you how many text lines it needs, and you'll use this value to calculate the height of the rectangle. To calculate the height of the cell in which the title will fit, the program uses the following statements:

```
Dim cols, lines As Integer
e.Graphics.MeasureString(strTitle, tableFont, _
          New SizeF(W2, 100), New StringFormat(), _
          lines, cols)
```

strTitle is a string variable that holds the title, and *tableFont* is the font in which the string will be rendered. *W2* is the width of the second column of the grid, in which the title appears. This is a fixed value, calculated ahead of time. The initial height of the rectangle is 100 pixels, but this value is totally arbitrary. It is possible for a given cell's text to be so long that it will take a page and a half to print. The PrintTable project can't handle similar extreme situations. You will have to provide additional code to handle the overflow of a cell to the following page.

The *lines* and *cols* variables are passed by reference, so they can be set by the `MeasureString` method to the number of lines and number of characters that will fit in the specified rectangle. After we have the number of lines it takes for the title to be printed in the specified width, we can advance the vertical coordinate by the following amount:

```
lines * tableFont.GetHeight(e.Graphics)
```

where *tableFont* is the font we use to print the table. Its GetHeight method returns the height of the font when rendered on the Graphics object passed as an argument. The few statements shown here will take care of breaking long titles into multiple lines, which is the most challenging aspect of the code. The last cell in each row contains a line for each author. The following loop goes through all the authors and prints them, each one on a separate line:

```
For subitm = 2 To ListView1.Items(itm).SubItems.Count - 1
   str = ListView1.Items(itm).SubItems(subitm).Text
   e.Graphics.DrawString(str, tableFont, Brushes.Black, X3, Yc)
   Yc = Yc + tableFont.Height + 2
Next
```

The y-coordinate of the last author is stored in the variable *Yc*. To calculate the y-coordinate of the next row of the table, we compare the *Y* and *Yc* variables and keep the larger value. This value, plus a small displacement, is used as the y-coordinate for the following line. Listing 20.6 is the complete listing of the PrintPage event handler of the PrintTable project.

LISTING 20.6: The *PrintPage* Event Handler of the PrintTable Project

```
Private Sub PrintDocument1_PrintPage( _
          ByVal sender As Object, ByVal e As _
          System.Drawing.Printing.PrintPageEventArgs) _
          Handles PrintDocument1.PrintPage
   Y = PrintDocument1.DefaultPageSettings.Margins.Top + 20
   e.Graphics.DrawString("ISBN", TitleFont, Brushes.Black, X1, Y)
   e.Graphics.DrawString("Title", TitleFont, Brushes.Black, X2, Y)
   e.Graphics.DrawString("Author(s)", TitleFont, Brushes.Black, X3, Y)
   Y = Y + 30
   While itm < ListView1.Items.Count
      Dim str As String
      str = ListView1.Items(itm).Text
      e.Graphics.DrawString(str, tableFont, Brushes.Black, X1, Y)
      str = ListView1.Items(itm).SubItems(1).Text
      Dim R As New RectangleF(X2, Y, W2, 80)
      e.Graphics.DrawString(str, tableFont, Brushes.Black, R)
      Dim lines, cols As Integer
      e.Graphics.MeasureString(str, tableFont, _
           New SizeF(W2, 50), New StringFormat(), _
           cols, lines)
      Dim subitm As Integer, Yc As Integer
      Yc = Y
      For subitm = 2 To ListView1.Items(itm).SubItems.Count - 1
         str = ListView1.Items(itm).SubItems(subitm).Text
         e.Graphics.DrawString(str, tableFont, Brushes.Black, X3, Yc)
         Yc = Yc + tableFont.Height + 2
      Next
      Y = Y + lines * tableFont.Height + 5
      Y = Math.Max(Y, Yc)
```

```
            With PrintDocument1.DefaultPageSettings
                e.Graphics.DrawLine(Pens.Black, _
                      .Margins.Left, Y, PaperSize.Width - _
                      .Margins.Right, Y)
                If Y > 0.95 * (.PaperSize.Height - .Margins.Bottom) Then
                    e.HasMorePages = True
                    Exit Sub
                End If
            End With
            itm = itm + 1
        End While
        e.HasMorePages = False
    End Sub
```

The code uses a few variables that are declared on the form level with the following statements:

```
Dim tableFont, titleFont As Font
Dim X1, X2, X3 As Integer
Dim W1, W2, W3 As Integer
Dim Y As Integer
Dim itm As Integer
```

Setting the Column Widths

Before we can print, we must specify the widths of the columns. Because we know the information we're going to display in each column, we can make a good estimate of the column widths. The first column, in which the ISBN is displayed, starts at the left margin of the page and extends 120 units to the right, which is an adequate width for printing 13 characters. The default unit is 1/100 of an inch, so the ISBN column's width is 1.2 inches. The Title column should take up most of the page's width. In the PrintTable example, I gave 50 percent of the available page width to this column. The remaining space goes to the Author(s) column. You can't use fixed widths for all columns, because you don't know the paper size or the page's orientation. That's why I'm mixing percentages and allow the last column to fill the space to the right edge of the page. The variables *X1*, *X2*, and *X3* are the x-coordinates of the left edge of each column, whereas the variables *W1*, *W2*, and *W3* are the widths of the columns. These variables are set in the Print button's `Click` event handler. Then, the subroutine displays the Print Preview dialog box with the document's preview. Listing 20.7 shows the Print button's `Click` event handler. I'm also using different fonts for the headers and the table's cells.

LISTING 20.7: Setting Up the Columns and Printing the Table

```
    Private Sub Button2_Click(...) _
                       Handles Button2.Click
        PageSetupDialog1.PageSettings = PrintDocument1.DefaultPageSettings
        If PageSetupDialog1.ShowDialog() Then
            PrintDocument1.DefaultPageSettings = PageSetupDialog1.PageSettings
        End If
```

```
      tableFont = New Font("Arial", 8)
      titleFont = New Font("Arial", 12, FontStyle.Bold)
      X1 = PrintDocument1.DefaultPageSettings.Margins.Left
      Dim pageWidth As Integer
      With PrintDocument1.DefaultPageSettings
         pageWidth = .PaperSize.Width - .Margins.Left - .Margins.Right
      End With
      X2 = X1 + 100
      X3 = X2 + pageWidth * 0.5
      W1 = X2 - X1
      W2 = X3 - X2
      W3 = pageWidth - X3
      PrintPreviewDialog1.Document = PrintDocument1
      PrintPreviewDialog1.ShowDialog()
      itm = 0
   End Sub
```

After setting the coordinates and widths of the columns, you can call the `ShowDialog` method of the PrintPreviewDialog control to preview the document. This method fires the `PrintPage` event, where we start printing the report by printing the header of the table via the following statements:

```
Y = PrintDocument1.DefaultPageSettings.Margins.Top + 20
e.Graphics.DrawString("ISBN", titleFont, Brushes.Black, X1, Y)
e.Graphics.DrawString("Title", titleFont, Brushes.Black, X2, Y)
e.Graphics.DrawString("Author(s)", titleFont, Brushes.Black, X3, Y)
Y = Y + 30
```

`titleFont` is a Font object that represents the font we use for the table header and is declared on the form level. The rest of the program uses the *`tableFont`* object, which represents the font in which the table's cells will be rendered.

Then we set up two nested loops. The outer loop goes through all the items on the ListView control, and the inner loop goes through the subitems of the current item. The structure of the two loops is the following:

```
While itm < ListView1.Items.Count
   { print current item }
   For subitm = 2 To ListView1.Items(itm).SubItems.Count - 1
      { print all subitems }
   Next
End While
```

The PrintTable project is based on the assumption that the author names will fit in the specified width. If not, part of the author name will be truncated. Alternatively, you can print the report in landscape mode — you will have to adjust the widths of the Title and Author(s) columns.

The PrintTable project is the starting point for tabular reports, and it demonstrates the core of an application that prints tables. You will have to add a title to each page, a header and a footer for each page (with page numbers and dates), and quite possibly a grid to enclose the cells. Experiment with the PrintTable project by adding more features to it. You can become as creative as you want with this application. I should also bring to your attention the fact that the PrintTable application ends the page when the report's height exceeds 95 percent of the page's printable area. This test takes place after printing each item. If the last title printed on a page has a dozen different authors, it will run over the bottom of the page. You can change this value depending on the type of report and the font you're using. I'm assuming that no row can fit comfortably in 5 percent of the available printable area, so I end the current page when this point it reached. Note that there's no mechanism to prevent the last row from overflowing the bottom margin (not by a whole lot, of course). If your report's cells may contain from 1 to 10 lines of text, you'll have to come up with a more elaborate test for the end-of-page condition. The application's code is adequately commented, and you'll be able to tweak it to your needs.

Because you're printing the contents of a ListView control, you base the widths of the printout's columns on the widths of the columns of the ListView control. A column that takes up 20 percent of the control's width should also take up 20 percent of the width of the form's printable area. This way, you won't have to come up with any arbitrary rules for the column widths.

Using Static Variables

The `PrintPage` event handler produces all the pages, one after the other. These pages, however, are not independent of one another. When you print a long text file, for example, you must keep track of the pages printed so far or the current line. When printing a tabular report, you might have to keep track of the current row. If you set up a variable that keeps track of the current line, you shouldn't reset this variable every time the `PrintPage` event handler is executed. One way to maintain the value of a variable between consecutive calls of the same procedure is to declare it with the `Static` keyword. Static variables maintain their values between calls, unlike the private variables.

In the PrintTable project, I used the *`itm`* variable to keep track of the item being printed. By making the variable *`itm`* static, we're sure that it won't be reset every time the `PrintPage` event handler is entered. After the completion of the printout, however, we must reset the static variables in anticipation of a new printout. If you neglect to reset the `itm` variable, the next you time you click the Preview & Print Data button, the code will attempt to print rows past the last one on the ListView control.

Printing Plain Text

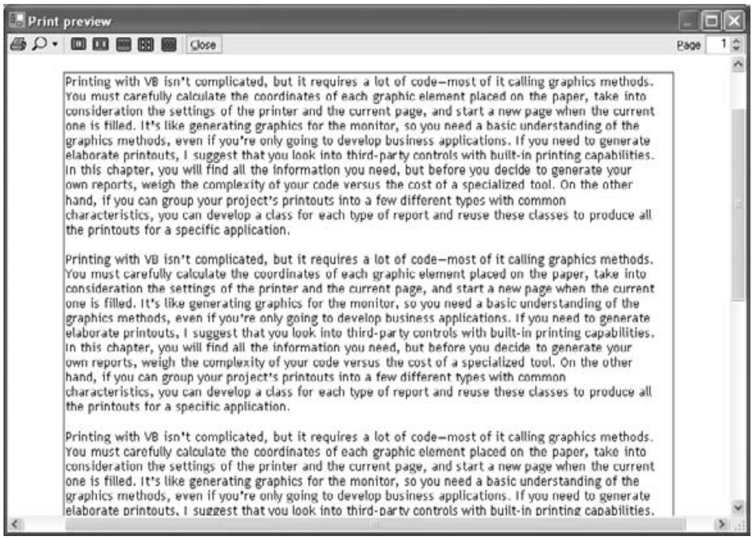
In this section, we'll examine a less-exciting operation: the printing of a text file. It should be a trivial task after the program that prints the tabular reports, but it's not nearly as trivial as you might think. But why bother with a simple operation such as printing plain text? The reason is that no control has built-in printing capabilities, and text files are still quite common. Printing formatted text is even more complicated, so we'll start with plain-text files.

Plain text means that all characters are printed in the same font, size, and style — just like the text you enter in a TextBox control. Your task is to start a new page when the current one fills and to break the lines at or before the right margin. Because the text is totally uniform, you know in advance the height of each line and you can easily calculate the number of lines per page ahead of time.

VB 2008 AT WORK: THE PRINTTEXT PROJECT

In this section, we'll build the PrintText application. The main form of this application contains a TextBox control and a button that prints the text on the control. The program displays a preview of the text in a Print Preview dialog box, and you can print the text by clicking the Print button in the dialog box. Figure 20.9 shows a section of text previewed with the PrintText application.

FIGURE 20.9
Printing and previewing documents with the PrintText application



Printing with VB isn't complicated, but it requires a lot of code—most of it calling graphics methods. You must carefully calculate the coordinates of each graphic element placed on the paper, take into consideration the settings of the printer and the current page, and start a new page when the current one is filled. It's like generating graphics for the monitor, so you need a basic understanding of the graphics methods, even if you're only going to develop business applications. If you need to generate elaborate printouts, I suggest that you look into third-party controls with built-in printing capabilities. In this chapter, you will find all the information you need, but before you decide to generate your own reports, weigh the complexity of your code versus the cost of a specialized tool. On the other hand, if you can group your project's printouts into a few different types with common characteristics, you can develop a class for each type of report and reuse these classes to produce all the printouts for a specific application.

Printing with VB isn't complicated, but it requires a lot of code—most of it calling graphics methods. You must carefully calculate the coordinates of each graphic element placed on the paper, take into consideration the settings of the printer and the current page, and start a new page when the current one is filled. It's like generating graphics for the monitor, so you need a basic understanding of the graphics methods, even if you're only going to develop business applications. If you need to generate elaborate printouts, I suggest that you look into third-party controls with built-in printing capabilities. In this chapter, you will find all the information you need, but before you decide to generate your own reports, weigh the complexity of your code versus the cost of a specialized tool. On the other hand, if you can group your project's printouts into a few different types with common characteristics, you can develop a class for each type of report and reuse these classes to produce all the printouts for a specific application.

Printing with VB isn't complicated, but it requires a lot of code—most of it calling graphics methods. You must carefully calculate the coordinates of each graphic element placed on the paper, take into consideration the settings of the printer and the current page, and start a new page when the current one is filled. It's like generating graphics for the monitor, so you need a basic understanding of the graphics methods, even if you're only going to develop business applications. If you need to generate elaborate printouts, I suggest that you look into third-party controls with built-in printing capabilities.

The idea is to instantiate a Rectangle object that represents the printable area of the page. Then call the `MeasureString` method to find out how many characters will fit into the rectangle, and print that many characters with the `DrawString` method. Just two method calls, and the first page is ready. Repeat the same process for the following pages, starting with the character following the last character printed on the previous page.

The text to be printed is stored in the *`textToPrint`* variable, which is declared at the form's level. To make the application more flexible, I added a Page Setup dialog box, in which users can specify the margins and the orientation of the printout. The application displays the Page Setup dialog box by calling the `ShowDialog` method of the PageSetupDialog control. Then it initiates printing on an instance of the PrintPreviewDialog control by calling its `ShowDialog` method. Listing 20.8 shows the code behind the Preview Printout button on the form, which initiates the printing, and the `PrintPreview()` subroutine.

LISTING 20.8: Initiating the Printing of Plain Text

```
Private Sub bttnPreview_Click(...) Handles bttnPreview.Click
    PrintPreview()
End Sub

Public Sub PrintPreview()
    PD = New Printing.PrintDocument
    PSetup.PageSettings = PD.DefaultPageSettings
    If PSetup.ShowDialog() = DialogResult.OK Then
        PPView.Document = PD
        PPView.ShowDialog()
    End If
End Sub
```

The `ShowDialog` method of the PrintPreviewDialog control is equivalent to calling the `Print` method of the PrintDocument control. After that, a series of `PrintPage` events will follow. Listing 20.9 shows the code in the `PrintPage` event's handler.

Listing 20.9: Printing Plain Text

```
Private Sub PD_PrintPage(ByVal sender As Object, _
            ByVal e As System.Drawing.Printing.PrintPageEventArgs) _
            Handles PD.PrintPage
    Static currentChar As Integer
    Static currentLine As Integer
    Dim txtFont As Font = TextBox1.Font
    Dim txtH, txtW As Integer
    Dim LMargin, TMargin As Integer
    ' Calculate the dimensions of the printable area of the page
    With PD.DefaultPageSettings
        txtH = .PaperSize.Height - _
            .Margins.Top - .Margins.Bottom
        txtW = .PaperSize.Width - _
            .Margins.Left - .Margins.Right
        LMargin = PD.DefaultPageSettings.Margins.Left
        TMargin = PD.DefaultPageSettings.Margins.Top
    End With
    e.Graphics.DrawRectangle(Pens.Blue, _
               New Rectangle(LMargin, TMargin, txtW, txtH))
    ' If the text is printed sideways, swap the printable area's
    ' width and height
    If PD.DefaultPageSettings.Landscape Then
        Dim tmp As Integer
        tmp = txtH
        txtH = txtW
        txtW = tmp
    End If
    ' Calculate the number of lines per page
    Dim linesperpage As Integer = _
              CInt(Math.Round(txtH / txtFont.Height))
    ' R is the rectangle in which the text should fit
    Dim R As New RectangleF(LMargin, TMargin, txtW, txtH)
    Dim fmt As StringFormat
    If Not TextBox1.WordWrap Then
        fmt = New StringFormat(StringFormatFlags.NoWrap)
        fmt.Trimming = StringTrimming.EllipsisWord
        Dim i As Integer
        For i = currentLine To Math.Min(currentLine + linesperpage, _
                   TextBox1.Lines.Length - 1)
            e.Graphics.DrawString( _
                        TextBox1.Lines(i), txtFont, _
                        Brushes.Black,
                        New RectangleF(LMargin, _
                        TMargin + txtFont.Height * (i - currentLine), _
                        txtW, txtFont.Height), fmt)
        Next
        currentLine += linesperpage
        If currentLine >= TextBox1.Lines.Length Then
            e.HasMorePages = False
            currentLine = 0
        Else
            e.HasMorePages = True
        End If
        Exit Sub
    End If
    fmt = New StringFormat(StringFormatFlags.LineLimit)
    Dim lines, chars As Integer
    e.Graphics.MeasureString(Mid(TextBox1.Text, currentChar + 1), _
              txtFont, _
              New SizeF(txtW, txtH), fmt, chars, lines)
    If currentChar + chars  < TextBox1.Text.Length Then
        If TextBox1.Text.Substring(currentChar + chars, 1) <> " " And _
          TextBox1.Text.Substring(currentChar + chars, 1) <> vbLf Then
            While chars > 0
               AndAlso TextBox1.Text.Substring _
                  (currentChar + chars, 1)<> "
               AndAlso _
               TextBox1.Text.Substring(currentChar + chars, 1) <> vbLf
               chars -= 1
            End While
            chars += 1
        End If
    End If
    e.Graphics.DrawString(TextBox1.Text.Substring(currentChar, chars), _
                          txtFont, Brushes.Black, R, fmt)
    currentChar = currentChar + chars
    If currentChar < TextBox1.Text.Length Then
        e.HasMorePages = True
    Else
        e.HasMorePages = False
        currentChar = 0
    End If
End Sub
```

The PrintPage event handler is quite lengthy, but if you open the PrintText project, you will find a lot of comments that will help you understand how it works. The core of the printing code is concentrated in the following three statements:

```
e.Graphics.MeasureString(textToPrint.SubString( currentChar + 1), _
               txtFont, New SizeF(txtW, txtH), fmt, chars, lines)
e.Graphics.DrawString(textToPrint.SubString(currentChar + 1), _
               txtFont, Brushes.Black, R, fmt)
currentChar = currentChar + chars
```

The first statement determines the number of characters that will fit in a rectangle with dimensions *txtW* and *txtH* when rendered on the page in the specified font. The *fmt* argument is crucial for the proper operation of the application, and I will explain it momentarily. The MeasureString method calculates the number of characters that will fit in the specified rectangle, because all characters will be rendered in the same font and carriage returns are normal characters (in a way, they're printed and move to the next line).

The second statement prints the segment of the text that will fit in this rectangle. Notice that the code is using the SubString method to pass not the entire text, but a segment starting at the location following the last character on the previous page. The location of the first character on the page is given by the *currentChar* variable, which is increased by the number of characters printed on the current page. The number of characters printed on the current page is retrieved by the MeasureString method and stored in the *chars* variable.

And the trick that makes this code work is how the *fmt* StringFormat object is declared. The height of the printable area of the page might not (and usually does not) accommodate an integer number of lines. The MeasureString method will attempt to fit as many text lines in the specified rectangle as possible, even if the last line fits only partially. To force the MeasureString and DrawString methods to work with an integer number of lines, create a FormatString object passing the constant *StringFormatFlags.LineLimit* as an argument:

```
Dim fmt As New StringFormat(StringFormatFlags.LineLimit)
```

If you pass the *fmt* object as argument to both the MeasureString and DrawString methods, they ignore partial lines, and the rest of the printing code works as expected.

If the user changes the orientation of the page, the code switches the page's width and height (see the If PD.DefaultPageSettings.Landscape block in the listing). This is all it takes to print text in landscape orientation. The page's margins are also accounted for.

The program also takes into consideration the control's WordWrap property. If word wrapping has been turned off, the program prints only the section of the line that will fit across the page. You can adjust the code to handle program listings. Program listings are plain-text files, like the ones you can print with this application, but you must mark long code lines that are broken to fit on the page. You can insert a special symbol either at the end of a code line that continues on the following line on the page, or in front of the continued line. This symbol is usually a bent arrow that resembles the Enter key. You can also number the lines while printing them.

Printing Bitmaps

If you have a color printer, you probably want to print images, too. Actually, most black-and-white printers print images in grayscale too, so you can experiment with the material of this chapter even if you have only a black-and-white laser printer. As you have probably guessed, you call the DrawImage method to send the bitmap to the printer. As a reminder, the simplest form of the DrawImage method of the Graphics object accepts two arguments, the bitmap to be drawn (an Image object) and a rectangle in which the image will be drawn:

```
Graphics.DrawImage(image, rectangle)
```

The method will stretch the bitmap specified by the *image* argument to fill the rectangle specified by the *rectangle* argument. It's imperative that you carefully calculate the dimensions of the rectangle, so that they will retain their original aspect ratio. If not, the image will be distorted in the process. Most applications will let the user specify a zoom factor, and then apply it to both dimensions. If the image fits on the page in actual size, you can make the rectangle equal to the dimensions of the image and not worry about distortions.

Because the reduced image will, most likely, be smaller than the dimensions of the paper on which it will be printed, you must also center the image on the paper. To do so, you can subtract the image's width from the paper's width and split the difference on the two sides of the image (you will do the same for the vertical margins). These operations will be demonstrated with the code of the PrintBitmap application, whose main form is shown in Figure 20.10. The application allows you to load an image and zoom in or out. The Zoom ➢ Auto command resizes the image to fit the current size of the form as best as possible, while the Zoom ➢ Normal command displays the image in actual size, regardless of whether it fits on the form or not. If not, the appropriate scroll bars will be attached automatically, because the form's AutoSize property is set to True.

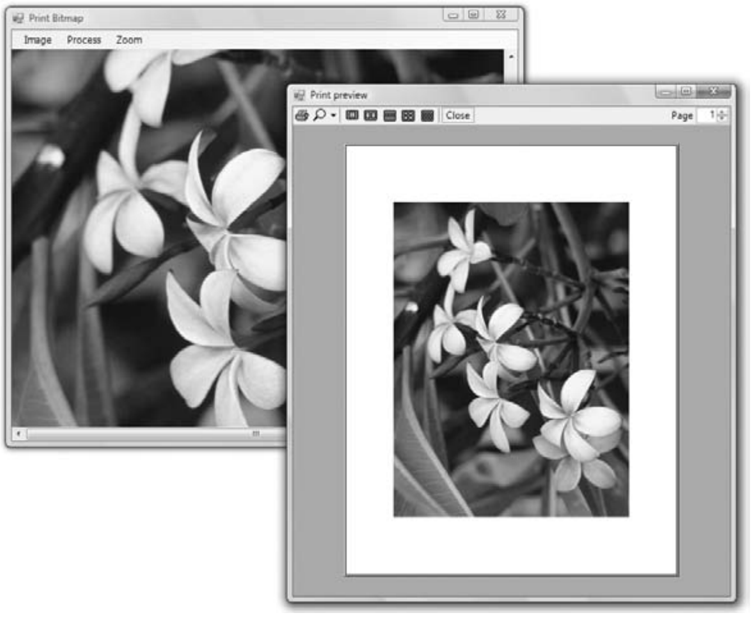


FIGURE 20.10
The PrintBitmap application resizes and rotates bitmaps to best fit the width of the page and prints them.

If you specify a rectangle the same size as the image, the image will be printed at its actual size. A common image resolution is 72 dots per inch. If the bitmap is 1,024 pixels wide, it will take approximately 14 inches across the page — this means that part of the image won't be printed.

If the bitmap is too large for a letter-size page, you must reduce its size. The following statements, which must appear in the PrintDocument event, print the image centered on the page.

If the image doesn't fit on the page, its top-left corner is printed at the origin, and the rightmost and bottommost parts of the image will be cropped. Notice also that the image isn't printed in actual size; instead, it's printed at the current magnification. Listing 20.10 provides the code of the `PrintPage` event handler.

Listing 20.10: Scaling and Printing a Bitmap

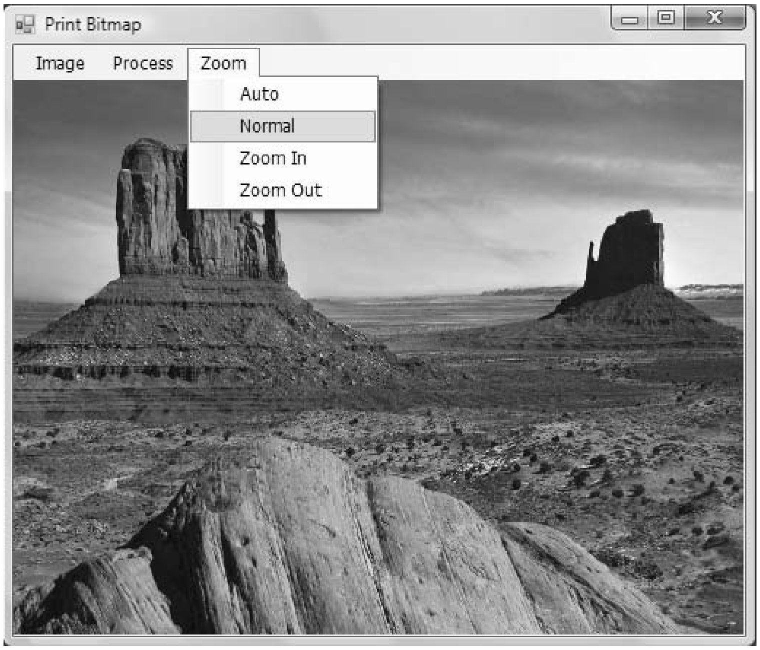
```
Private Sub PrintDocument1_PrintPage( _
        ByVal sender As Object, ByVal e As _
        System.Drawing.Printing.PrintPageEventArgs) _
        Handles PrintDocument1.PrintPage
   Dim R As Rectangle
   Dim PictWidth, PictHeight, PictLeft, PictTop As Integer
   PictWidth = PictureBox1.Width
   PictHeight = PictureBox1.Height
   With PrintDocument1.DefaultPageSettings.PaperSize
      If PictWidth < .Width Then
         PictLeft = (.Width - PWidth) / 2
      Else
         PictLeft = 0
      End If
      If PictHeight < .Height Then
         PictTop = (.Height - PHeight) / 2
      Else
         PictTop = 0
      End If
   End With
   R = New Rectangle(PictLeft, PictTop, PictWidth, PictHeight)
   e.Graphics.DrawImage(PictureBox1.Image, R)
End Sub
```

The *PictWidth* and *PictHeight* variables hold the dimensions of the scaled image, whereas *PictLeft* and *PictTop* are the coordinates of the image's top-left corner on the page. To initiate the printing process, you must call the PrintDocument object's `Print` method, or you can display the Print Preview dialog box, which is what the following code does:

```
Private Sub bttnPrint_Click(...) Handles bttnPrint.Click
   PrintPreviewDialog1.Document = PrintDocument1
   PrintPreviewDialog1.ShowDialog()
End Sub
```

The PrintBitmap application allows the user to resize and rotate the image before printing it. These rotation commands can be found in the main form's Process menu; the Zoom menu has four options: Auto, Normal, Zoom In, and Zoom Out (Figure 20.11). The last two commands zoom in and out, respectively, by 25 percent at a time. These commands change the size of the PictureBox control that holds the image, and the `PrintPage` event handler uses the dimensions of this control to determine the dimensions of the printed image. The Normal command resets the image to its actual size, and the Auto command resizes the image proportionally so that its height is 400 pixels.

FIGURE 20.11
The PrintBitmap application's main form



The Bottom Line

Use the printing controls and dialog boxes. To print with the .NET Framework, you must add an instance of the PrintDocument control to your form and call its `Print` method. To preview the same document, you simply assign the PrintDocument object to the `Document` property of the PrintPreviewDialog control and then call the `ShowDialog` method of the PrintPreviewDialog control to display the preview window. You can also display the Print dialog box, where users can select the printer to which the output will be sent, and the Page Setup dialog box, where users can specify the page's orientation and margins. The two dialog boxes are implemented with the PrintDialog and PageSetupDialog controls.

Master It Explain the process of generating a simple printout. How will you handle multiple report pages?

Master It Assuming that you have displayed the Page Setup dialog box control to the user, how will you draw a rectangle that delimits the printing area on the page, taking into consideration the user-specified margins?

Print plain text and images. Typical business applications generate printouts with text and a few borders or grids. The `DrawString` method of the Graphics object can print a string at a specific location on the page. To print images, call the `DrawImage` method of the Graphics object, passing as an argument the image you want to print and the rectangle on the page where you want the image to appear.

Master It Outline the process of printing the contents of a TextBox control.

Print tabular data. Business applications make heavy use of reports, and you should provide a mechanism to print these out. Printing tabular data isn't a simple task, but after you break the page into rows and columns, you can draw the appropriate string into its corresponding cell.

Master It Describe the process of building a tabular report.

Chapter 21

Basic Concepts of Relational Databases

In this and the following three chapters, you'll explore databases and data drive programming, starting with the basics: how databases store data, how to update a database, and how to retrieve the information you need from a database. The database-related topics discussed in this book were chosen to help you get started with database programming. I've selected topics that will help you master basic concepts of databases and ADO, rather than attempt to touch on a large number of topics.

In this chapter, you'll look at the basic concepts of relational databases, the visual data tools, and the Structured Query Language (SQL). This chapter isn't about VB, and you can skip it if you're familiar with databases and SQL. Because I can't assume that all readers are comfortable with these topics, I'm including this chapter to help readers understand the foundations of database programming. Databases are among the most complicated objects in programming, yet they're based on common-sense principles. Once you understand these principles, you'll find that database programming isn't as complicated as you may have thought.

In this chapter, you'll learn how to do the following:

- Use relational databases
- Utilize the data tools of Visual Studio
- Use the Structured Query Language for accessing tables

What Is a Database?

A *database* is a container for storing complex structured information. The same is true for a file, or even for the file system on your hard disk. What makes a database unique is that databases are designed to make data easily retrievable. The purpose of a database is not so much the storage of information as its quick retrieval. In other words, you must structure your database so that it can be queried quickly and efficiently.

Databases are maintained by special programs, such as Microsoft Office Access and SQL Server. These programs are called *database management systems (DBMSs)*, and they're among the most complicated applications. A fundamental characteristic of a DBMS is that it isolates much of the complexity of the database from the developer. Regardless of how each DBMS stores data on disk, you see your data organized in tables with relationships between tables. To access or update the data stored in the database, you use a special language, Structured Query Language (SQL). Unlike other areas of programming, SQL is a truly universal language, and all major DBMSs support this language.

The recommended DBMS for Visual Studio 2008 is SQL Server 2008. In fact, the Visual Studio 2008 setup program offers to install a developer version of SQL Server 2008 called SQL Server 2008 Express. However, you can use Access or even non-Microsoft databases such as Oracle. Although this chapter was written with SQL Server 2008, most of the examples will work with Access as well.

Data is stored in *tables*, and each table contains entities of the same type. In a database that stores information about books, there will be a table with titles, another table with authors, and a table with publishers. The table with the titles contains information such as the title of the book, the number of pages, and the book's description. Author names are stored in a different table because each author might appear in multiple titles. If author information were stored along with each title, we'd be repeating author names. So every time we wanted to change an author's name, we'd have to modify multiple entries in the titles table. Even retrieving a list of unique author names would be a challenge because you'd have to scan the entire table with the titles, retrieve all the authors, and then get rid of the duplicate entries. The same is true for publishers. Publishers are stored in a separate table, and each title contains a pointer to the appropriate row in the publishers table.

The reason for breaking the information we want to store in a database into separate tables is to avoid duplication of information. This is a key point in database design. Duplication of information will sooner or later lead to inconsistencies in the database. The process of breaking the data into related tables that eliminate all possible forms of information duplication is called *normalization*, and there are rules for normalizing databases. The topic of database normalization is not discussed further in this book. However, all it really takes to design a functional database is common sense. After you learn how to extract data from your database's tables with SQL statements, you'll develop a better understanding of the way databases should be structured.

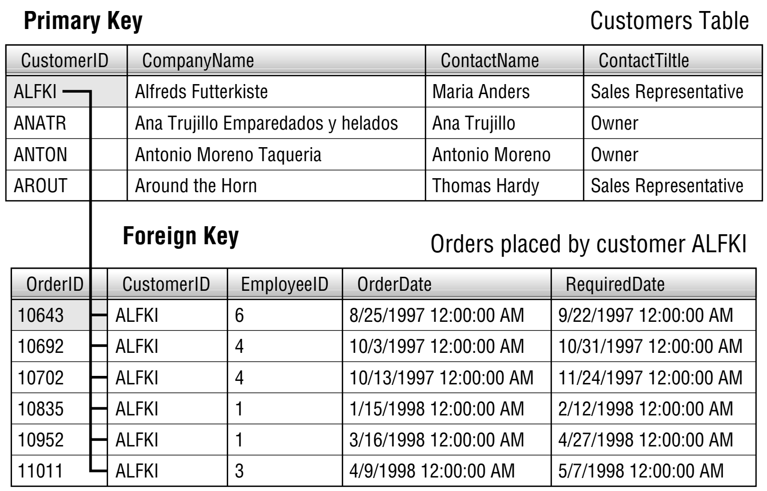
Using Relational Databases

The databases we're interested in are called *relational* because they are based on relationships among the data they contain. The data is stored in tables, and tables contain related data, or *entities,* such as persons, products, orders, and so on. The idea is to keep the tables small and manageable; thus, separate entities are kept in their own tables. If you start mixing customers and invoices, products and their suppliers, or books, publishers, and authors in the same table, you'll end up repeating information — a highly undesirable situation. If there's one rule to live by as a database designer and programmer, this is it: *Do not duplicate information*.

Of course, entities are not independent of each other. For example, orders are placed by specific customers, so the rows of the Customers table must be linked to the rows of the Orders table that stores the orders of the customers. Figure 21.1 shows a segment of a table with customers (top) and the rows of a table with orders that correspond to one of the customers (bottom).

As you can see in Figure 21.1, relationships are implemented by inserting columns with matching values in the two related tables; the *`CustomerID`* column is repeated in both tables. The rows with a common value in the *`CustomerID`* fields are related. In other words, the lines that connect the two tables simply indicate that there are two fields, one on each side of the relationship, with a common value. The customer with the ID value ALFKI has placed the orders 10643 and 10692 (among others). To find all the orders placed by a customer, we can scan the Orders table and retrieve the rows in which the `CustomerID` field has the same value as the ID of the specific customer in the Customers table. Likewise, you can locate customer information for each order by looking up the row of the Customers table that has the same ID as the one in the `CustomerID` field of the Orders table.

The two fields used in a relationship are called *key fields.* The `CustomerID` field of the Customers table is the *primary key* because it identifies a single customer. Each customer has a unique value in the `CustomerID` field. The `CustomerID` field of the Orders table is the *foreign key* of the relationship. A `CustomerID` value appears in a single row of the Customers table and identifies that row; it's the table's primary key. However, it might appear in multiple rows of the Orders table because the `CustomerID` field is the foreign key in this table. In fact, it will appear in as many rows of the Orders table as there are orders for the specific customer. Note that the primary and foreign keys need not have the same names, but it's convenient to use the same name because they both represent the same entity.



CustomerID	CompanyName	ContactName	ContactTiltle
ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative
ANATR	Ana Trujillo Emparedados y helados	Ana Trujillo	Owner
ANTON	Antonio Moreno Taqueria	Antonio Moreno	Owner
AROUT	Around the Horn	Thomas Hardy	Sales Representative

OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate
10643	ALFKI	6	8/25/1997 12:00:00 AM	9/22/1997 12:00:00 AM
10692	ALFKI	4	10/3/1997 12:00:00 AM	10/31/1997 12:00:00 AM
10702	ALFKI	4	10/13/1997 12:00:00 AM	11/24/1997 12:00:00 AM
10835	ALFKI	1	1/15/1998 12:00:00 AM	2/12/1998 12:00:00 AM
10952	ALFKI	1	3/16/1998 12:00:00 AM	4/27/1998 12:00:00 AM
11011	ALFKI	3	4/9/1998 12:00:00 AM	5/7/1998 12:00:00 AM

FIGURE 21.1
Linking customers and orders with relationships

The operation of matching rows in two tables based on their primary and foreign keys is called a *join*. Joins are basic operations in manipulating tables and are discussed in detail in the section ''Structured Query Language'' later in this chapter.

To help you understand relational databases, I will present the structure of the two sample databases used for the examples in this and the following chapters. If you're not familiar with the Northwind and Pubs databases, read the following two sections and you'll find it easier to follow the examples.

Obtaining the Northwind and Pubs Sample Databases

SQL Server 2008 developers will wonder where the Northwind and Pubs databases have gone. Microsoft has replaced both databases with a single new database called AdventureWorks. Microsoft made the change to demonstrate new SQL Server features in an environment that more closely matches large enterprise systems. Because the AdventureWorks database is extremely complex and not very friendly for teaching database principles, this book won't rely on it. However, you might want to look at the AdventureWorks database anyway to see what it provides and understand how complex databases can become.

Many developers are used to working with the Northwind and Pubs databases with other Microsoft products. These two databases have become so standard that many authors, including myself, rely on the presence of these databases to ensure that everyone can see example code without a lot of extra effort. Unfortunately, you won't find an option for installing them as part of the standard SQL Server 2008 installation. However, you can find scripts for creating these databases in SQL Server Express online at `www.microsoft.com/downloads/details.aspx?familyid=06616212-0356-46a0-8da2-eebc53a68034`. The name of the file you'll receive is `SQL2000SampleDb.MSI`. Even though Microsoft originally created this file for SQL Server 2000, it works just fine with SQL Server 2008.

After you download the script files, you need to install them. Right-click the file and choose Install from the context menu. You will see a Welcome dialog box, telling you that this file contains the sample databases for SQL Server 2000. Click Next, read the licensing agreement, and agree to it. Keep following the prompts until you install the sample database scripts in the appropriate directory.

At this point, you have two scripts for creating the sample databases. If you used the default installation settings, these files appear in the `\Program Files\Microsoft SQL Server 2000 Sample Database Scripts` folder of your machine. The `InstNwnd.SQL` file will create the Northwind database, and the `InstPubs.SQL` file will create the Pubs database.

Double-click the name of each SQL file, and each will open in SQL Server's Management Studio. Then click the Execute button in the toolbar (it's the button with the icon of an exclamation mark) to run the script, which will install the appropriate database.

To install the databases for the Express version of SQL Server 2008, open a command prompt. Type **`OSQL -E -i InstNwnd.SQL`** and press Enter. The OSQL utility will create the Northwind database for you (this process can take quite some time). After the Northwind database is complete, type **`OSQL -E -i InstPubs.SQL`** and press Enter. The process will repeat itself.

If you try to run the OSQL utility and receive an error message at the command prompt, the SQL Server 2008 installation didn't modify the path information for your system as it should have. In some cases, this makes your installation suspect, and you should reinstall the product if you experience other problems. To use the installation scripts, copy them from the installation folder to the `\Program Files\Microsoft SQL Server\90\Tools\binn` folder. You can run the OSQL utility at the command prompt from this folder to create the two sample databases.

You'll want to test the installation to make sure it worked. Open Visual Studio and choose View ➢ Server Explorer to display the Server Explorer. Right-click Data Connections and choose Add Connection from the context menu. Server Explorer will display the Add Connection dialog box shown in Figure 21.2 (this one already has all the information filled out).

In the Server Name field, type the name of your machine or select one with the mouse. Click the down arrow in the Select Or Enter A Database Name field. You should see both the Northwind and Pubs databases, as shown in Figure 21.2. If you don't see these entries, it means that an error occurred. Try running the scripts a second time.

You need to make sure that you can access the databases. Choose the Northwind database. Click Test Connection. When the scripts install the databases properly and you can access them, you'll see a message indicating that you have successfully connected to the selected database.

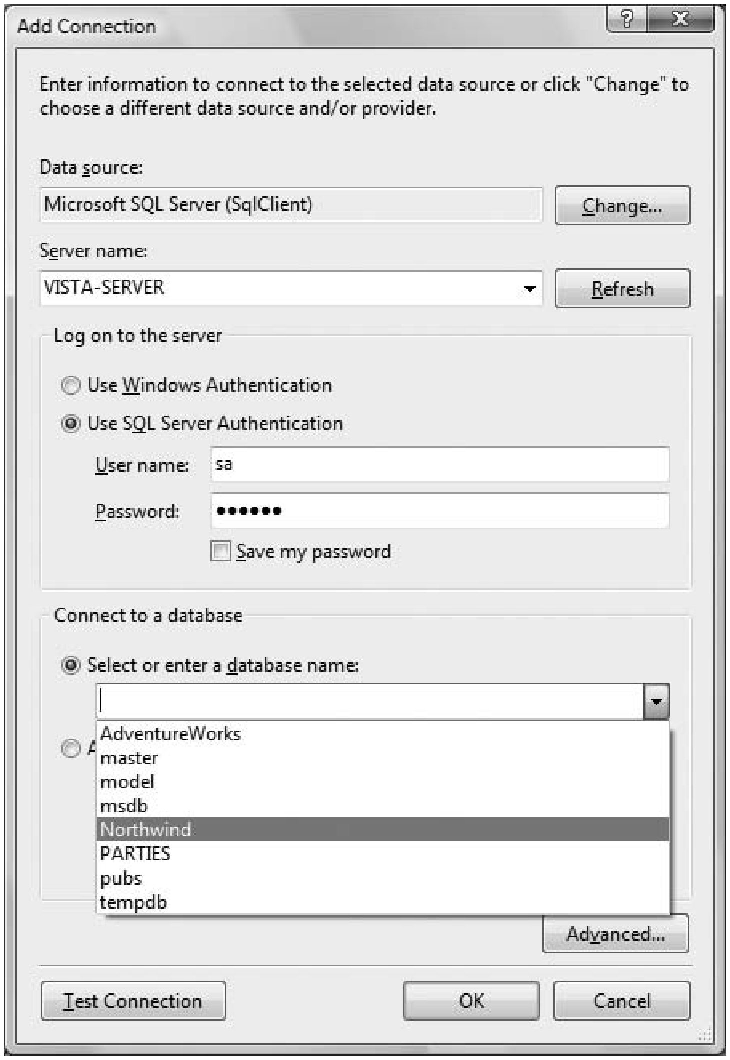
Exploring the Northwind Database

In this section, you'll explore the structure of the Northwind sample database. The Northwind database stores products, customers, and sales data, and many of you are already familiar with the structure of the database.

To view a table's contents, expand the Table section of the tree under the Northwind connection in Server Explorer and locate the name of the table you want to examine. Right-click the name and choose Show Table Data from the context menu. This will open the table, and you can view and edit its rows. If you choose the Open Table Definition command from the same menu, you will see the definitions of the table's columns. You can change the type of the columns (each column stores items of the same type), their length, and set a few more properties that are discussed a little later in this chapter. To follow the description of the sample databases, open the tables in view mode.

If you have installed Visual Studio 2008, you can use the SQL Server Managements Studio to explore the same database. Just right-click the Northwind database and from the context menu select Open Table to view the data, or Design to change the table's definition.

FIGURE 21.2
Use the Add Connection dialog box to check for the two databases.


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)
Change...
Server name:
VISTA-SERVER
Refresh
Log on to the server
Use Windows Authentication
Use SQL Server Authentication
User name:
sa
Password:
••••••
Save my password
Connect to a database
Select or enter a database name:
AdventureWorks
master
model
msdb
Northwind
PARTIES
pubs
tempdb
Advanced...
Test Connection
OK
Cancel

PRODUCTS TABLE

The Products table stores information about the products of the Northwind Corporation. This information includes the product's name, packaging information, price, and other relevant fields. Each product (or row) in the table is identified by a unique numeric ID. Because each ID is unique, the `ProductID` column is the table's primary key. The rows of the Products table are referenced by invoices (the Order Details table, which is discussed later), so the product IDs appear in the Order Details table as well.

SUPPLIERS TABLE

Each product has a supplier, too. Because the same supplier can offer more than one product, the supplier information is stored in a different table, and a common field, the `SupplierID` field, is used to link each product to its supplier (as shown in Figure 21.3). For example, the products Chai, Chang, and Aniseed Syrup are purchased from the same supplier: Exotic Liquids. Their `SupplierID` fields all point to the same row in the Suppliers table.

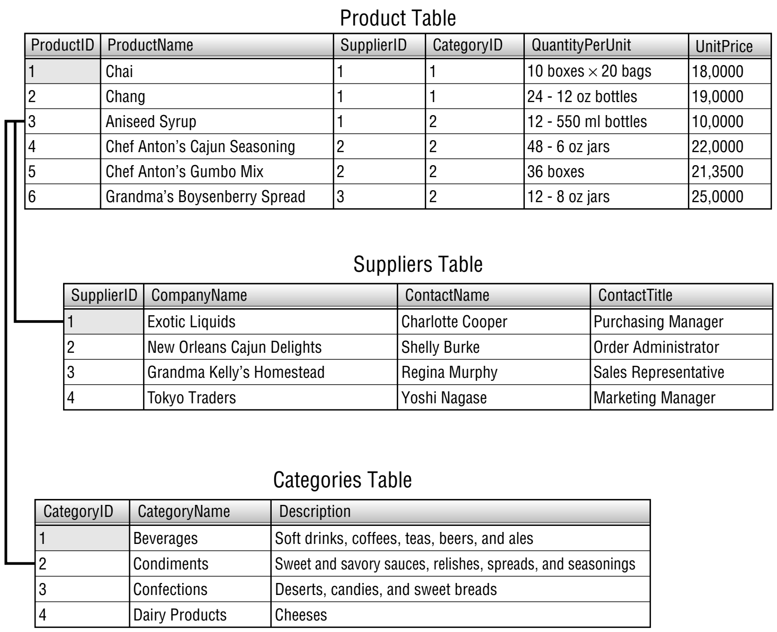


FIGURE 21.3 Linking products to their suppliers and their categories

Product Table

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice
1	Chai	1	1	10 boxes × 20 bags	18,0000
2	Chang	1	1	24 - 12 oz bottles	19,0000
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10,0000
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22,0000
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21,3500
6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars	25,0000

Suppliers Table

SupplierID	CompanyName	ContactName	ContactTitle
1	Exotic Liquids	Charlotte Cooper	Purchasing Manager
2	New Orleans Cajun Delights	Shelly Burke	Order Administrator
3	Grandma Kelly's Homestead	Regina Murphy	Sales Representative
4	Tokyo Traders	Yoshi Nagase	Marketing Manager

Categories Table

CategoryID	CategoryName	Description
1	Beverages	Soft drinks, coffees, teas, beers, and ales
2	Condiments	Sweet and savory sauces, relishes, spreads, and seasonings
3	Confections	Deserts, candies, and sweet breads
4	Dairy Products	Cheeses

CATEGORIES TABLE

In addition to having a supplier, each product belongs to a category. Categories are not stored along with product names; they are stored separately in the Categories table. Again, each category is identified by a numeric value (field *`CategoryID`*) and has a name (field *`CategoryName`*). In addition, the Categories table has two more columns: *`Description`*, which contains text, and *`Picture`*, which stores a bitmap. The *`CategoryID`* field in the Categories table is the primary key, and the field by the same name in the Products table is the corresponding foreign key.

CUSTOMERS TABLE

The Customers table stores information about the company's customers. Each customer is stored in a separate row of this table, and customers are referenced by the Orders table. Unlike product IDs, customer IDs are five-character strings and are stored in the *`CustomerID`* column. This is an unusual choice for IDs, which are usually numeric values.

ORDERS TABLE

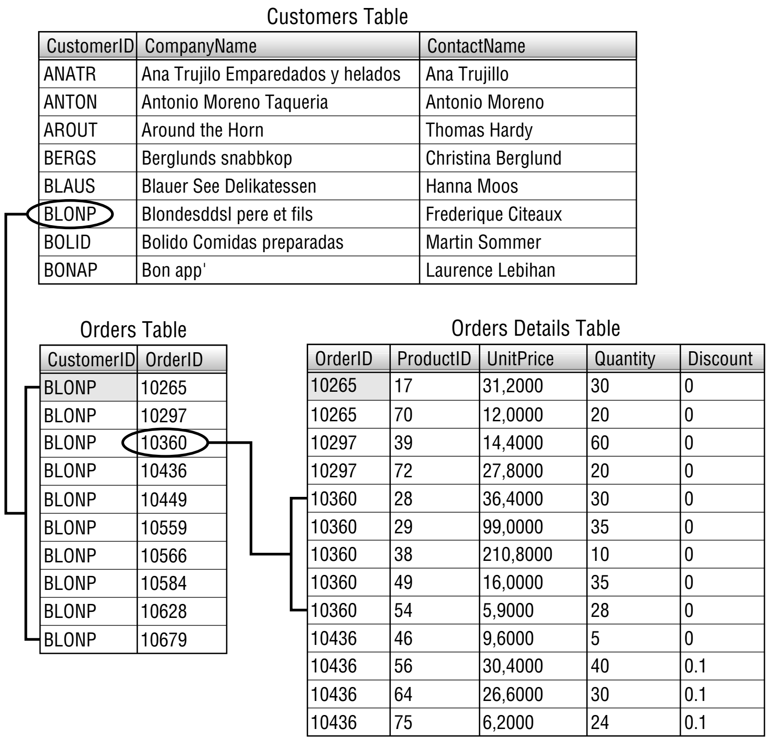
The Orders table stores information about the orders placed by Northwind's customers. The *`OrderID`* field, which is an integer value, identifies each order. Orders are numbered sequentially, so this field is also the order's number. Each time you append a new row to the Orders table, the value of the new *`OrderID`* field is generated automatically by the database. The *`OrderID`* column is not only the table's primary key, it's also an `AutoIncrement` column: Every time a new column is added to the table, the *`OrderID`* field is assigned the next available integer value automatically by SQL Server. Usually, the primary key in any table is an `AutoIncrement` column and SQL Server ensures that this column has unique values.

The Orders table is linked to the Customers table through the *`CustomerID`* column. By matching rows that have identical values in their *`CustomerID`* fields in the two tables, we can recombine customers with their orders. Figure 21.1 showed how customers are linked to their orders.

Order Details Table

The Orders table doesn't store any details about the items ordered; this information is stored in the Order Details table (see Figure 21.4). Each order is made up of one or more items; and each item has a price, a quantity, and a discount. In addition to these fields, the Order Details table contains an *`OrderID`* column, which holds the ID of the order to which the detail line belongs.

Figure 21.4
Customers, Orders, and Order Details tables and their relations



Customers Table

CustomerID	CompanyName	ContactName
ANATR	Ana Trujilo Emparedados y helados	Ana Trujillo
ANTON	Antonio Moreno Taqueria	Antonio Moreno
AROUT	Around the Horn	Thomas Hardy
BERGS	Berglunds snabbkop	Christina Berglund
BLAUS	Blauer See Delikatessen	Hanna Moos
BLONP	Blondesddsl pere et fils	Frederique Citeaux
BOLID	Bolido Comidas preparadas	Martin Sommer
BONAP	Bon app'	Laurence Lebihan

Orders Table

CustomerID	OrderID
BLONP	10265
BLONP	10297
BLONP	10360
BLONP	10436
BLONP	10449
BLONP	10559
BLONP	10566
BLONP	10584
BLONP	10628
BLONP	10679

Orders Details Table

OrderID	ProductID	UnitPrice	Quantity	Discount
10265	17	31,2000	30	0
10265	70	12,0000	20	0
10297	39	14,4000	60	0
10297	72	27,8000	20	0
10360	28	36,4000	30	0
10360	29	99,0000	35	0
10360	38	210,8000	10	0
10360	49	16,0000	35	0
10360	54	5,9000	28	0
10436	46	9,6000	5	0
10436	56	30,4000	40	0.1
10436	64	26,6000	30	0.1
10436	75	6,2000	24	0.1

The reason why details aren't stored along with the order's header is that the Orders and Order Details tables store different entities. The order's header, which contains information about the customer who placed the order, the date of the order, and so on, is quite different from the information you must store for each item ordered. If you attempt to store the entire order into a single table, you'll end up repeating a lot of information. Notice also that the Order Details table stores the IDs of the products, not the product names.

Employees Table

This table holds employee information. Each employee is identified by a numeric ID, which appears in each order. When a sale is made, the ID of the employee who made the sale is recorded in the Orders table. An interesting technique was used in the design of the Employees table: Each employee has a manager, which is another employee. The employee's manager is identified by the `ReportsTo` field, which is set to the ID of the employee's manager. The rows of the Employees table contain references to the same table. This table contains a foreign key that points to the primary key of the same table, a relation that allows you to identify the hierarchy of employees in the corporation.

SHIPPERS TABLE

Each order is shipped with one of the three shippers stored in the Shippers table. The appropriate shipper's ID is stored in the Orders table.

Exploring the Pubs Database

Before looking at SQL and more practical techniques for manipulating tables, let's look at the structure of another sample database I'm going to use in this chapter, the Pubs database. Pubs is a database for storing book, author, and publisher information, not unlike a database you might build for an online bookstore.

The Pubs database is made up of really small tables, but it was carefully designed to demonstrate many of the features of SQL, so it's a prime candidate for sample code. Just about any book about SQL Server uses the Pubs database. In the examples of the following sections, I will use the Northwind database because it's closer to a typical business database, and the type of information stored in the Northwind database is closer to the needs of the average VB programmer than the Pubs database. Some of the fine points of SQL, however, can't be demonstrated with the data of the Northwind database, and so in this section I'll show examples that use the ubiquitous Pubs database.

TITLES TABLE

The Titles table contains information about individual books (the book's title, ID, price, and so on). Each title is identified by an ID, which is not a numeric value, that's stored in the `title_id` column. The IDs of the books look like this: BU2075.

AUTHORS TABLE

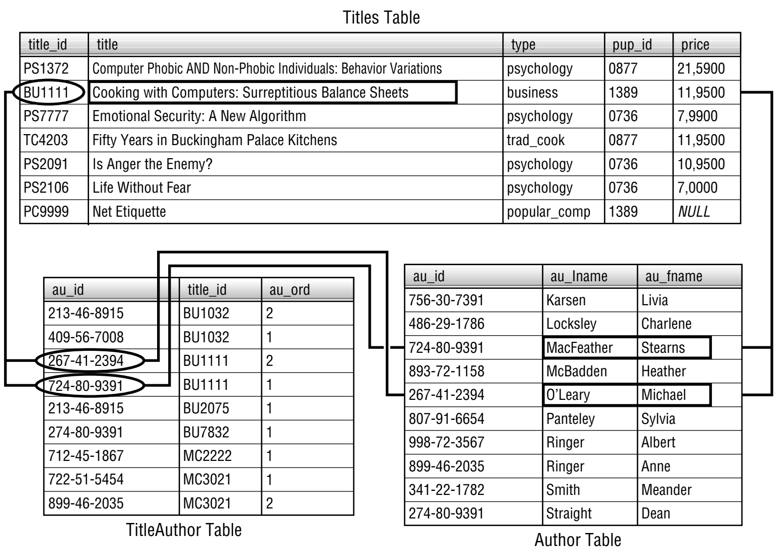
The Authors table contains information about authors. Each author is identified by an ID, which is stored in the `au_id` field. This field is a string with a value such as 172-32-1176 (they resemble U.S. Social Security numbers).

TITLEAUTHOR TABLE

The Titles and Authors tables are not directly related because they can't be joined via a one-to-many relationship; the relationship between the two tables is many-to-many. The relations you have seen so far are *one-to-many* because they relate one row in the table that has the primary key to one or more rows in the table that has the foreign key: One order contains many detail lines, one customer has many orders, one category contains many products, and so on.

The relation between titles and authors is *many-to-many*, because each book may have multiple authors, and each author may have written multiple titles. If you stop and think about the relationship between these two tables, you'll realize that it can't be implemented with a primary and a foreign key (like the Order-Customer relationship or the Order-Shipper relationship in the Northwind database). To establish a many-to-many relationship, you must create a table between the other two, and this table must have a one-to-many relationship with both tables.

Figure 21.5 shows how the Titles and Authors tables of the Pubs database are related to one another. The table between them holds pairs of title IDs and author IDs. If a book was written by two authors, the TitleAuthor table contains two entries with the same title ID and different author IDs. The book with a `title_id` of BU1111 was written by two authors. The IDs of the authors appear in the TitleAuthor table along with the ID of the book. The IDs of these two authors are 267-41-2394 and 724-80-9391. Likewise, if an author has written more than one book, the author's ID will appear many times in the TitleAuthor table — each time paired with a different title ID.



Titles Table

title_id	title	type	pup_id	price
PS1372	Computer Phobic AND Non-Phobic Individuals: Behavior Variations	psychology	0877	21,5900
BU1111	Cooking with Computers: Surreptitious Balance Sheets	business	1389	11,9500
PS7777	Emotional Security: A New Algorithm	psychology	0736	7,9900
TC4203	Fifty Years in Buckingham Palace Kitchens	trad_cook	0877	11,9500
PS2091	Is Anger the Enemy?	psychology	0736	10,9500
PS2106	Life Without Fear	psychology	0736	7,0000
PC9999	Net Etiquette	popular_comp	1389	*NULL*

au_id	title_id	au_ord
213-46-8915	BU1032	2
409-56-7008	BU1032	1
267-41-2394	BU1111	2
724-80-9391	BU1111	1
213-46-8915	BU2075	1
274-80-9391	BU7832	1
712-45-1867	MC2222	1
722-51-5454	MC3021	1
899-46-2035	MC3021	2

TitleAuthor Table

au_id	au_lname	au_fname
756-30-7391	Karsen	Livia
486-29-1786	Locksley	Charlene
724-80-9391	MacFeather	Stearns
893-72-1158	McBadden	Heather
267-41-2394	O'Leary	Michael
807-91-6654	Panteley	Sylvia
998-72-3567	Ringer	Albert
899-46-2035	Ringer	Anne
341-22-1782	Smith	Meander
274-80-9391	Straight	Dean

Author Table

FIGURE 21.5
The TitleAuthor table links titles to authors.

At times you won't be able to establish the desired relationship directly between two tables because the relationship is many-to-many. When you discover a conflict between two tables, you must create a third one between them. A many-to-many relation is actually implemented as two one-to-many relations.

PUBLISHERS TABLE

The Publishers table contains information about publishers. Each title has a `pub_id` field, which points to the matching row of the Publishers table. Unlike the other major tables of the Pubs database, the Publishers table uses a numeric value to identify each publisher.

Understanding Relations

In a database, each table has a field with a unique value for every row. As indicated earlier in this chapter, this field is the table's *primary key*. The primary key does not have to be a meaningful entity because in most cases there's no single field that's unique for each row. Books can be identified by their ISBNs, and employees by their SSNs, but these are exceptions to the rule. In general, you can't come up with a meaningful key that's universally unique. The primary key need not resemble the entity it identifies. The only requirement is that primary keys be unique in the entire table. In most designs, we use an integer as the primary key. To make sure they're unique, we even let the DBMS generate a new integer for each row added to the table. Each table can have one primary key only, and the DBMS can automatically generate an integer value for a primary key field every time a new row is added. SQL Server uses the term *Identity* for this data type, and there can be only one Identity field in each table.

The related rows in a table repeat the primary key of the row they are related to in another table. The copies of the primary keys in all other tables are called *foreign keys.* Foreign keys need not be unique (in fact, they aren't), and any field can serve as a foreign key. What makes a field a foreign key is that it matches the primary key of another table. The `CategoryID` field is the primary key of the Categories table because it identifies each category. The `CategoryID` field in the Products table is the foreign key because the same value might appear in many rows (many products can belong to the same category). Whereas the primary key refers to a table, the foreign key refers to a relationship. The `CategoryID` column of the Products table is the foreign key in the relationship between the Categories and Products tables. The Products table contains another foreign key, the `SupplierID` column, which forms the relationship between the Suppliers and Products tables.

Referential Integrity

Maintaining the links between tables is not a trivial task. When you add an invoice line, for instance, you must make sure that the product ID that you insert in the Order Details table corresponds to a row in the Products table. An important aspect of a database is its *integrity.* To be specific, you must ensure that the relations are always valid, and this type of integrity is called *referential integrity.* There are other types of integrity (for example, setting a product's value to a negative value will compromise the integrity of the database), but this is not nearly as important as referential integrity. The wrong price can be easily fixed. But issuing an invoice to a customer who doesn't exist isn't easy (if even possible) to fix. Modern databases come with many tools to help ensure their integrity, especially referential integrity. These tools are constraints you enter when you design the database, and the DBMS makes sure that the constraints are not violated as the various programs manipulate the database.

When you relate the Products and Categories tables, for example, you must also make sure of the following:

- Every product added to the foreign table must point to a valid entry in the primary table. If you are not sure which category the product belongs to, you can leave the `CategoryID` field of the Products table empty (the field will have a Null value). Or, you can create a generic category, the UNKNOWN or UNDECIDED category, and use this category if no information is available.

- No rows in the Categories table should be removed if there are rows in the Products table pointing to the specific category. This situation would make the corresponding rows of the Products table point to an invalid category (the rows that have no matching row in the primary table are called *orphan rows*).

These two restrictions would be quite a burden on the programmer if the DBMS didn't protect the database against actions that could impair its integrity. The referential integrity of your database depends on the validity of the relations. Fortunately, all DBMSs can enforce rules to maintain their integrity, and you'll learn how to enforce rules that guarantee the integrity of your database later in this chapter. In fact, when you create the relationship, you can select a couple of check boxes that tell SQL Server to enforce the relationship (that is, not to accept any changes in the data that violate the relationship). If you leave these check boxes deselected, be ready to face a real disaster sooner or later.

Visual Database Tools

To simplify the development of database applications, Visual Studio 2008 comes with some visual tools, the most important of which are briefly described here and then discussed in the following sections:

Server Explorer This is the first and most prominent tool. Server Explorer is the toolbox for database applications, in the sense that it contains all the basic tools for connecting to databases and manipulating their objects.

Query Builder This is a tool for creating SQL queries (statements that retrieve the data we want from a database or update the data in the database). SQL is a language in its own right, and we'll discuss it later in this chapter. Query Builder lets you specify the operations you want to perform on the tables of a database with point-and-click operations. In the background, Query Builder builds the appropriate SQL statement and executes it against the database.

Database Designer and Tables Designer These tools allow you to work with an entire database or its tables. When you work with the database, you can add new tables, establish relationships between the tables, and so on. When you work with individual tables, you can manipulate the structure of the tables, edit their data, and add constraints. You can use these tools to manipulate a complicated object — the database — with point-and-click operations.

Server Explorer

Your starting point for developing database applications with VB 2008 is the Server Explorer. This toolbox is your gateway to the databases on your system or network, and you can use it to locate and retrieve the tables you're interested in. Place the pointer over the Server Explorer tab to expand the corresponding toolbox, which looks something like the one shown in Figure 21.6. The two main objects in the Server Explorer are Data Connections and the Servers object. Under the Data Connections branch, you see the connections to databases you're programming against. Under the Servers branch, you see the servers you can access from your computer and various objects they expose.

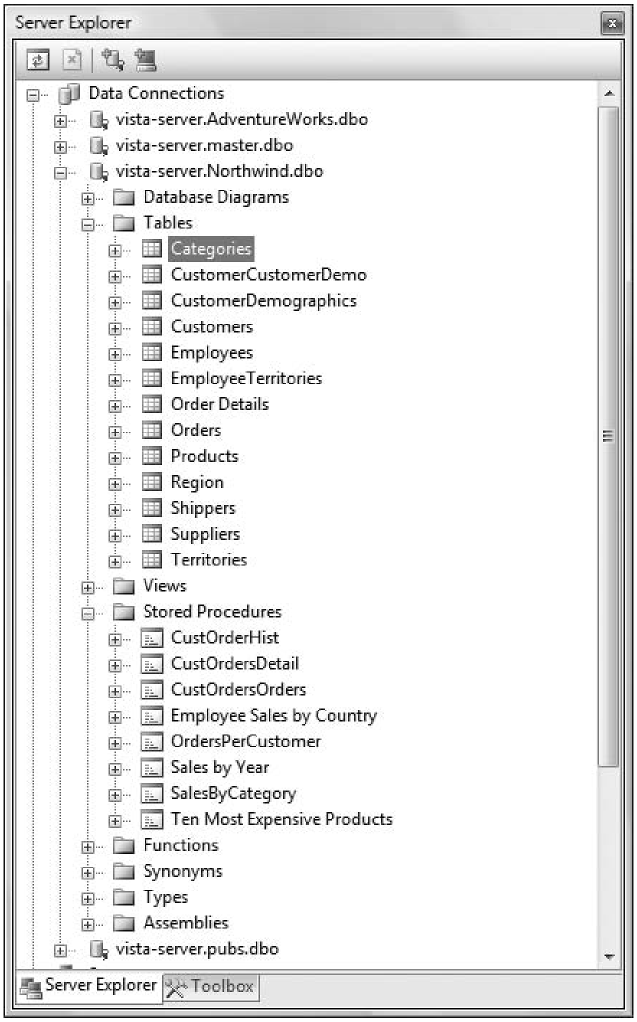
Under the Data Connections item, you may see one or more connections to existing databases (if you have already experimented with the visual data tools of Visual Studio). If you don't see a connection to the Northwind database, you must create one. Right-click the Data Connections icon and choose Add Connection from the context menu. Every new connection you add remains under the Data Connections branch until you decide to remove it, and you can use it in any number of projects.

To add a new connection, choose the Add Connection command. The first time you create a connection in Visual Studio, you'll see the Change Data Source dialog box. For the examples in this part of the book, I will use SQL Server databases. Select the Microsoft SQL Server entry and click OK.

After you click OK in the Change Data Source dialog box, you see the Add Connection dialog box, shown earlier in Figure 21.2. Here you must enter the User Name and Password in the appropriate text boxes. If you (or the administrator) have set up SQL Server to use Windows integrated security, just select the Use Windows Authentication radio button. Then drop down the top list box at the bottom of the dialog box to select one of the SQL Server databases your computer can access. You will see the local SQL Server database, as well as any other SQL Server database on the network. (If you don't see the SQL Server database you need to use, make sure that the SQL Server

Browser and SQL Server Agent services are both started. When you know the server is running, you can also type its name into the field.) Select the local SQL Server, or type the string **`localhost`** in the Server Name box; then, select the Northwind database in the second drop-down list in the dialog box.

FIGURE 21.6
The Server Explorer contains the database objects you can access on your computer.



```
Data Connections
  vista-server.AdventureWorks.dbo
  vista-server.master.dbo
  vista-server.Northwind.dbo
    Database Diagrams
    Tables
      Categories
      CustomerCustomerDemo
      CustomerDemographics
      Customers
      Employees
      EmployeeTerritories
      Order Details
      Orders
      Products
      Region
      Shippers
      Suppliers
      Territories
    Views
    Stored Procedures
      CustOrderHist
      CustOrdersDetail
      CustOrdersOrders
      Employee Sales by Country
      OrdersPerCustomer
      Sales by Year
      SalesByCategory
      Ten Most Expensive Products
    Functions
    Synonyms
    Types
    Assemblies
  vista-server.pubs.dbo
```

If you have Access 2007 installed on your system, click the Change button to see the Change Data Source dialog box, where you can select the Microsoft Access Database File entry, and then locate the MDF file of the database on your disk. On the Connection tab, click the Test Connection button to make sure that you can connect to the database. If not, make sure that SQL Server is running and that the username and password you specified are correct.

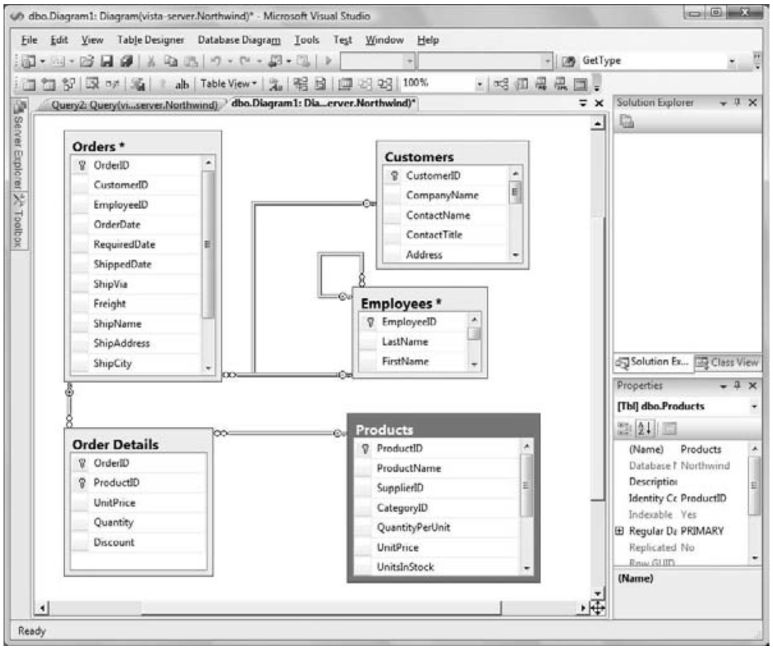
Click OK to close the Add Connection dialog box, and the name of the new connection appears under the Data Connections branch of the tree in the Server Explorer window. The default name of the connection is made up of the name of the computer followed by the name of the database — for my server, `Powerkit.Northwind` — but you can change it. Right-click a connection and choose Rename from the context menu.

Switch back to the Server Explorer tab and expand the new connection. You will see the following entries under it:

Database Diagrams This is where you can examine the various diagrams of the database. A *database diagram* is a visual representation of a set of related tables and the relations between the tables. Relations are indicated by line segments between two related tables, and you can quickly learn a lot about the structure of a database by looking at a database diagram.

Right-click the Database Diagrams item in Server Explorer and choose New Diagram from the context menu. You will see a dialog box with the names of the tables, and you must select the ones you want to add to the diagram. Add the following tables by clicking each table's name and then the Add button: Orders, Order Details, Customers, and Employees. You will see a diagram like the one shown in Figure 21.7 (you will have to rearrange the tables on the designer to view them all at once). Each table is represented by a box with the names of its columns. You can specify the items that will be displayed for each column by choosing the Table View options from the context menu of each table. All the relations in the selected tables are *one-to-many:* On one side of the relation there is a key icon, which signifies the primary key of the relation, and on the other side of the same relation there's an infinity icon, which signifies the foreign key. The same customer, for example, can place multiple items. Conversely, many orders belong to the same customer.

FIGURE 21.7
Viewing the database diagram of a section of the database



You can extend the diagram by adding more tables to it: The rows of the Order Details table reference products with the *`ProductID`* column. To view this relation, add the Products table to the diagram by right-clicking in the designer and choosing Add Table (you'll see the same dialog box, where you can select one or more tables to add to the diagram). The relation between the Order Details and Products tables will be added automatically for you in the diagram.

Tables This is where you can select a table and edit it, or add a new table to the database. You can edit the table itself (change its design by adding/removing columns or change the data types of one or more columns) or edit the table's rows. To edit the table's structure, choose Open Table Definition from the table's context menu. To view or edit the table's data, choose Show Table Data from the same menu.

Views This is where you specify the various views you want to use in your applications. Sometimes the tables are not the most convenient (or even the most expedient) method of looking at your data. If the database contains a table of employees, and this table includes wages or other sensitive data, you can create a view that's identical to the table but excludes selected columns.

Views are created with SQL `SELECT` statements, which are discussed in detail later in this chapter. A `SELECT` statement allows you to specify the information you want to retrieve from the database. This information can be stored in a View object, which is just like another table to your application. You can use views in your code as you would use tables — you just can't design views like tables: Views are based on selection queries and are discussed later in this chapter.

Stored Procedures *Stored procedures* are (usually small) programs that are stored in the database and perform specific and often repetitive tasks. By coding many of the operations you want to perform against the database as stored procedures, you won't have to access the database directly. Moreover, you can call the same stored procedure from several places in your VB code, and you can be sure that the same action is performed every time. Once created, the stored procedure becomes part of the database, and programmers (as well as users) can call it by name, passing the appropriate arguments if necessary. A typical example is a stored procedure for removing orders. The stored procedure must remove the order details first and then remove the order (and possibly update the customer's balance, the stock, and so on). Once written and tested, all parts of the application (or all applications) that need to remove orders from the database will call this procedure, passing the ID of the order as argument.

Functions The functions of SQL Server are just like VB functions: They perform specific tasks on the database (retrieve or update data), taking into consideration the arguments passed to the functions when they were called. ADO.NET is built around SQL statements and stored procedures, so we won't discuss SQL Server functions in this book.

Synonyms Using a synonym lets you refer to a SQL Server object by using another name. Although this might not seem a useful feature, you can use it to make your code more readable. Using synonyms can also protect your code from changes to the objects within the database. The object can change, but the synonym remains the same. For example, you might originally locate the Price table on Server1, but then move the table to Server2 when it gets too large. Your code can still use the same synonym, despite the move from one server to another.

Types The Types entry defines all the types for the database and includes the following folders: System Data Types, User-Defined Data Types, User-Defined Types, and XML Schema Collections. The System Data Types folder contains a list of all the data types that SQL Server natively understands, categorized by type. The User-Defined Data Types folder contains a list of the special data types that you define based on the system data types; for example, you could create a special part number data type to hold company part information. The User-Defined Types folder contains a list of other types that you create to express special database needs. You always define a user-defined type as part of an assembly. Finally, the XML Schema Collections defines the content of the XML data that a database contains.

Assemblies The Assemblies folder contains a list of specialized code modules that you create by using VB 2008. You use assemblies to define unique data-processing requirements that are difficult or impossible to do when using other techniques. An assembly can contain functions, stored procedures, triggers, user-defined types, and user-defined aggregate functions. Generally, you don't need to resort to using assemblies except with complex and large databases. Stored procedures and other standard SQL techniques usually work fine and require less work to implement.

Working with Tables

Expand the connection to the database under the Data Connections item of Server Explorer to view the database's objects. One of them is the Tables item that contains the database's tables. Expand the Tables tree under the connection to the Northwind database you created earlier to see the list of tables in the Northwind database. If you right-click one of them, you will see the following (among other trivial options).

Show Table Data

This command displays the entire table onto a grid. You can edit any row, and even delete rows or add new ones. To experiment with tables, open the Categories table by right-clicking its entry and choosing Show Table Data from the context menu. Select a row by clicking the gray button in front of the row and then click the Delete button. First, you'll be warned that you're about to remove a row and that the action can't be undone. If you click Yes, the row should be removed. If you attempt to remove a row from the Categories table, however, you'll get the following warning:

```
No rows were deleted.

A problem occurred trying to delete row 1.
Error Source: .Net SqlClient Data Provider
Error Message: The DELETE statement conflicted with the
REFERENCE constraint "FK_Products_Categories". The
conflict occurred in database "Northwind", table
"Products", column "CategoryID".
The statement has been terminated.

Correct errors and attempt to delete the row again or
press "ESC" to cancel the change(s).
```

This warning means that there's a constraint in the database that will be violated if you remove this line. The constraint is between the Products and Categories tables. Each product belongs to a category, and if you remove a category, some of the products will be left without a category. The designers of the Northwind database added the appropriate constraints so that users won't accidentally violate the integrity of the database. As you will see, it's easy to add new constraints to a table and to protect the integrity of the database from mistakes of programmers and users alike.

To add a row, press Ctrl+End to go to the last line, place the cursor in the first cell of the row marked with an asterisk (this is the new row), and start typing. To move to the next cell, press Tab. To commit the new row to the database, move the pointer to another row. As long as you're editing a row, the pen icon appears in the first column of the grid. While this icon is displayed, the original row in the table hasn't changed yet. After you're finished entering values, move the pointer to another row, and the pen icon will disappear, indicating that the row has been successfully added. If the data are not consistent with the structure of the database, the changes will be aborted by the database. If you attempt to enter a new product with a category ID that doesn't exist in the Categories table, the new row will not be accepted. Likewise, if you attempt to create a new order for a nonexistent customer, the new row will be rejected. If you click on a cell that belongs to an `Identity` column, like the *`CategoryID`* column of the Categories table, the warning ''Cell is read-only'' will appear in the status bar. This warning tells you that you can't edit the *`CategoryID`* field of the new row; this cell will be assigned a value automatically by the DBMS as soon as you submit the new row to the database.

Open Table Definition

Close the table, return to the Server Explorer, right-click one of the tables, and this time choose Open Table Definition. The table's structure will appear on a grid, as shown in Figure 21.8. The first column contains the table's column names (the fields of each row).

Each column has a name, a data type, and a length. To set the data type of a column, click the Data Type cell of a field. This cell is a ComboBox displaying all available data types.

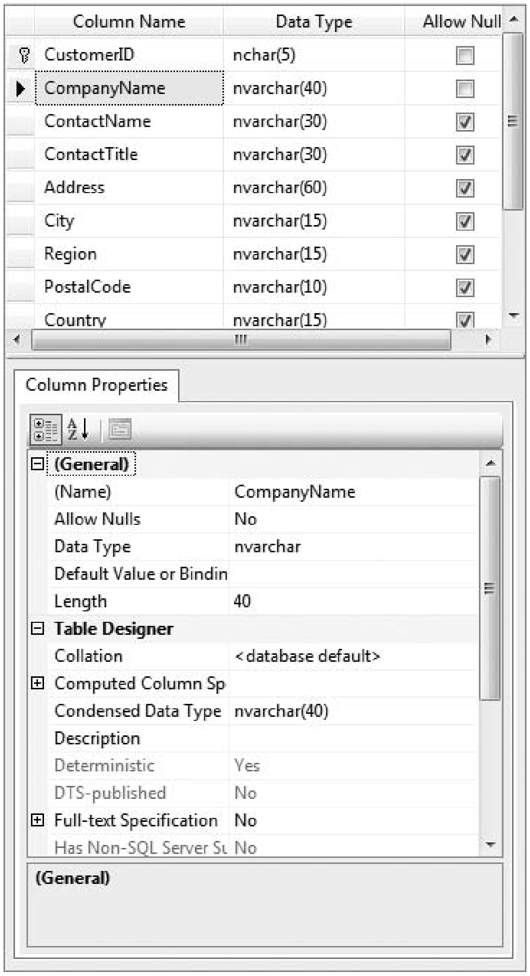
The most common data types are the `char` and `varchar`types, which store strings, and the numeric types, including the `money` data type. There's also a special field for storing dates and times. Basically, you can use all the data types available in VB and a few more data types that are unique to SQL Server.

The difference between the `char` and `varchar` data types is that the `char` type stores strings of fixed length (the length is specified by the value in parentheses following the data type's name), and the `varchar` type stores strings of variable length. The value in parentheses is the maximum allowed length of the string for variable-length strings. Always use the `varchar` type for storing text, because the `char` fields are padded with spaces to the specified length. Use the `char` data type for fields that have an exact length, such as book ISBNs, Social Security numbers, zip codes, and so on.

You can also set the Allow Nulls field to indicate whether a field may have no value. Null is a special value in database programming. It's not the numeric zero and it's not an empty string, as with Visual Basic; Null means that the field has no value.

In the lower section of the window, you see additional information about the selected field. Each field has a Description, a Default Value, and a Collation setting (the last setting applies to character fields only). The Default Value is a value that will be placed in this field automatically when a new row is added to the table, unless a different value is specified. The Collation setting determines how the rows will be sorted, as well as how the database will search for values in the specific column. Normally, you set a collation sequence when you set up the database. You can specify a different sort order for a specific field by setting its Collation property. If you click the ellipses button next to the Collation setting, you'll see a large number of settings. The strings *CS* and *CI* stand for *case-sensitive* and *case-insensitive*, respectively. Searches are usually case-insensitive, so that the argument *McDonald* will locate *MCDonald* and *McDONALD*. The strings *AS* and *AI* stand for *accent-sensitive* and *accent-insensitive*; they're used with languages that recognize accent marks.

FIGURE 21.8
The Customers table in design view



Column Name	Data Type	Allow Null
CustomerID	nchar(5)	☐
CompanyName	nvarchar(40)	☐
ContactName	nvarchar(30)	☑
ContactTitle	nvarchar(30)	☑
Address	nvarchar(60)	☑
City	nvarchar(15)	☑
Region	nvarchar(15)	☑
PostalCode	nvarchar(10)	☑
Country	nvarchar(15)	☑

Column Properties

(General)	
(Name)	CompanyName
Allow Nulls	No
Data Type	nvarchar
Default Value or Bindin	
Length	40
Table Designer	
Collation	<database default>
Computed Column Sp	
Condensed Data Type	nvarchar(40)
Description	
Deterministic	Yes
DTS-published	No
Full-text Specification	No
Has Non-SQL Server Sı	No

(General)

Integer data types have an `Identity` property: In each table, you can have one identity field, which is an integer value. This value is incremented by the DBMS every time a new row is added, and it's guaranteed to be unique. The actual value is of no interest to users; they don't even have to see this field. All you really want is for the primary key in one row to have the same value as the foreign key(s) in the related table(s).

ADD NEW TABLE

This command adds a new table to the database. If you select it, you will see a grid like the one shown in Figure 21.8, only all rows will be empty. You can start adding fields by specifying a name, a data type, and its Allow Nulls property. For the purposes of this book, I assume that the database has already been designed for you. Designing databases is no trivial task, and programmers shouldn't be adding tables to simplify their code. We organize our data into tables, create the tables, and set up relations between them. And only then can we code against the database. It's not uncommon to add a table to an existing database at a later stage, but this reveals some flaw in the initial database design.

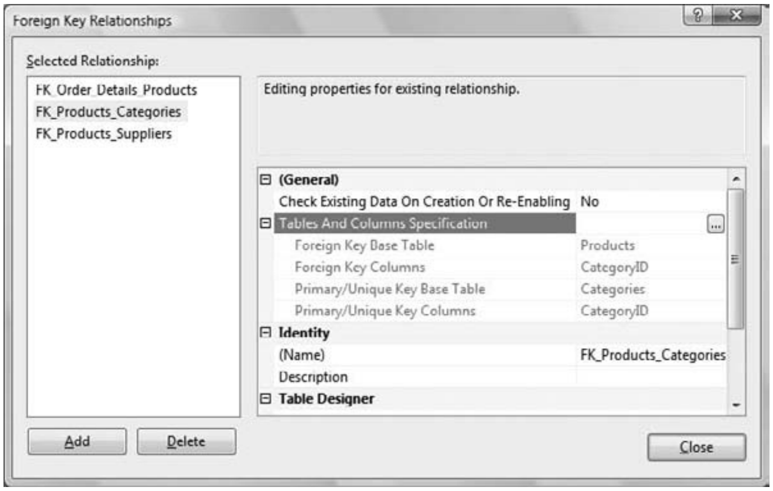
Working with Relationships, Indices, and Constraints

To manipulate relationships, indices, and constraints, open one of the tables in design mode. The Table Designer menu contains a list of tasks you can perform with the table, including changing the relationships, indexes, and constraints.

RELATIONSHIPS

Relationships are the core of a relational database, because they relate tables to one another. To create a relationship, double-click a table's name in Server Explorer and then choose Table Designer ➤ Relationships, which displays the Foreign Key Relationships dialog box shown in Figure 21.9. This figure shows that there is already a relationship between the Categories table and the Products table. The relationship is called `FK_Products_Categories`, and it relates the primary and foreign keys of the two tables (field *CategoryID*). The names of the two related tables appear in two read-only boxes. When you create a new relationship, you can select a table from a drop-down list. Under each table's name, you see a list of fields. Here you select the matching fields in the two tables. Most relationships are based on a single field, which is common to both tables. However, you can relate two tables based on multiple fields (in which case, all pairs must match in a relationship). The check boxes at the bottom of the page specify how the DBMS will handle the relationship (they are discussed shortly).

FIGURE 21.9
The Foreign Key Relationships dialog box for the Categories table



FK_Order_Details_Products
FK_Products_Categories
FK_Products_Suppliers

Editing properties for existing relationship.

(General)	
Check Existing Data On Creation Or Re-Enabling	No
Tables And Columns Specification	
Foreign Key Base Table	Products
Foreign Key Columns	CategoryID
Primary/Unique Key Base Table	Categories
Primary/Unique Key Columns	CategoryID
Identity	
(Name)	FK_Products_Categories
Description	
Table Designer	

To create a new relationship with another table, click the Add button. A new relationship will be added with a default name, which you can change. Like all other objects, relationships have unique names, too. Expand the Tables And Columns Specification entry and click the ellipses button in this field. You'll see the Tables And Columns dialog box. In the Primary Key Table column, you can select the name of the table that has the primary key in the relationship. The Foreign Key Table column always defaults to the current table, so when you create a relationship, you must create it in the table where the foreign key will appear. The default relationship names starts with the string *FK* (which stands for *foreign key*), followed by an underscore character and the name of the foreign table, then followed by another underscore and the name of the primary table. You can change the relationship's name to anything you like.

If the relationship is based on a compound key, select all the fields that make up the primary and foreign keys, in the same order. After you select the fields, click OK to create the relationship. At the right side of the Foreign Key Relationships dialog box, you see a few options that you can change:

Check Existing Data On Creation Or Re-enabling If the existing data violate the relationship, the new relationship won't be established. You will have to fix the data and then attempt to establish the relationship again.

Enforce For Replication The relationship is enforced when the database is replicated.

Enforce Foreign Key Constraint The relationship is enforced when you add new data or update existing data. If you attempt to add data that violate the relationship, the new data (or the update) will be rejected.

Update Rule, Delete Rule When you change the primary key in one table, or delete it, some rows of a related table may be left with an invalid foreign key. If you delete a publisher, for example, all the titles that pointed to this publisher will become invalid after you change the publisher's ID. If you change a publisher's key, you may leave some books without a publisher. You can set this option to take no action at all, automatically cascade the change (or deletion), set the affected data to Null, or set the data to the default value that you selected as part of the design.

You can also create relationships on a database diagram by dragging the primary key field from its table and dropping it onto the foreign key of the related table. Just click the gray header of the primary key to select it, not the name of the field.

To view or edit the details of a relationship, right-click the line that represents the relationship, and you will see the following commands:

Delete Relationship From Database This command removes the relationship between the two tables.

Properties This command brings up the Properties pages of the primary table, in which you can specify additional relationships or constraints.

Earlier in this chapter, you saw that you couldn't remove a row from the Categories table because this action conflicted with the `FK_Products_Categories` constraint. If you open the first diagram you created in this section and examine the properties of the relation between the Product and Categories tables, you'll see that the `FK_Products_Categories` relationship is enforced. If you want to be able to delete categories, you must delete all the products that are associated with the specific category first. SQL Server can take care of deleting the related rows for you if you select Cascade option in the Delete Rule field. This is a rather dangerous practice, and you shouldn't select it without good reason. In the case of the Products table, you shouldn't enable cascade deletions. The products are also linked to the Order Details table, which means that the corresponding detail lines would also disappear from the database. Allowing cascade deletion in a database such as Northwind will result in loss of valuable information irrevocably.

There are other situations in which cascade deletion is not such a critical issue. You can enable cascade deletions in the Pubs database, for instance, so that each time you delete a title, the corresponding rows in the TitleAuthor table will also be removed. When you delete a book, you obviously don't need any information about this book in the TitleAuthor table.

Indices/Keys

You created a few tables and have entered some data into them. Now the most important thing you can do with a database is extract data from it (or else, why store the information in the first place?). We rarely browse the rows of a single table. Instead, we're interested in summary information that will help us make business decisions. We need answers to questions such as ''What's the most popular product in California?'' or ''What month has the largest sales for a specific product?'' To retrieve this type of information, you must combine multiple tables. To answer the first question, you must locate all the customers in California, retrieve their orders, sum the quantities of the items they have purchased, and then select the product with the largest sum of quantities. As you can guess, a DBMS must be able to scan the tables and locate the desired rows quickly.

Computers use a special technique, called *indexing,* to locate information quickly. This technique requires that the data be maintained in some order. The indexed rows need not be in a specific physical order, as long as you can retrieve them in a specific order. Indeed, an index is an ordering of the rows, and you can maintain the same rows sorted in many different ways. Depending on the operation, the DBMS will select the appropriate index to speed up the operation. If you want to retrieve the name of the category of a specific product, the rows of the Categories table must be ordered according to the `CategoryID` field, which is the value that links each row in the Products table to the corresponding row in the Categories table. The DBMS retrieves the `CategoryID` field of a specific product and then instantly locates the matching row in the Categories table because the rows of this table are indexed according to their `CategoryID` field.

Fortunately, you don't have to maintain the rows of the tables in any order yourself. All you have to do is define the order, and the DBMS will maintain the indices for you. Every time a new row is added or an existing row is deleted or edited, the table's indices are automatically updated. To speed up the searches, you can maintain an index for each field you want to search. Of course, although indexing will help the search operations, maintaining too many indices will slow the insertion and update operations. At any rate, all columns that are used in joins must be indexed, or else the selection process will be very slow.

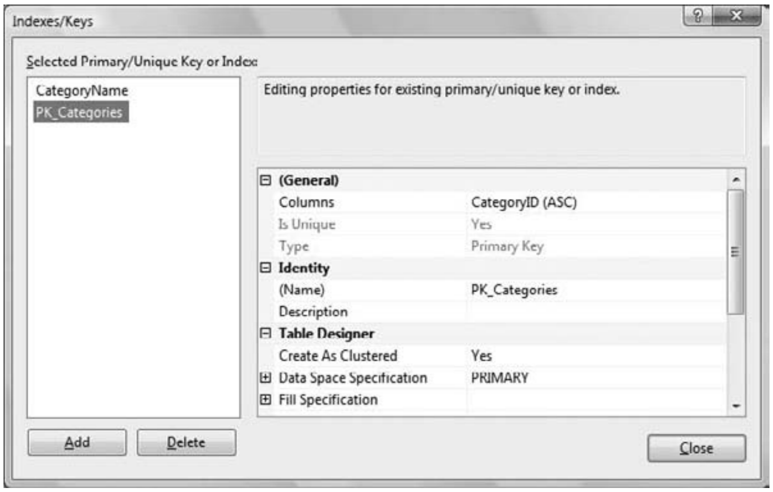
Use the Table Designer ➢ Indexes/Keys command to display the Indexes/Keys dialog box for the Categories table. Figure 21.10 shows the properties of the `PK_Categories` index of the Categories table. This index is based on the column `CategoryID` of the table and maintains the rows of the Categories table in ascending order according to their ID. The prefix *PK* stands for *primary key*. To specify that an index is also the table's primary key, you must set the Is Unique property to Yes. You can create as many indices as necessary for each table, but only one of them can be the primary key. The Is Unique property in Figure 21.10 is disabled because the primary key is involved in one or more relationships — therefore, you can't change the table's primary key because you'll break some of the existing relationships with other tables.

To create a new index, click the Add button. Specify the column on which the new index will be based by clicking the ellipses in the Columns property and choosing the columns in the Index Columns dialog box. Enter a name for the new index (or accept the default one) using the (Name) property.

Check Constraints

A *constraint* is another important object of a database. The entity represented by a field can be subject to physical constraints. The `Discount` field, for example, should be a positive value no greater than 1 (or 100, depending on how you want to store it). Prices are also positive values. Other fields are subject to more-complicated constraints. The DBMS can make sure that the values assigned to those fields do not violate the constraints. Otherwise, you'd have to make sure that all the applications that access the same fields conform to the physical constraints.

FIGURE 21.10
The Indexes/Keys tab of the Properties pages



Selected Primary/Unique Key or Index:

CategoryName
PK_Categories

Editing properties for existing primary/unique key or index.

```
⊟ (General)
    Columns                      CategoryID (ASC)
    Is Unique                    Yes
    Type                         Primary Key
⊟ Identity
    (Name)                       PK_Categories
    Description
⊟ Table Designer
    Create As Clustered          Yes
⊞ Data Space Specification       PRIMARY
⊞ Fill Specification
```

To make a constraint part of the database, open the table that contains the field on which you want to impose a constraint, in design view. Use the Table Designer ➢ Check Constraints command to display the Check Constraints dialog box. The names of the constraints start with the *CK* prefix, followed by an underscore, the name of the table, another underscore, and finally the name of the field to which the constraint applies. The `CK_Products_UnitPrice` constraint is the expression that appears in the Expression property (the *`UnitPrice`* field must be positive):

```
([UnitPrice]>=(0))
```

Constraints have a syntax similar to the syntax of SQL restrictions and are quite trivial. (I'll get into SQL in the following section.) Another interesting constraint exists in the Employees table, and it's expressed as follows:

```
([BirthDate]<GetDate())
```

This constraint prevents users and programs from inserting an employee that hasn't been born yet. `GetDate()`is a built-in function that returns the current date and time.

So far, you should have a good idea about how databases are organized, what the relationships are for, and why they're so critical for the integrity of the data stored in the tables. Now you'll look at ways to retrieve data from a database. To specify the rows and columns you want to retrieve from one or more tables, you must use SQL statements, which are the topic of the following section.

Structured Query Language

Structured Query Language (SQL) is a universal language for manipulating tables. Almost every DBMS supports it, so you should invest the time and effort to learn it. You can generate SQL statements with point-and-click operations (the Query Builder is a visual tool for generating SQL statements), but this is no substitute for understanding SQL and writing your own statements. The visual tools are nothing more than a user-friendly interface for specifying SQL statements. In the background, they generate the appropriate SQL statement, and you will get the most out of these tools if you understand the basics of SQL. I will start with an overview of SQL and then I'll show you how to use the Query Builder utility to specify a few advanced queries.

By the way, the SQL version of SQL Server is called T-SQL, which stands for Transact-SQL. T-SQL is a superset of SQL and provides advanced programming features that are not available with SQL. I'm not going to discuss T-SQL in this book, but once you have understood SQL you'll find it easy to leverage this knowledge to T-SQL.

SQL is a *nonprocedural language,* which means that SQL doesn't provide traditional programming structures such as `If` statements or loops. Instead, it's a language for specifying the operation you want to perform against a database at a high level. The details of the implementation are left to the DBMS. SQL is an *imperative language,* like Language Integrated Query (LINQ), as opposed to a traditional programming language, such as VB. Traditional languages are *declarative:* The statements you write tell the compiler how to perform the desired actions. This is good news for nonprogrammers, but many programmers new to SQL might wish it had the structure of a more traditional language. You will get used to SQL and soon be able to combine the best of both worlds: the programming model of VB and the simplicity of SQL. Besides, there are many similarities between SQL and LINQ, and you'll be able to leverage your skills in any of the two areas.

SQL Is Not Case-Sensitive

SQL is not case-sensitive, but it's customary to use uppercase for SQL statements and keywords. In the examples in this book, I use uppercase for SQL statements. This is just a style to help you distinguish between the SQL keywords and the table/field names of the query. Also, unlike VB, SQL literals must be embedded in single quotes, not double quotes.

To retrieve all the company names from the Customers table of the Northwind database, you issue a statement like this one:

```
SELECT CompanyName
FROM Customers
```

To select customers from a specific country, you must use the `WHERE` clause to limit the selected rows, as in the following statement:

```
SELECT CompanyName
FROM Customers
WHERE Country = 'Germany'
```

The DBMS will retrieve and return the rows you requested. As you can see, this is not the way you'd retrieve rows with Visual Basic. With a procedural language such as VB, you'd have to write loops to scan the entire table, examine the value of the Country column, and either select or reject the row. Then you would display the selected rows. With SQL, you don't have to specify how the selection operation will take place; you simply specify *what* you want the database to do for you — not *how* to do it.

SQL statements are divided into two major categories, which are actually considered separate languages: the statements for manipulating the data, which form the Data Manipulation Language (DML), and the statements for defining database objects, such as tables or their indexes, which form the Data Definition Language (DDL). The DDL is not of interest to every database developer, and we will not discuss it in this book. The DML is covered in depth because you'll use these statements to retrieve data, insert new data into the database, and edit or delete existing data.

The statements of the DML part of the SQL language are also known as *queries,* and there are two types of queries: *selection queries* and *action queries.* Selection queries retrieve information from the database. A selection query returns a set of rows with identical structure. The columns can come from different tables, but all the rows returned by the query have the same number of columns. Action queries modify the database's objects, or create new objects and add them to the database (new tables, relationships, and so on).

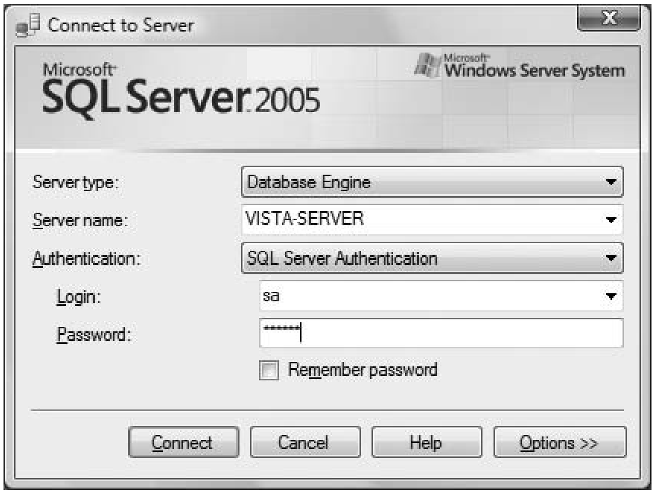
Executing SQL Statements

If you are not familiar with SQL, I suggest that you follow the examples in this chapter and experiment with the sample databases. To follow the examples, you have two options: the SQL Server Management Studio (SSMS) and the Query Designer of Visual Studio. The SSMS helps you manage databases in various ways, including creating queries to extract data. The Query Designer is an editor for SQL statements that also allows you to execute them and see the results. In addition to the Query Designer, you can also use the Query Builder, which is part of the SSMS and Visual Studio. The Query Builder lets you build the statements with visual tools and you don't have to know the syntax of SQL in order to create queries with the Query Builder. After a quick overview of the SQL statements, I will describe the Query Builder and show you how to use its interface to build fairly elaborate queries.

Using the SQL Server Management Studio (SSMS)

One of the applications installed with SQL Server is the SSMS. To start it, choose Start ➢ Programs ➢ SQL Server ➢ SQL Server Management Studio. When this application starts, you see the Connect To Server dialog box (Figure 21.11). Choose Database Engine in the Server Type field so you can work with databases on your system. Select the server you want to use in the Server Name field. Provide your credentials and click Connect.

Figure 21.11
SSMS provides access to all the database engine objects, including databases.


Server type: Database Engine
Server name: VISTA-SERVER
Authentication: SQL Server Authentication
Login: sa
Password: *******
Remember password

After you're connected, right-click the database you want to use and choose New Query from the context menu. Enter the SQL statement you want to execute in the blank query that SSMS creates. The SQL statement will be executed against the selected database when you press Ctrl+E or click the Execute button (it's the button with the exclamation point icon). Alternatively, you can prefix the SQL statement with the USE statement, which specifies the database against which the statement will be executed. To retrieve all the Northwind customers located in Germany, enter this statement:

```
USE Northwind
SELECT CompanyName FROM Customers
WHERE Country = 'Germany'
```

The USE statement isn't part of the query; it simply tells SSMS the database against which it must execute the query. I'm including the USE statement with all the queries so you know the database used for each example. If you're executing the sample code from within Visual Studio, you need not use the USE statement, because all queries are executed against the selected database. Actually, the statement isn't supported by the Query Designer of Visual Studio.

The results of the query, known as the *result set*, will appear in a grid in the lower pane. An action query that updates a table (adds a new row, edits, or deletes an existing row) doesn't return any rows; it simply displays the number of rows affected on the Messages tab.

To execute another query, enter another statement in the upper pane, or edit the previous statement and press Ctrl+E again. You can also save SQL statements into files, so that you won't have to type them again. To do so, open the File menu, choose Save As or Save, and enter the name of the file in which the contents of the Query pane will be stored. The statement will be stored in a text file with the extension .sql.

Using Visual Studio

To execute the same queries with Visual Studio, open the Server Explorer window and right-click the name of the database against which you want to execute the query. From the context menu, choose New Query, and a new query window will open. You will also see a dialog box prompting you to select one or more tables. For the time being, close this dialog box, because you will supply the names of the tables in the query; later in this chapter, you'll learn how to use the visual tools to build queries.

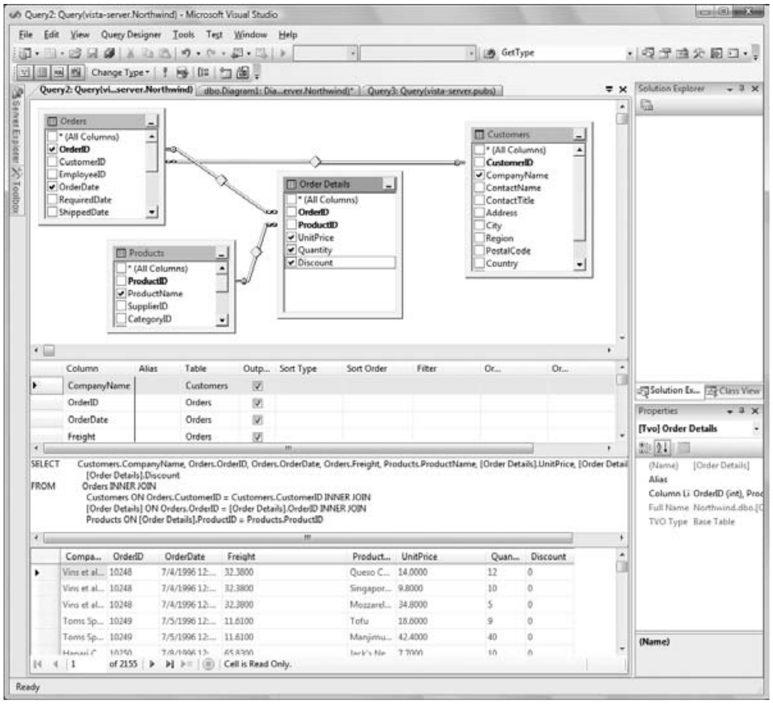
The Query Designer of Visual Studio is made up of four panes (Figure 21.12). The upper pane (which is the Table Diagram pane) displays the tables involved in the query, their fields, and the relationships between the tables — if any. The next pane shows the fields that will be included in the output of the query. Here you specify the output of the query, as well as the selection criteria. This pane is the Query Builder, the tool that lets you design queries visually. It's discussed later in this chapter. In the next pane, the SQL pane, you see the SQL statement produced by the visual tools. If you modify the query with the visual tools, the SQL statement is updated automatically; likewise, when you edit the query, the other two panes are updated automatically to reflect the changes. The last pane, the Results pane, contains a grid with the query's output. Every time you execute the query by clicking the button with the exclamation mark in the toolbar, the bottom pane is populated with the results of the query. For the examples in this section, ignore the top two panes. Just enter the SQL statements in the SQL pane and execute them.

Using Selection Queries

We'll start our discussion of SQL with the SELECT statement. After you learn how to express the criteria for selecting the desired rows with the SELECT statement, you can apply this information to other data-manipulation statements. The simplest form of the SELECT statement is

```
SELECT fields
FROM tables
```

FIGURE 21.12
Executing queries with Visual Studio



Column	Alias	Table	Outp...	Sort Type	Sort Order	Filter	Or...	Or...
CompanyName		Customers	✓					
OrderID		Orders	✓					
OrderDate		Orders	✓					
Freight		Orders	✓					

```
SELECT    Customers.CompanyName, Orders.OrderID, Orders.OrderDate, Orders.Freight, Products.ProductName, [Order Details].UnitPrice, [Order Detai
            [Order Details].Discount
FROM       Orders INNER JOIN
            Customers ON Orders.CustomerID = Customers.CustomerID INNER JOIN
            [Order Details] ON Orders.OrderID = [Order Details].OrderID INNER JOIN
            Products ON [Order Details].ProductID = Products.ProductID
```

Compa...	OrderID	OrderDate	Freight	Product...	UnitPrice	Quan...	Discount
Vins et al...	10248	7/4/1996 12:...	32.3800	Queso C...	14.0000	12	0
Vins et al...	10248	7/4/1996 12:...	32.3800	Singapor...	9.8000	10	0
Vins et al...	10248	7/4/1996 12:...	32.3800	Mozzarel...	34.8000	5	0
Toms Sp...	10249	7/5/1996 12:...	11.6100	Tofu	18.6000	9	0
Toms Sp...	10249	7/5/1996 12:...	11.6100	Manjimu...	42.4000	40	0
Hanari C...	10250	7/8/1996 12:...	65.8300	Jack's Ne...	7.7000	10	0

1 of 2155 Cell is Read Only.

where *fields* and *tables* are comma-separated lists of the fields you want to retrieve from the database and the tables they belong to. The list of fields following the SELECT statement is referred to as the *selection list*. To select the contact information from all the companies in the Customers table, use this statement:

```
USE Northwind
SELECT CompanyName, ContactName, ContactTitle
FROM Customers
```

To retrieve all the fields, use the asterisk (*). The following statement selects all the fields from the Customers table:

```
SELECT * FROM Customers
```

As soon as you execute a statement that uses the asterisk to select all columns, the Query Designer will replace the asterisk with the names of all columns in the table.

LIMITING THE SELECTION WITH *WHERE*

The unconditional form of the SELECT statement used in the previous section is quite trivial. You rarely retrieve data from all rows in a table. Usually you specify criteria, such as ''all companies from Germany,'' ''all customers who have placed three or more orders in the last six months,'' or even more-complicated expressions. To restrict the rows returned by the query, use the WHERE clause of the SELECT statement. The most common form of the SELECT statement is the following:

```
SELECT fields
FROM tables
WHERE condition
```

The *fields* and *tables* arguments are the same as before, and *condition* is an expression that limits the rows to be selected. The syntax of the WHERE clause can get quite complicated, so we'll start with the simpler forms of the selection criteria. The *condition* argument can be a relational expression, such as the ones you use in VB. To select all the customers from Germany, use the following condition:

```
WHERE Country = 'Germany'
```

To select customers from multiple countries, use the OR operator to combine multiple conditions:

```
WHERE Country = 'Germany' OR
      Country = 'Austria'
```

You can also combine multiple conditions with the AND operator.

Selecting Columns from Multiple Tables

It is possible to retrieve data from two or more tables by using a single statement. (This is the most common type of query, actually.) When you combine multiple tables in a query, you can use the WHERE clause to specify how the rows of the two tables will be combined. Let's say you want a list of all product names, along with their categories. For this query, you must extract the product names from the Products table and the category names from the Categories table and specify that the ProductID field in the two tables must match. The statement

```
USE Northwind
SELECT ProductName, CategoryName
FROM Products, Categories
WHERE Products.CategoryID = Categories.CategoryID
```

retrieves the names of all products, along with their category names. Here's how this statement is executed. For each row in the Products table, the SQL engine locates the matching row in the Categories table and then appends the *ProductName* and *CategoryName* fields to the result.

If a product has no category, that product is not included in the result. If you want all the products, even the ones that don't belong to a category, you must use the JOIN keyword, which is described later in this chapter. Using the WHERE clause to combine rows from multiple tables might lead to unexpected results because it can combine rows only with matching fields. If the foreign key in the Products table is Null, this product is not selected. This is a fine point in combining multiple tables, and many programmers abuse the WHERE clause. As a result, they retrieve fewer rows from the database and don't even know it.

Resolving Column Names

When fields in two tables have the same names, you must prefix them with the table's name to remove the ambiguity. When you execute a SQL statement, the Query Builder automatically prefixes all column names with the name of the table they belong to. Also, some field names might contain spaces. These field names must appear in square brackets. The Publishers table of the Pubs sample database contains a field named `Publisher Name`. To use this field in a query, enclose it in brackets: `Publishers.[Publisher Name]`. The table prefix is optional (no other table contains a column by that name), but the brackets are mandatory.

To retrieve all the titles published by a specific publisher, the New Moon Books publisher, use a statement like the following:

```
USE pubs
SELECT titles.title
FROM titles, publishers
WHERE titles.pub_id = publishers.pub_id AND publishers.pub_name = 'New Moon Books'
```

This statement combines two tables and selects the titles of a publisher specified by name. To match titles and publisher, it requests the following:

- The publisher's name in the Publishers table should be New Moon Books.
- The *pub_id* field in the Titles table should match the *pub_id* field in the Publishers table.

Knowing *WHERE* You're Going

If you specify multiple tables without the `WHERE` clause, the SQL statement will return an enormous set of rows, which is known as a *cursor*. If you issue the following statement, you will not get a line for each product name followed by its category:

```
SELECT ProductName, CategoryName FROM Categories, Products
```

You will get a cursor with 616 rows, which are all possible combinations of product names and category names. In this example, the Categories table has 8 rows, and the Products table has 77 rows, so their cross-product contains 616 rows. It's extremely rare to request the cross-product of two tables. If the two tables have many rows, you will have to stop the execution of the query by clicking the round button with the red square in the status bar of the Query Designer window, next to the number of selected rows.

Notice that we did not specify the publisher's name (field *pub_name*) in the selection list. All the desired books have the same publisher, so we need not include the publisher's names in the result set.

Aliasing Table Names

To avoid typing long table names, you can alias them with a shorter name and use this shorthand notation in the rest of the query. The query that retrieves titles and publishers can be written as follows:

```
USE pubs
SELECT T.title
FROM titles T, publishers P
WHERE T.pub_id = P.pub_id AND
      P.pub_name = 'New Moon Books'
```

The table names are aliased in the FROM clause, and the alias is used in the rest of the query. You can use the AS keyword, but this is optional:

```
FROM titles AS T, publishers AS P
```

Aliasing Column Names with *AS*

By default, each column of a query is labeled after the actual field name in the output. If a table contains two fields named *CustLName* and *CustFName*, you can display them with different labels by using the AS keyword. The following *SELECT* statement produces two columns labeled *CustLName* and CustFName:

```
SELECT CustLName, CustFName
```

The query's output looks much better if you change the labels of these two columns with a statement like the following:

```
SELECT CustLName AS [Last Name],
       CustFName AS [First Name]
```

It is also possible to concatenate two fields in the SELECT list with the concatenation operator. Concatenated fields are labeled automatically as *Expr1*, *Expr2*, and so on, so you must supply your own name for the combined field. The following statement creates a single column for the customer's name and labels it *Customer Name*:

```
SELECT CustLName + ', ' + CustFName AS [Customer Name]
```

Skipping Duplicates with *DISTINCT*

The DISTINCT keyword eliminates from the cursor any duplicates retrieved by the SELECT statement. Let's say you want a list of all countries with at least one customer. If you retrieve all country names from the Customers table, you'll end up with many duplicates. To eliminate them, use the DISTINCT keyword, as shown in the following statement:

```
USE Northwind
SELECT DISTINCT Country
FROM Customers
```

The *LIKE* Operator

The LIKE operator uses pattern-matching characters like the ones you use to select multiple files in DOS. The LIKE operator recognizes several pattern-matching characters (or *wildcard* characters) to match one or more characters, numeric digits, ranges of letters, and so on. These characters are listed in Table 21.1.

Table 21.1: SQL Wildcard Characters

Wildcard Character	Description
%	Matches any number of characters. The pattern `program%` will find *program*, *programming*, *programmer*, and so on. The pattern `%program%` will locate strings that contain the words *program, programming, nonprogrammer*, and so on.
_	(Underscore character.) Matches any single alphabetic character. The pattern `b_y` will find *boy* and *bay*, but not *boysenberry*.
[ ]	Matches any single character within the brackets. The pattern `Santa [YI]nez` will find both *Santa Ynez* and *Santa Inez*.
[^]	Matches any character not in the brackets. The pattern `%q[^u]%` will find words that contain the character *q* not followed by *u* (they are misspelled words).
[-]	Matches any one of a range of characters. The characters must be consecutive in the alphabet and specified in ascending order (*A* to *Z*, not *Z* to *A*). The pattern `[a-c]%` will find all words that begin with *a*, *b*, or *c* (in lowercase or uppercase).
#	Matches any single numeric character. The pattern `D1##` will find *D100* and *D139*, but not *D1000* or *D10*.

You can use the LIKE operator to retrieve all titles about Windows from the Pubs database, by using a statement like the following one:

```
USE pubs
SELECT titles.title
FROM titles
WHERE titles.title LIKE '%Windows%'
```

The percent signs mean that any character(s) may appear in front of or after the word *Windows* in the title.

To include a wildcard character itself in your search argument, enclose it in square brackets. The pattern `%50[%]%` will match any field that contains the string *50%*.

Null Values and the *ISNULL* Function

A common operation for manipulating and maintaining databases is to locate Null values in fields. The expressions IS NULL and IS NOT NULL find field values that are (or are not) Null. To locate the rows of the Customers table that have a Null value in their *CompanyName* column, use the following WHERE clause:

```
WHERE CompanyName IS NULL
```

You can easily locate the products without prices and edit them. The following statement locates products without prices:

```
USE Northwind
SELECT * FROM Products WHERE UnitPrice IS NULL
```

A related function, the ISNULL() function, allows you to specify the value to be returned when a specific field is Null. The ISNULL() SQL function accepts two arguments: a column name and a string. The function returns the value of the specified column, unless this value is Null, in which case it returns the value of the second argument. To return the string *** for customers without a company name, use the following expression:

```
USE Northwind
SELECT CustomerID,
ISNULL(CompanyName, '***') AS Company,
       ContactName
FROM Customers
```

Sorting the Rows with *ORDER BY*

The rows of a query are not in any particular order. To request that the rows be returned in a specific order, use the ORDER BY clause, which has this syntax:

```
ORDER BY col1, col2, . . .
```

You can specify any number of columns in the ORDER BY list. The output of the query is ordered according to the values of the first column. If two rows have identical values in this column, they are sorted according to the second column, and so on. The following statement displays the customers ordered by country and then by city within each country:

```
USE Northwind
SELECT CompanyName, ContactName, Country, City
FROM Customers
ORDER BY Country, City
```

Limiting the Number of Rows with *TOP*

Some queries retrieve a large number of rows, but you're interested in the top few rows only. The TOP *N* keyword allows you to select the first N rows and ignore the remaining ones. Let's say you want to see the list of the 10 top-selling products. Retrieve the products and the number of items sold for each item, order the rows according to the number of items sold, and keep the first 10 rows with the TOP keyword. To limit the number of rows returned by the query, specify the TOP keyword followed by the desired number of rows after the SELECT statement, as shown here:

```
SELECT TOP 10
FROM ...
```

You can also limit the percentage of the selected rows, not just their absolute number. To return the top 3 percent of the qualifying rows, use the following statement:

```
SELECT TOP (3) PERCENT
FROM ...
```

The TOP keyword is used only when the rows are ordered according to some meaningful criteria. Limiting a query's output to the alphabetically top *N* rows isn't very practical. However, when the rows are sorted according to items sold, revenue generated, and so on, it makes sense to limit the query's output to *N* rows.

Working with Calculated Fields

In addition to column names, you can specify calculated columns in the SELECT statement. The Order Details table contains a row for each invoice line. Invoice 10248, for instance, contains four lines (four items sold), and each detail line appears in a separate row in the Order Details table. Each row holds the number of items sold, the item's price, and the corresponding discount. To display the line's subtotal, you must multiply the quantity by the price minus the discount, as shown in the following statement:

```
USE Northwind
SELECT Orders.OrderID, [Order Details].ProductID,
       [Order Details].[Order Details].UnitPrice *
       [Order Details].Quantity *
       (1 - [Order Details].Discount) AS SubTotal
FROM   Orders INNER JOIN [Order Details]
  ON  Orders.OrderID = [Order Details].OrderID
```

(Because the Order Details table's name contains spaces, it's embedded in square brackets).

Here the selection list contains an expression based on several fields of the Order Details table. This statement calculates the subtotal for each line in the invoices issued to all Northwind customers and displays them along with the order number, as shown in Figure 21.13. The order numbers are repeated as many times as there are products in the order (or lines in the invoice). In the following section, ''Calculating Aggregates,'' you will find out how to calculate totals too.

Calculating Aggregates

SQL supports some aggregate functions, which act on selected fields of all the rows returned by the query. The basic aggregate functions listed in Table 21.2 perform basic calculations such as summing, counting, and averaging numeric values. There are a few more aggregate functions for calculating statistics such as the variance and standard deviation, but I have omitted them from Table 21.2. Aggregate functions accept field names (or calculated fields) as arguments and return a single value, which is the sum (or average) of all values.

These functions operate on a single column (which could be a calculated column) and return a single value. The rows involved in the calculations are specified with the proper WHERE clause. The SUM() and AVG() functions can process only numeric values. The other three functions can process both numeric and text values.

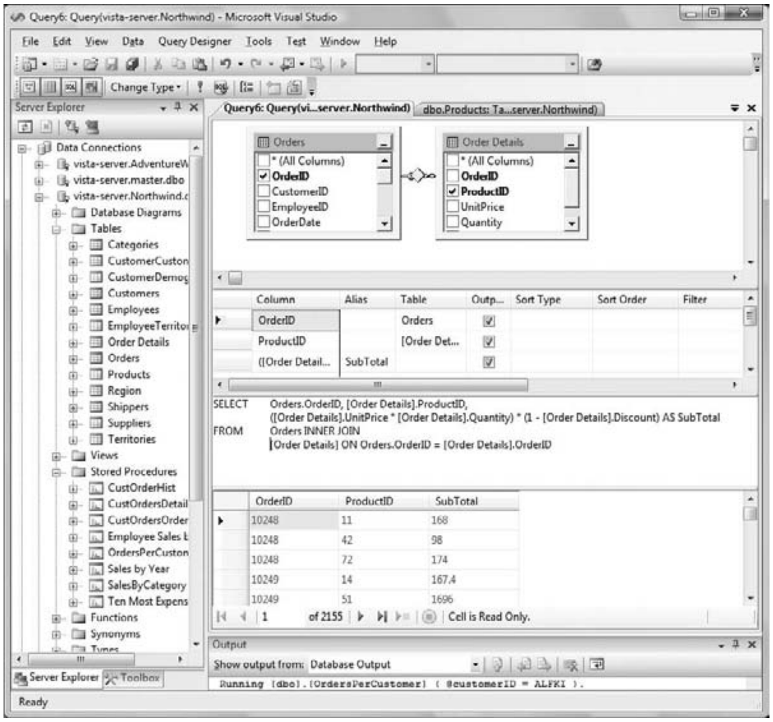


FIGURE 21.13
Calculating the subtotals for each item sold

Column	Alias	Table	Outp...	Sort Type	Sort Order	Filter
OrderID		Orders	✓			
ProductID		[Order Det...	✓			
([Order Detail...	SubTotal		✓			

```
SELECT    Orders.OrderID, [Order Details].ProductID,
          ([Order Details].UnitPrice * [Order Details].Quantity) * (1 - [Order Details].Discount) AS SubTotal
FROM      Orders INNER JOIN
          [Order Details] ON Orders.OrderID = [Order Details].OrderID
```

OrderID	ProductID	SubTotal
10248	11	168
10248	42	98
10248	72	174
10249	14	167.4
10249	51	1696

```
Running [dbo].[OrdersPerCustomer] ( @customerID = ALFKI ).
```

TABLE 21.2: SQL's Common Aggregate Functions

FUNCTION	RETURNS
`COUNT()`	The number (count) of values in a specified column
`SUM()`	The sum of values in a specified column
`AVG()`	The average of the values in a specified column
`MIN()`	The smallest value in a specified column
`MAX()`	The largest value in a specified column

The aggregate functions are used to summarize data from one or more tables. Let's say you want to know the number of Northwind database customers located in Germany. The following SQL statement returns the desired value:

```
USE Northwind
SELECT COUNT(CustomerID)
FROM Customers
WHERE Country = 'Germany'
```

The aggregate functions ignore the Null values unless you specify the * argument. The following statement returns the count of all rows in the Customers table, even if some of them have a Null value in the Country column:

```
USE Northwind
SELECT COUNT(*)
FROM Customers
```

The SUM() function is used to total the values of a specific field in the specified rows. To find out how many units of the product with ID = 11 (queso Cabrales) have been sold, use the following statement:

```
USE Northwind
SELECT SUM(Quantity)
FROM [Order Details]
WHERE ProductID = 11
```

The SQL statement that returns the total revenue generated by a single product is a bit more complicated. To calculate it, you must multiply the quantities by their prices and then add the resulting products together, taking into consideration each invoice's discount:

```
USE Northwind
SELECT SUM(Quantity * UnitPrice * (1 - Discount))
FROM [Order Details]
WHERE ProductID = 11
```

Queso Cabrales generated a total revenue of $12,901.77. If you want to know the number of items of this product that were sold, add one more aggregate function to the query to sum the quantities of each row that refers to the specific product ID:

```
USE Northwind
SELECT SUM(Quantity),
       SUM(Quantity * UnitPrice * (1 - Discount))
FROM   [Order Details]
WHERE  ProductID = 11
```

If you add the *ProductID* column in the selection list and delete the WHERE clause to retrieve the totals for all products, the query will generate an error message to the effect that the columns haven't been grouped. You will learn how to group the results a little later in this chapter.

Using SQL Joins

Joins specify how you connect multiple tables in a query. There are four types of joins:

- Left outer, or left, join
- Right outer, or right, join
- Full outer, or full, join
- Inner join

A join operation combines all the rows of one table with the rows of another table. Joins are usually followed by a condition that determines which records on either side of the join appear in the result. The WHERE clause of the SELECT statement is similar to a join, but there are some fine points that will be explained momentarily.

The left, right, and full joins are sometimes called *outer joins* to differentiate them from an inner join. *Left join* and *left outer join* mean the same thing, as do *right join* and *right outer join.*

Left Joins

The *left join* displays all the records in the left table and only those records of the table on the right that match certain user-supplied criteria. This join has the following syntax:

```
FROM (primary table) LEFT JOIN (secondary table) ON
(primary table).(field) = (secondary table).(field)
```

The left outer join retrieves all rows in the primary table and the matching rows from a secondary table. The following statement retrieves all the titles from the Pubs database along with their publisher. If some titles have no publisher, they will be included in the result:

```
USE pubs
SELECT  title, pub_name
FROM    titles LEFT JOIN publishers
           ON titles.pub_id = publishers.pub_id
```

Right Joins

The *right join* is similar to the left outer join, except that it selects all rows in the table on the right, and only the matching rows from the left table. This join has the following syntax:

```
FROM (secondary table) RIGHT JOIN (primary table)
ON (secondary table).(field) = (primary table).(field)
```

The following statement retrieves all the publishers from the Pubs database along with their titles. If a publisher has no titles, the publisher name will be included in the result set. Notice that this statement is almost exactly the same as the example of the left outer join entry. I changed only LEFT to RIGHT:

```
USE pubs
SELECT  title, pub_name
FROM    titles RIGHT JOIN publishers
           ON titles.pub_id = publishers.pub_id
```

Full Joins

The *full join* returns all the rows of the two tables, regardless of whether there are matching rows. In effect, it's a combination of left and right joins. To retrieve all titles and all publishers, and to match publishers to their titles, use the following join:

```
USE pubs
SELECT  title, pub_name
```

```
FROM    titles FULL JOIN publishers
           ON titles.pub_id = publishers.pub_id
```

This query will include titles without a publisher, as well as publishers without a title.

Inner Joins

The *inner join* returns the matching rows of both tables, similar to the WHERE clause, and has the following syntax:

```
FROM (primary table) INNER JOIN (secondary table)
ON (primary table).(field) = (secondary table).(field)
```

The following SQL statement combines records from the Titles and Publishers tables of the Pubs database if their *pub_id* fields match. It returns all the titles and their publishers. Titles without publishers, or publishers without titles, will not be included in the result.

```
USE pubs
SELECT titles.title, publishers.pub_name FROM titles, publishers
WHERE titles.pub_id = publishers.pub_id
```

You can retrieve the same rows by using an inner join, as follows:

```
USE pubs
SELECT titles.title, publishers.pub_name
FROM titles INNER JOIN publishers ON titles.pub_id = publishers.pub_id
```

Real World Scenario

Do Not Join Tables with the *WHERE* Clause

The proper method of retrieving rows from multiple tables is to use joins. It's not uncommon to write a dozen joins one after the other (if you have that many tables to join). You can also join two tables by using the WHERE clause. Here are two statements that return the total revenue for each of the customers in the Northwind database. The first one uses the INNER JOIN statement, and the second one uses the WHERE clause. The INNER JOIN is equivalent to the WHERE clause: they both return the same rows.

Query 1

```
SELECT
     C.CompanyName,
     SUM((OD.UnitPrice * OD.Quantity) * (1 - OD.Discount)) AS Revenue
FROM Customers AS C
     INNER JOIN Orders AS O ON C.CustomerID = O.CustomerID
     INNER JOIN [Order Details] AS OD ON O.OrderID = OD.OrderID
GROUP BY C.CompanyName
```

Query 2

```
SELECT
    C.CompanyName,
    SUM((OD.UnitPrice * OD.Quantity) * (1 - OD.Discount)) AS Revenue
FROM Customers AS C
     INNER JOIN Orders AS O ON C.CustomerID = O.CustomerID
     INNER JOIN [Order Details] AS OD ON O.OrderID = OD.OrderID
GROUP BY C.CompanyName
```

Both statements assume that all customers have placed an order. If you change the INNER JOIN in the first statement to a LEFT JOIN, the result will contain two more rows: The customers FISSA and PARIS have not placed any orders and they're not included in the output. If you know that all your customers have placed an order, or you don't care about customers without orders, use the WHERE clause or an inner join. It's important to keep in mind that if you want to see all customers, regardless of whether they have placed an order, you must use joins.

An even better example is that of retrieving titles along with their authors. An inner join will return titles that have one (or more) authors. A left join will return all titles, even the ones without authors. A right join will return all authors, even if some of them are not associated with any titles. Finally, a full outer join will return both titles without authors and authors without titles. Here's the statement that retrieves titles and authors from the Pubs database. Change the type of joins to see how they affect the result set:

```
SELECT   titles.title,
         authors.au_lname + ', ' + authors.au_fname AS Author
FROM     authors
   INNER JOIN titleauthor ON authors.au_id = titleauthor.au_id
   INNER JOIN titles ON titleauthor.title_id = titles.title_id
ORDER BY titles.title
```

There's a shorthand notation for specifying left and right joins with the WHERE clause. When you use the operator *= in a WHERE clause, a left join will be created. Likewise, the =* operator is equivalent to a right join.

Grouping Rows

Sometimes you need to group the results of a query so that you can calculate subtotals. Let's say you need not only the total revenues generated by a single product, but a list of all products and the revenues they generated. The example of the earlier section ''Calculating Aggregates'' calculates the total revenue generated by a single product. It is possible to use the SUM() function to break the calculations at each new product ID, as demonstrated in the following statement. To do so, you must group the product IDs together with the GROUP BY clause:

```
USE Northwind
SELECT   ProductID,
         SUM(Quantity * UnitPrice *(1 - Discount)) AS [Total Revenues]
```

```
FROM     [Order Details]
GROUP BY ProductID
ORDER BY ProductID
```

The preceding statement produces the following output:

```
ProductID   Total Revenues
1           12788.10
2           16355.96
3            3044.0
4            8567.89
5            5347.20
6            7137.0
7           22044.29
```

The aggregate functions work in tandem with the GROUP BY clause (when there is one) to produce subtotals. The GROUP BY clause groups all the rows with the same values in the specified column and forces the aggregate functions to act on each group separately. SQL Server sorts the rows according to the column specified in the GROUP BY clause and starts calculating the aggregate functions. Every time it runs into a new group, it generates a new row and resets the aggregate function(s).

If you use the GROUP BY clause in a SQL statement, you must be aware of the following rule:

All the fields included in the SELECT *list must be either part of an aggregate function or part of the* GROUP BY *clause.*

Let's say you want to change the previous statement to display the names of the products rather than their IDs. The following statement does just that. Notice that the *ProductName* field doesn't appear as an argument to an aggregate function, so it must be part of the GROUP BY clause:

```
USE Northwind
SELECT   ProductName,
         SUM(Quantity * [Order Details].UnitPrice * (1 - Discount))
            AS [Total Revenues]
FROM     [Order Details], Products
WHERE    Products.ProductID = [Order Details].ProductID
GROUP BY ProductName
ORDER BY ProductName
```

These are the first few lines of the output produced by this statement:

```
ProductName            Total Revenues
Alice Mutton            32698.38
Aniseed Syrup           3044.0
Boston Crab Meat        17910.63
Camembert Pierrot       46927.48
Carnarvon Tigers        29171.87
```

If you omit the GROUP BY clause, the query will generate an error message indicating that the *ProductName* column in the selection list is not involved in an aggregate or a GROUP BY clause.

You can also combine multiple aggregate functions in the selection list. The following statement calculates the total number of items sold for each product, along with the revenue generated and the number of invoices that contain the specific product:

```
USE Northwind
SELECT   ProductID AS Product,
         COUNT(ProductID) AS Invoices,
         SUM(Quantity) AS [Units Sold],
         SUM(Quantity * UnitPrice *(1 - Discount)) AS Revenue
FROM     [Order Details]
GROUP BY ProductID
ORDER BY ProductID
```

Here are the first few lines returned by the preceding query:

```
Product   Invoices    Units Sold   Revenue
1           38           828       12788.1000595092
2           44          1057       16355.9600448608
```

You should try to revise the preceding statement so that it displays product names instead of IDs, by adding another join to the query as explained already.

Limiting Groups with *HAVING*

The HAVING clause limits the groups that will appear at the cursor. In a way, it is similar to the WHERE clause, but the HAVING clause is used with aggregate functions and the GROUP BY clause, and the expression used with the HAVING clause usually involves one or more aggregates. The following statement returns the IDs of the products whose sales exceed 1,000 units:

```
USE NORTHWIND
SELECT ProductID, SUM(Quantity)
FROM [Order Details]
GROUP BY ProductID
HAVING SUM(Quantity) > 1000
```

You can't use the WHERE clause here, because no aggregates may appear in the WHERE clause. To see product names instead of IDs, join the Order Details table to the Products table by matching their *ProductID* columns. Note that the expression in the HAVING clause need not be included in the selection list. You can change the previous statement to retrieve the total quantities sold with a discount of 10 percent or more with the following HAVING clause:

```
HAVING Discount >= 0.1
```

However, the Discount column must be included in the GROUP BY clause, because it's not part of an aggregate.

Selecting Groups with *IN* and *NOT IN*

The IN and NOT IN keywords are used in a WHERE clause to specify a list of values that a column must match (or not match). They are more of a shorthand notation for multiple OR operators. The following statement retrieves the names of the customers in all German-speaking countries:

```
USE Northwind
SELECT CompanyName
FROM Customers
WHERE Country IN ('Germany', 'Austria', 'Switzerland')
```

Selecting Ranges with *BETWEEN*

The BETWEEN keyword lets you specify a range of values and limit the selection to the rows that have a specific column in this range. The BETWEEN keyword is a shorthand notation for an expression like this:

```
column >= minValue AND column <= maxValue
```

To retrieve the orders placed in 1997, use the following statement:

```
USE Northwind
SELECT OrderID, OrderDate, CompanyName
FROM   Orders, Customers
WHERE  Orders.CustomerID = Customers.CustomerID AND
       (OrderDate BETWEEN '1/1/1997' AND '12/31/1997')
```

Action Queries

In addition to the selection queries we examined so far, you can also execute queries that alter the data in the database's tables. These queries are called *action queries,* and they're quite simple compared with the selection queries. There are three types of actions you can perform against a database: insertions of new rows, deletions of existing rows, and updates (edits) of existing rows. For each type of action, there's a SQL statement, appropriately named INSERT, DELETE, and UPDATE. Their syntax is very simple, and the only complication is how you specify the affected rows (for deletions and updates). As you can guess, the rows to be affected are specified with a WHERE clause, followed by the criteria discussed with selection queries.

The first difference between action and selection queries is that action queries don't return any rows. They return the number of rows affected, but you can disable this feature by calling the following statement:

```
SET NOCOUNT ON
```

This statement can be used when working with a SQL Server database. Let's look at the syntax of the three action SQL statements, starting with the simplest: the DELETE statement.

Deleting Rows

The DELETE statement deletes one or more rows from a table; its syntax is as follows:

```
DELETE table_name WHERE criteria
```

The WHERE clause specifies the criteria that the rows must meet in order to be deleted. The criteria expression is no different from the criteria you specify in the WHERE clause of the selection query. To delete the orders placed before 1998, use a statement like this one:

```
USE Northwind
DELETE Orders
WHERE  OrderDate < '1/1/1998'
```

Of course, the specified rows will be deleted only if the Orders table allows cascade deletions or if the rows to be deleted are not linked to related rows. If you attempt to execute the preceding query, you'll get an error with the following description:

```
The DELETE statement conflicted with the REFERENCE
constraint "FK_Order_Details_Orders". The conflict
occurred in database "Northwind",
table "dbo.Order Details", column 'OrderID'.
```

This error message tells you that you can't delete rows from the Orders table that are referenced by rows in the Order Details table. If you were allowed to delete rows from the Orders table, some rows in the related table would remain *orphaned* (they would refer to an order that doesn't exist). To delete rows from the Orders table, you must first delete the related rows from the Order Details table, and then delete the same rows from the Orders table. Here are the statements that will delete orders placed before 1998. (Do not execute this query unless you're willing to reinstall the Northwind database; there's no undo feature when executing SQL statements against a database.):

```
USE Northwind
DELETE [Order Details]
WHERE (OrderID IN
        (SELECT OrderID
         FROM   Orders
         WHERE  (OrderDate < '1/1/1998')))

DELETE Orders WHERE OrderDate < '1/1/1998'
```

As you can see, the operation takes two action queries: one to delete rows from the Order Details table, and another to delete the corresponding rows from the Orders table.

The DELETE statement returns the number of rows deleted. You can retrieve a table with the deleted rows by using the OUTPUT clause:

```
DELETE Customers
OUTPUT DELETED.*
WHERE Country IS NULL
```

To test the OUTPUT clause, insert a few fake rows in the Customers table:

```
INSERT Customers (CustomerID, CompanyName)
VALUES ('AAAAA', 'Company A)
INSERT Customers (CustomerID, CompanyName)
VALUES ('BBBBB', 'Company B)
```

And then delete them with the following statement:

```
DELETE Customers
OUTPUT DELETED.*
WHERE Country IS NULL
```

If you execute the preceding statements, the deleted rows will be returned as the output of the query. If you want to be safe, you can insert the deleted rows into a temporary table, so you can insert them back into the database (should you delete more rows than intended). My suggestion is that you first execute a selection query that returns the rows you plan to delete, examine the output of this query, and, if you see only the rows you want to delete and no more, write a DELETE statement with the same WHERE clause. To insert the deleted rows to a temporary table, use the INSERT INTO statement, which is described in the following section.

Inserting New Rows

The INSERT statement inserts new rows in a table; its syntax is as follows:

```
INSERT table_name (column_names) VALUES (values)
```

column_names and *values* are comma-separated lists of columns and their respective values. Values are mapped to their columns by the order in which they appear in the two lists.

Notice that you don't have to specify values for all columns in the table, but the *values* list must contain as many items as there are column names in the first list. To add a new row to the Customers table, use a statement like the following:

```
INSERT Customers (CustomerID, CompanyName) VALUES ('FRYOG', 'Fruit & Yogurt')
```

This statement inserts a new row, provided that the FRYOG key isn't already in use. Only two of the new row's columns are set, and they're the columns that can't accept Null values.

If you want to specify values for all the columns of the new row, you can omit the list of columns. The following statement retrieves a number of rows from the Products table and inserts them into the SelectedProducts table, which has the exact same structure:

```
INSERT INTO SelectedProducts VALUES (values)
```

If the values come from a table, you can replace the VALUES keyword with a SELECT statement:

```
INSERT INTO SelectedProducts
       SELECT * FROM Products WHERE CategoryID = 4
```

The INSERT INTO statement allows you to select columns from one table and insert them into another one. The second table must have the same structure as the output of the selection query. Note that you need not create the new table ahead of time; you can create a new table with the CREATE TABLE statement. The following statement creates a new table to accept the CustomerID, CompanyName, and ContactName columns of the Customers table:

```
DECLARE @tbl table
    (ID char(5),
     name varchar(100),
     contact varchar(100))
```

After the table has been created, you can populate it with the appropriate fields of the deleted rows:

```
DELETE Customers
OUTPUT DELETED.CustomerID,
       DELETED.CompanyName, DELETED.ContactName
INTO @tbl
WHERE Country IS NULL
SELECT * FROM @tbl
```

Execute these statements and you will see in the Results pane the two rows that were inserted momentarily into the Customers table and then immediately deleted.

Editing Existing Rows

The UPDATE statement edits a row's fields; its syntax is the following:

```
UPDATE table_name SET field1 = value1, field2 = value2,. . .
WHERE criteria
```

The *criteria* expression is no different from the criteria you specify in the WHERE clause of selection query. To change the country from *UK* to *United Kingdom* in the Customers table, use the following statement:

```
UPDATE Customers SET Country='United Kingdom'
WHERE  Country = 'UK'
```

This statement will locate all the rows in the Customers table that meet the specified criteria (their Country field is UK) and change this field's value to United Kingdom.

The Query Builder

The *Query Builder* is a visual tool for building SQL statements, and it's available with both SQL Server Management Studio (SSMS) and Visual Studio. It's a highly useful tool that generates SQL statements for you — you just specify the data you want to retrieve with point-and-click operations, instead of typing complicated expressions. A basic understanding of SQL is obviously required, which is why I described the basic keywords of SQL in the previous section, but it is possible to build SQL queries with the Query Builder without knowing anything about SQL.

I suggest you use this tool to quickly build SQL statements, but don't expect it to do your work for you. It's a great tool for beginners, but you can't get far by ignoring SQL. The Query Builder is also a great tool for learning SQL because you specify the query with point-and-click operations and the Query Builder builds the appropriate SQL statement. You can also edit the SQL statement manually and see how the other panes are affected.

When working in SSMS, you can click Design Query In Editor on the SQL Editor toolbar or you can use the Query ➢ Design Query In Editor command. Using either of these methods creates a new query. You can also right-click the Views folder for a particular database and choose New View from the context menu. You can also create new queries by creating a new view. A view is the result of a query: It's a virtual table that consists of columns from one or more tables selected with a SQL `SELECT` statement. The Query Builder's window is shown earlier in Figure 21.12.

The Query Builder Interface

As mentioned earlier, the Query Builder contains four panes: Diagram, Criteria, SQL, and Results. You can open or close any of these panes by clicking the Show Diagram Pane, Show Criteria Pane, Show SQL Pane, and Show Results Pane buttons on the Query Builder toolbar.

Diagram Pane

In the Diagram pane, you can select the tables you want to use in your queries — the tables in which the required data reside. To select a table, right-click anywhere on the Diagram pane and choose Add Table from the context menu. You will see the Add Table dialog box. Select as many tables as you need and then close the dialog box.

The selected tables appear on the Diagram pane as small boxes, along with their fields, as shown earlier in Figure 21.12. The tables involved in the query are related to one another, and the relations are indicated as lines between the tables. These lines connect the primary and foreign keys of the relation. The symbol of a key at one end of the line shows the primary key of the relationship, and the other end of the arrow is either a key (indicating a one-to-one relationship) or the infinity symbol (indicating a one-to-many relationship).

The little shape in the middle of the line indicates the type of join that must be performed on the two tables, and it can take several shapes. To change the type of the relation, you can right-click the shape and choose one of the options in the context menu when working in SSMS. When working in Visual Studio, you select the relation and change the type by using the Properties window. The diamond-shaped icon that you can see in Figure 21.12 indicates an inner join, which requires that only rows with matching primary and foreign keys will be retrieved. By default, the Query Builder treats all joins as inner joins, but you can change the type of join.

The first step in building a query is the selection of the fields that will be included in the result. Select the fields you want to include in your query by selecting the check box in front of their names, in the corresponding tables. As you select and deselect fields, their names appear in the Criteria pane. Notice that all fields are prefixed by the name of the table they came from, so there will be no ambiguities.

Right-click the Diagram pane and choose Add Table. In the dialog box that pops up, select the Products and Categories tables from the Tables tab, click Add, and then click Close to close the dialog box.

Criteria Pane

The Criteria pane contains the selected fields. Some fields might not be part of the output — you can use them only for selection purposes — but their names appear in this pane. To exclude them from the output, clear the check box in the Output column.

The Alias column contains a name for the field. By default, the column's name is the alias. This is the heading of each column in the output, and you can change the default name to any string that suits you.

SQL Pane

As you build the statement with point-and-click operations, the Query Builder generates the SQL statement that must be executed against the database to retrieve the specified data. The statement that retrieves product names along with their categories is shown next:

```
SELECT  dbo.Products.ProductName, dbo.Categories.CategoryName
FROM    dbo.Categories INNER JOIN dbo.Products
           ON dbo.Categories.CategoryID = dbo.Products.CategoryID
```

If you paste this statement in the SQL pane and then execute it, you see a list of product names along with their categories. To execute the query, right-click somewhere in the Query Builder window and choose Execute SQL from the context menu. The Query Builder first fills out the remaining panes (if you chose to enter the SQL statement) and then executes the query. It displays the tables involved in the query in the Tables pane, inserts the appropriate rows in the Criteria pane, executes the query, and displays the results in the Results pane.

Results Pane

When you execute a statement, the Query Builder displays the results in the Results pane at the bottom of the window. The heading of each column is the column's name, unless you specified an alias for the column. In the following section, you'll build a few fairly complicated queries with the visual tools of Query Builder, and in the process I will discuss additional features of the Query Builder.

SQL at Work: Calculating Sums

Let's use the Query Builder to build a query that uses aggregates to retrieve all the products along with the quantities sold. The names of the products come from the Products table, whereas the quantities must be retrieved from the Order Details table. Because the same product appears in multiple rows of the Order Details table (each product appears in multiple invoices with different quantities), you must sum the quantities of all rows that refer to the same product.

Create a new view in the Server Explorer to start the Query Builder, right-click the upper pane, and choose Add Table. In the Add Table dialog box, select the tables Products and Order Details, and then close the dialog box. The two tables will appear in the Diagram pane with a relationship between them.

Now select the columns you want to include in the query: Select the *`ProductName`* column in the Products table and the *`Quantity`* column in the Order Details table. Expand the options in the Sort Type box in the *`ProductName`* row and select Ascending. The Query Builder generates the following SQL statement:

```
SELECT  dbo.Products.ProductName, dbo.[Order Details].Quantity
FROM    dbo.Products INNER JOIN dbo.[Order Details]
           ON dbo.Products.ProductID = dbo.[Order Details].ProductID
ORDER BY dbo.Products.ProductName
```

Execute this statement, and the first few lines in the Results pane are the following:

```
Alice Mutton   30
Alice Mutton   15
Alice Mutton   15
Alice Mutton   40
```

The Query Builder knows how the two tables are related (it picked up the relationship from the database) and retrieved the matching rows from the two tables. It also inserted a line that links the two tables in the Tables pane. Now you'll specify that you want the sum of the quantities. Right-click the *`Quantity`* column in the Criteria pane and choose the Add Group By option from the context menu. A new column is inserted after the Sort Order column. This column is set automatically to Group By for all the fields.

Now select the Group By cell of the *`Quantity`* row, expand the drop-down list, and select the Sum option. You have just specified that the *`Quantity`* column must be summed. The Group By option tells the Query Builder to group together all the rows that refer to the same product. This ensures that the sum includes all the products because the rows of the Order Details table that refer to the same product are grouped together.

Notice that the Alias cell of the Quantity row has become *`Expr1`* (it's no longer a column, but an aggregate). Set the alias to **`[Total Items]`**. (Make sure to include the square brackets, because the name contains a space.) Something has changed in the Diagram pane, too. The summation symbol has appeared next to the *`Quantity`* column (even though this column isn't selected to appear in the output of the query), and the grouping symbol (the nested brackets) has appeared next to the *`ProductName`* column, as shown in Figure 21.14.

Run the query now and see the results in the lower pane. Each product name appears only once, and the number next to it is the total number of items sold.

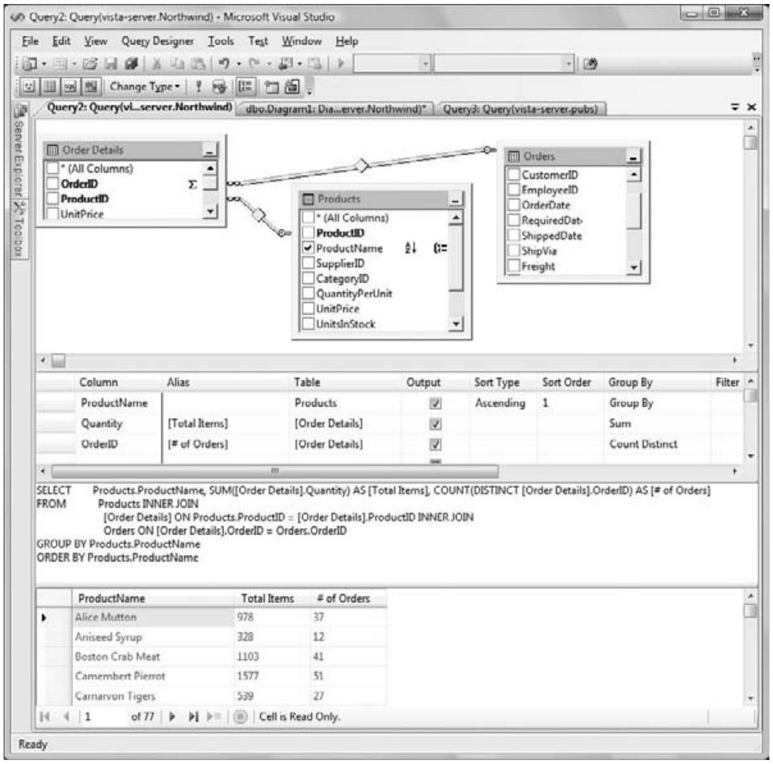
If you close the Query Builder window now, you'll be prompted about whether you want to save the new view and to specify a name for it. The definition will be saved to the Northwind database, along with the other objects of the database.

SQL at Work: Counting Rows

Let's say you want to find out the number of orders in which each product appears. Go back to the Server Explorer and open the previous view. Add the Orders table, which will be automatically related to the Order Details table via the *`OrderID`* field. Click the *`OrderID`* field in the Orders table. A new line will be added to the Criteria pane, and its Group By column will be set automatically to Group By. Set it to Count Distinct and its alias to **`[# Of Orders]`**. You'll sum the orders in which each product appears. The Count Distinct aggregate function is similar to the Count function, but it does not include the same order twice (if the same product appears in two rows of the same order). Run the query. This time you'll get one line per product. The Alice Mutton item has been ordered 37 times, and the total items sold are 978, as in the preceding query.

```
Alice Mutton       978     37
Aniseed Syrup      328     12
Boston Crab Meat   1103    41
Camembert Pierrot  1577    51
```

FIGURE 21.14
A query with totals



Column	Alias	Table	Output	Sort Type	Sort Order	Group By	Filter
ProductName		Products	✓	Ascending	1	Group By	
Quantity	[Total Items]	[Order Details]	✓			Sum	
OrderID	[# of Orders]	[Order Details]	✓			Count Distinct	

```
SELECT     Products.ProductName, SUM([Order Details].Quantity) AS [Total Items], COUNT(DISTINCT [Order Details].OrderID) AS [# of Orders]
FROM         Products INNER JOIN
               [Order Details] ON Products.ProductID = [Order Details].ProductID INNER JOIN
               Orders ON [Order Details].OrderID = Orders.OrderID
GROUP BY Products.ProductName
ORDER BY Products.ProductName
```

ProductName	Total Items	# of Orders
Alice Mutton	978	37
Aniseed Syrup	328	12
Boston Crab Meat	1103	41
Camembert Pierrot	1577	51
Carnarvon Tigers	539	27

1 of 77 Cell is Read Only.

The SELECT statement generated by the Query Builder is the following. Notice that the Orders table isn't involved in the query. All the information needed resides in the Order Details table. The Products table is included, so you can display product names instead of product IDs.

```
SELECT     Products.ProductName,
           SUM([Order Details].Quantity) AS [Total Items],
           COUNT(DISTINCT Orders.OrderID) AS [# Of Orders]
FROM         [Order Details]
INNER JOIN   Products ON [Order Details].ProductID = Products.ProductID
INNER JOIN   Orders ON [Order Details].OrderID = Orders.OrderID
GROUP BY Products.ProductName
ORDER BY Products.ProductName
```

Parameterized Queries

How about running the same query with different dates? Let's modify our query so that it prompts us for two dates and then calculates the totals in the corresponding period. Select the [Order Date] field from the Orders table and then switch to the following pane and enter this expression in the Filter cell for this row:

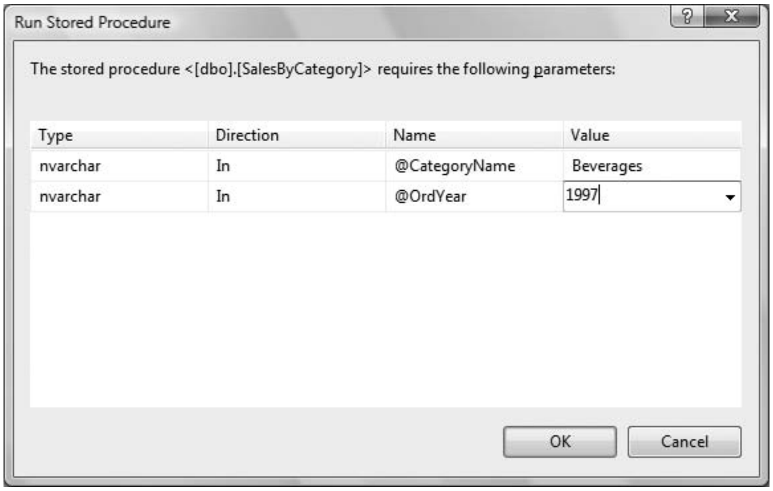
```
Between ? And ?
```

The Designer will replace the two question marks with two generic parameter names:

```
Between @param1 and @param2
```

You should change their names to something more meaningful, such as *@startDate* and *@endDate*. If you run the query, you'll be prompted to enter the values of the two parameters (Figure 21.15). A question mark in a query corresponds to a parameter, and you must supply the values for the parameters in the order in which they appear in the query. Every time you execute this query, the Define Query Parameters dialog will be displayed, where you must enter the values of the two parameters. When you close this dialog box, the query will be executed and you'll see its output in the Results pane.

FIGURE 21.15
Specifying the parameters for a query


Run Stored Procedure

The stored procedure <[dbo].[SalesByCategory]> requires the following parameters:

Type	Direction	Name	Value
nvarchar	In	@CategoryName	Beverages
nvarchar	In	@OrdYear	1997

Calculated Columns

Let's add yet another step of complexity to our query. We'll modify our query so that it calculates the total revenues generated by each product. Move down in the `Field` column of the Criteria pane, and enter the following expression in the first free cell:

```
Quantity * UnitPrice * (1 - Discount)
```

The wizard replaces the field names with fully qualified names:

```
([Order Details].Quantity * [Order Details].UnitPrice)
  * (1 - [Order Details].Discount)
```

This expression calculates the subtotal for each line in the Order Details table. You multiply the price by the quantity, taking into consideration the discount. Shortly, you'll sum the subtotals for each product.

Because this is a calculated column, its Alias becomes `Expr1`. Change this value to **Revenue**. In the Group By column of the row that corresponds to the order total, select Sum. Make sure that the Output column is selected and then run the query. You'll have the same results as before, only this time with an extra column, which is the revenue generated by the corresponding product:

```
Alice Mutton       978    37   32698.379981994629
Aniseed Syrup      328    12   3044
Boston Crab Meat   1103   41   17910.629892349243
```

The SQL statement generated by the Query Builder is as follows:

```
SELECT     Products.ProductName,
           SUM([Order Details].Quantity) AS [Total Items],
           COUNT(DISTINCT Orders.OrderID) AS [# Of Orders],
           SUM(([Order Details].Quantity * [Order Details].UnitPrice) *
             (1 - [Order Details].Discount)) AS Revenue
FROM         [Order Details]
INNER JOIN   Products ON [Order Details].ProductID = Products.ProductID
INNER JOIN   Orders ON [Order Details].OrderID = Orders.OrderID
WHERE     (Orders.OrderDate BETWEEN @Param1 AND @Param2)
GROUP BY Products.ProductName
ORDER BY Products.ProductName
```

This is a fairly complicated statement, and we won't get into any more complicated statements in this book. As you can see, you can create quite elaborate SQL statements to retrieve information from the database with point-and-click operations. But even if you don't want to enter your own SQL statements, some understanding of this language is required. All the keywords have been explained previously, and you can test your knowledge of SQL by examining the code generated by the Query Builder.

Stored Procedures

Stored procedures are short programs that are executed on the server and perform specific tasks. Any action you perform frequently against the database can be coded as a stored procedure, so that you can call it from within any application or from different parts of the same application. A stored procedure that retrieves customers by name is a typical example, and you'll call this stored procedure from many different places in your application.

You should use stored procedures for all the operations you want to perform against the database. Stored procedures isolate programmers from the database and minimize the risk of impairing the database's integrity. When all programmers access the same stored procedure to add a new invoice to the database, they don't have to know the structure of the tables involved or in what order to update these tables. They simply call the stored procedure, passing the invoice's fields as arguments. Another benefit of using stored procedures to update the database is that you don't risk implementing the same operation in two different ways. This is especially true for a team of developers because some developers might have not understood the business rules thoroughly. If the business rules change, you can modify the stored procedures accordingly, without touching the other parts of the application.

There's no penalty in using stored procedures versus SQL statements, and any SQL statement can be easily turned into a stored procedure, as you will see in this section. Stored procedures contain traditional programming statements that allow you to validate arguments, use default argument values, and so on. The language you use to write stored procedures is called T-SQL, which is a superset of SQL.

The SalesByCategory Stored Procedure

Let's explore stored procedures by looking at an existing one. Open the Server Explorer Toolbox, connect to the Northwind database, and then expand the Stored Procedures node. Locate the SalesByCategory stored procedure and double-click its name. The SalesByCategory stored procedure contains the statements from Listing 21.1, which appears in the editor's window.

LISTING 21.1: The SalesByCategory Stored Procedure

```
ALTER PROCEDURE dbo.SalesByCategory
    @CategoryName nvarchar(15),
    @OrdYear nvarchar(4) = '1998'
AS
IF @OrdYear != '1996' AND @OrdYear != '1997' AND @OrdYear != '1998'
BEGIN
   SELECT @OrdYear = '1998'
END
SELECT ProductName,
       TotalPurchase = ROUND(SUM(CONVERT(decimal(14,2),
       OD.Quantity * (1-OD.Discount) * OD.UnitPrice)), 0)
FROM [Order Details] OD, Orders O, Products P, Categories C
WHERE OD.OrderID = O.OrderID
      AND OD.ProductID = P.ProductID
      AND P.CategoryID = C.CategoryID
      AND C.CategoryName = @CategoryName
      AND SUBSTRING(CONVERT(nvarchar(22), O.OrderDate, 111), 1, 4) = @OrdYear
GROUP BY ProductName
ORDER BY ProductName
```

This type of code is probably new to you. You'll learn it quite well as you go along because it's really required in coding database applications. You can rely on the various wizards to create stored procedures for you, but you should be able to understand how they work. While you're editing a stored procedure, the sections of the stored procedure that are pure SQL are enclosed in a rectangle.

The first statement alters the procedure SalesByCategory, which is already stored in the database. If it's a new procedure, you can use the `CREATE` statement, instead of `ALTER`, to attach a new stored procedure to the database. The following lines until the `AS` keyword are the parameters of the stored procedure. All variables in T-SQL start with the `@` symbol. `@CategoryName` is a 15-character string, and `@OrdYear` is a string that also has a default value. If you omit the second argument when calling the SalesByCategory procedure, the year 1998 will be used automatically.

The `AS` keyword marks the beginning of the stored procedure. The first `IF` statement makes sure that the year is a valid one (from 1996 to 1998). If not, it will use the year 1998. The `BEGIN` and `END` keywords mark the beginning and end of the `IF` block (the same block that's delimited by the `If` and `End If` statements in VB code).

Following the IF statement is a long SELECT statement that uses the arguments passed to the stored procedure as parameters. This is a straight SQL statement that implements a parameterized query.

The second half of the stored procedure's code appears in a box in the editor's window. Right-click anywhere in this box and choose Design SQL Block. This block is a SQL statement that retrieves the total sales for the specified year and groups them by category. You can edit it either as a SQL segment or through the visual interface of the Query Builder. You already know how to handle SQL statements, so everything you learned about building SQL statements applies to stored procedures as well. The only difference is that you can embed traditional control structures — such as IF statements and WHILE loops — and mix them with SQL.

Right-click anywhere in the editor and choose Execute. A dialog box pops up and prompts you to enter the values for the two parameters the query expects: the name of the category and the year. Type **Beverages** and **1997** in the dialog box and then click OK. The stored procedure returns the qualifying rows, which display in the Output window.

The SalesByCategory stored procedure returned the following lines when executed with these parameters:

```
ProductName                              TotalPurchase
---------------------------------------- -------------
Chai                                     4887
Chang                                    7039
Chartreuse verte                         4476
Côte de Blaye                            49198
Guaran’ Fant’stica                       1630
No more results.
(12 row(s) returned)
@RETURN_VALUE = 0
Finished running dbo."SalesByCategory".
```

The Bottom Line

Use relational databases. Relational databases store their data in tables and are based on relationships between these tables. The data is stored in tables, and tables contain related data, or entities, such as persons, products, orders, and so on. Relationships are implemented by inserting columns with matching values in the two related tables.

Master It How will you relate two tables with a many-to-many relationship?

Utilize the data tools of Visual Studio. Visual Studio 2008 provides visual tools for working with databases. The Server Explorer is a visual representation of the databases you can access from your computer and their data. You can create new databases, edit existing ones, and manipulate their data. You can also create queries and test them right in the IDE.

Master It Describe the process of establishing a new relationship between two tables.

Use the Structured Query Language for accessing tables. Structured Query Language (SQL) is a universal language for manipulating tables. SQL is a nonprocedural language, which specifies the operation you want to perform against a database at a high level, unlike traditional languages such as Visual Basic, which specifies how to perform the operation. The details of the implementation are left to the DBMS. SQL consists of a small number of keywords and is optimized for selecting, inserting, updating, and deleting data.

Master It How would you write a SELECT statement to retrieve data from multiple tables?

Chapter 22

Programming with ADO.NET

In Chapter 21, ''Basic Concepts of Relational Databases,'' you learned how to access data stored in databases by using a universal data-manipulation language, SQL, and the Transact-SQL (T-SQL) extensions of SQL Server 2008. However, you can't maintain a real database by executing SQL statements from within SQL Server's Management Studio. You need special applications that access the database, display relevant data on a Windows or web form, and submit the changes made to the data by the user back to the database. These applications are known as *front-end applications,* because they interact with the user and update the data on a database server, or a back-end data store. They're also known as *data-driven applications,* because they interact not only with the user, but primarily with the database.

In this chapter, you'll explore the basic mechanisms of ADO.NET to interact with the sample databases. As you will see, it's fairly straightforward to write a few VB statements to execute SQL queries against the database in order to either edit or retrieve selected rows. The real challenge is the design and implementation of functional interfaces that display the data requested by the user (the data you'll retrieve via `SELECT` statements from the database), allow the user to navigate through the data and edit it, and finally submit the changes to the database. You'll learn how to execute `SELECT` statements against the database, retrieve data, and submit modified or new data to the database.

In this chapter, you'll learn how to do the following:

- ◆ Create and populate DataSets
- ◆ Establish relations between tables in the DataSet
- ◆ Submit changes in the DataSet back to the database

Stream- versus Set-Based Data Access

ADO.NET provides two basic methods of accessing data: *stream-based data access,* which establishes a stream to the database and retrieves the data from the server, and *set-based data access,* which creates a special data structure at the client and fills it with data.

This structure is the DataSet, which resembles a section of the database: It contains one or more DataTable objects, which correspond to tables and are made up of DataRow objects. These DataRow objects have the same structure as the rows in their corresponding tables. DataSets are populated by retrieving data from one or more database tables into the corresponding DataTables. As for submitting the data to the database with the stream-based approach, you must create the appropriate `INSERT`/`UPDATE`/`DELETE` statements and then execute them against the database.

The stream-based approach relies on the DataReader object, which makes the data returned by the database available to your application. The client application reads the data returned by a query through the DataReader object and must store it somehow at the client. Quite frequently, we use business objects to store the data at the client.

The set-based approach uses the same objects as the stream-based approach behind the scenes, and it abstracts most of the grunt work required to set up a link to the database, retrieve the data, and store it in the client computer's memory. So, it makes sense to start by exploring the stream-based approach and the basic objects provided by ADO.NET for accessing databases. After you understand the nature of ADO.NET and how to use it, you'll find it easy to see the abstraction introduced by the set-based approach and how to make the most of DataSets. As you will see in the following chapter, you can create DataSets and the supporting objects with the visual tools of the IDE.

The Basic Data-Access Classes

A data-driven application should be able to connect to a database and execute queries against it. The selected data is displayed on the appropriate interface, where the user can examine it or edit it. Finally, the edited data is submitted to the database. This is the cycle of a data-driven application:

1. Retrieve data from the database.
2. Present data to the user.
3. Allow the user to edit the data.
4. Submit changes to the database.

Of course, there are many issues that are not obvious from this outline. Designing the appropriate interface for navigating through the data (going from customers to their orders and from the selected order to its details) can be quite a task. Developing a functional interface for editing the data at the client is also a challenge, especially if several related tables are involved. We must also take into consideration that there are other users accessing the same database. What will happen if the product we're editing has been removed in the meantime by another user? Or what if a user has edited the same customer's data since our application read it? Do we overwrite the changes made by the other user, or do we reject the edits of the user who submits the edits last? I'll address these issues in this and the following chapter, but we need to start with the basics: the classes for accessing the database.

To connect to a database, you must create a Connection object, initialize it, and then call its `Open` method to establish a connection to the database. The Connection object is the channel between your application and the database; every command you want to execute against the same database must use this Connection object. When you're finished, you must close the connection by calling the Connection object's `Close` method. Because ADO.NET maintains a pool of Connection objects that are reused as needed, it's imperative that you keep connections open for the shortest possible time.

The object that will actually execute the command against the database is the Command object, which you must configure with the statement you want to execute and associate with a Connection object. To execute the statement, you can call one of the Command object's methods. The `ExecuteReader` method returns a DataReader object that allows you to read the data returned by the selection query. To execute a statement that updates a database table but doesn't return a set of rows, use the `ExecuteNonQuery` method, which executes the specified command and returns an integer, which is the number of rows affected by the statement. The following sections describe the Connection and Command classes in detail.

To summarize, ADO.NET provides three core classes for accessing databases: the Connection, Command, and DataReader classes. There are more data access–related classes, but they're all based on these three basic classes. After you understand how to interact with a database by using these classes, you'll find it easy to understand the additional classes, as well as the code generated by the visual data tools that come with Visual Studio.

The Connection Class

The Connection class is an abstract one, and you can't use it directly. Instead, you must use one of the classes that derive from the Connection class. Currently, there are three derived classes: SqlConnection, OracleConnection, and OleDbConnection. Likewise, the Command class is also an abstract class with three derived classes: SqlCommand, OracleCommand, and OleDbCommand.

The SqlConnection and SqlCommand classes belong to the SqlClient namespace, which you must import into your project via the following statement:

```
Imports System.Data.SqlClient
```

The examples of this book use the SQL Server 2008 DBMS, and it's implied that the SqlClient namespace is imported into every project that uses SQL Server.

To connect the application to a database, the Connection object must know the name of the server on which the database resides, the name of the database itself, and the credentials that will allow it to establish a connection to the database. These credentials are either a username and password or a Windows account that has been granted rights to the database. You obviously know what type of DBMS you're going to connect to, so you can select the appropriate Connection class. The most common method of initializing a Connection object in your code is the following:

```
Dim CN As New SqlConnection("Data Source = localhost;
        Initial Catalog = Northwind; uid = user_name;
        password = user_password")
```

`localhost` is a universal name for the local machine, `Northwind` is the name of the database, and `user_name` and `user_password` are the username and password of an account configured by the database administrator. The Northwind sample database isn't installed along with SQL Server 2008, but you can download it from MSDN and install it yourself. The process was described in the section ''Obtaining the Northwind and Pubs Sample Databases'' in Chapter 21. I'm assuming that you're using the same computer both for SQL Server and to write your VB applications. If SQL Server resides on a different computer in the network, use the server computer's name (or IP address) in place of the `localhost` name. If SQL Server is running on another machine on the network, use a setting like the following for the `Data Source` key:

```
Data Source = \\PowerServer
```

If the database is running on a remote machine, use the remote machine's IP address. If you're working from home, for example, you can establish a connection to your company's server with a connection string like the following:

```
Data Source = 213.16.178.100; Initial Catalog = BooksDB; uid = xxx; password = xxx
```

The *uid* and *password* keys are those of an account created by the database administrator, and not a Windows account. If you want to connect to the database by using each user's Windows credentials, you should omit the *uid* and *password* keys and use the *Integrated Security* key instead. If your network is based on a domain controller, you should use integrated security so that users can log in to SQL Server with their Windows account. This way you won't have to store any passwords in your code, or even an auxiliary file with the application settings.

If you're using an IP address to specify the database server, you may also have to include SQL Server's port by specifying an address such as `213.16.178.100, 1433`. The default port for SQL Server is 1433, and you can omit it. If the administrator has changed the default port, or has hidden the server's IP address behind another IP address for security purposes, you should contact the administrator to get the server's address. If you're connecting over a local network, you shouldn't have to use an IP address. If you want to connect to the company server remotely, you will probably have to request the server's IP address and the proper credentials from the server's administrator.

The basic property of the Connection object is the `ConnectionString` property, which is a semicolon-separated string of key-value pairs and specifies the information needed to establish a connection to the desired database. It's basically the same information you provide in various dialog boxes when you open the SQL Server Management Studio and select a database to work with. An alternate method of setting up a Connection object is to set its `ConnectionString` property:

```
Dim CN As New SqlConnection
CN.ConnectionString = _
        "Data Source = localhost; Initial Catalog = Northwind; " & _
        "Integrated Security = True"
```

One of the Connection class's properties is the `State` property, which returns the state of a connection; its value is a member of the `ConnectionState` enumeration: Connecting, Open, Executing, Fetching, Broken and Closed. If you call the `Close` method on a Connection object that's already closed, or the `Open` method on a Connection that's already open, an exception will be thrown. To avoid the exception, you must examine the Connection's `State` property and act accordingly.

The following code segment outlines the process of opening a connection to a database:

```
Dim CNstring As String = _
         "Data Source=localhost;Initial " & _
         "Catalog=Northwind;Integrated Security=True"
CNstring = InputBox( _
        "Please enter a Connection String",_
        "CONNECTION STRING", CNstring)
If CNstring.Trim = "" Then Exit Sub
Dim CN As New SqlConnection(CNstring)
Try
    CN.Open()
    If CN.State = ConnectionState.Open Then
        MsgBox("Workstation " & CN.WorkstationId & _
               " connected to database " & CN.Database & _
               " on the " & CN.DataSource & " server")
    End If
```

```
Catch ex As Exception
    MsgBox( _
      "FAILED TO OPEN CONNECTION TO DATABASE DUE TO THE FOLLOWING ERROR" & _
            vbCrLf & ex.Message)
End Try
' use the Connection object to execute statements
' against the database and then close the connection
If CN.State = ConnectionState.Open Then CN.Close()
```

The Command Class

The second major component of the ADO.NET model is the Command class, which allows you to execute SQL statements against the database. The two basic parameters of the Command object are a Connection object that specifies the database where the command will be executed, and the actual SQL command. To execute a SQL statement against a database, you must initialize a Command object and set its `Connection` property to the appropriate Connection object. It's the Connection object that knows how to connect to the database; the Command object simply submits a SQL statement to the database and retrieves the results.

The Command object exposes a number of methods for executing SQL statements against the database, depending on the type of statement we want to execute. The `ExecuteNonQuery` method executes `INSERT/DELETE/UPDATE` statements that do not return any rows, just an integer value, which is the number of rows affected by the query. The `ExecuteScalar` method returns a single value, which is usually the result of an aggregate operation, such as the count of rows meeting some criteria, the sum or average of a column over a number of rows, and so on. Finally, the `ExecuteReader` method is used with `SELECT` statements that return rows from one or more tables.

To execute an `UPDATE` statement, for example, you must create a new Command object and associate the appropriate SQL statement with it. One overloaded form of the constructor of the Command object allows you to specify the statement to be executed against the database, as well as a Connection object that points to the desired database as arguments:

```
Dim CMD As New SqlCommand( _
           "UPDATE Products SET UnitPrice = UnitPrice * 1.07 " & _
         "WHERE CategoryID = 3", CN)
CN.Open
Dim rows As Integer
rows = CMD.ExecuteNonQuery
If rows = 1 Then
    MsgBox("Table Products updated successfully")
Else
    MsgBox("Failed to update the Products table")
End If
If CN.State = ConnectionState.Open Then CN.Close
```

The `ExecuteNonQuery` method returns the number of rows affected by the query, and it's the same value that appears in the Output window of SQL Server's Management Studio when you execute an action query. The preceding statements mark up the price of all products in the Confections category by 7 percent. You can use the same structure to execute `INSERT` and `DELETE` statements; all you have to change is the actual SQL statement in the SqlCommand object's constructor. You can also set up a Command object by setting its Connection and CommandText properties:

```
Command.Connection = Connection
Command.CommandText = "SELECT COUNT(*) FROM Customers"
```

After you're finished with the Command object, you should close the Connection object. Although you can initialize a Connection object anywhere in your code, you should call its Open method as late as possible (that is, just before executing a statement) and its Close method as early as possible (that is, as soon as you have retrieved the results of the statement you executed).

The ExecuteScalar method executes the SQL statement associated with the Command object and returns a single value, which is the first value that the SQL statement would print in the Output window of SQL Server's Management Studio. The following statements read the number of rows in the Customers table of the Northwind database and store the result in the *count* variable:

```
Dim CMD As New SqlCommand( _
               "SELECT COUNT(*) FROM Customers", CN)
Dim count As Integer
CN.Open
count = CMD.ExecuteScalar
If CN.State = ConnectionState.Open Then CN.Close
```

If you want to execute a SELECT statement that retrieves multiple rows, you must use the ExecuteReader method of the Command object, as shown here:

```
Dim CMD As New SqlCommand( _
               "SELECT * FROM Customers", CN)
CN.Open
Dim Reader As SqlDataReader
Reader = CMD.ExecuteReader
While Reader.Read
    ' process the current row in the result set
End While
If CN.State = ConnectionState.Open Then CN.Close
```

You'll see shortly how to access the fields of each row returned by the ExecuteReader method through the properties of the SqlDataReader class.

Executing Stored Procedures

The command to be executed through the Command object is not always a SQL statement; it could be the name of a stored procedure, or the name of a table, in which case it retrieves all the rows of the table. You can specify the type of statement you want to execute with the CommandType property, whose value is a member of the CommandType enumeration: *Text* (for SQL statements), *StoredProcedure* (for stored procedures), and *TableDirect* (for a table). You don't have to specify the type of the command you want to execute, but then the Command object will have to figure it out, a process that will take a few moments, and you can avoid this unnecessary delay. The Northwind database comes with the Ten Most Expensive Products stored procedure. To execute this stored procedure, set up a Command object with the following statements:

```
Dim CMD As New SqlCommand
CMD.Connection = CN
CMD.CommandText = "[Ten Most Expensive Products]"
CMD.CommandType = CommandType.StoredProcedure
```

Finally, you can retrieve all the rows of the Customers table by setting up a Command object like the following:

```
Dim CMD As New SqlCommand
CMD.Connection = CN
CMD.CommandText = "Customers"
CMD.CommandType = CommandType.TableDirect
```

Executing Selection Queries

The most common SQL statements, the `SELECT` statements, retrieve a set of rows from one or more joined tables, the *result set*. These statements are executed with the `ExecuteReader` method, which returns a DataReader object — a SqlDataReader object for statements executed against SQL Server databases. The DataReader class provides the members for reading the results of the query in a forward-only manner. The connection remains open while you read the rows returned by the query, so it's imperative to read the rows and store them in a structure in the client computer's memory as soon as possible, and then close the connection. The DataReader object is read-only (you can't use it to update the underlying rows), so there's no reason to keep it open for long periods. Let's execute the following `SELECT` statement to retrieve selected columns of the rows of the Employees table of the Northwind database:

```
SELECT LastName + ' ' + FirstName AS Name,
       Title, Extension, HomePhone
FROM   Employees
```

Here are the VB statements that set up the appropriate Command object and retrieve the SqlDataReader object with the result set:

```
Dim CMD As New SqlCommand
CMD.Connection = CN
CMD.CommandText = _
     "SELECT LastName + ' ' + FirstName AS Name, " & _
     "Title, Extension, HomePhone FROM Employees"
Connection.Open()
Dim Reader As SqlDataReader
Reader = Command.ExecuteReader
While Reader.Read
    str &= Convert.ToString(Reader.Item("Name")) & vbTab
    str &= Convert.ToString(Reader.Item("Title")) & vbTab
    str &= Convert.ToString(Reader.Item("Extension")) & vbTab
    str &= Convert.ToString(Reader.Item("HomePhone")) & vbTab
```

```
        str &= vbCrLf
        count += 1
    End While
    Debug.WriteLine(vbCrLf & vbCrLf & "Read " & count.ToString & _
                    " rows:" & vbCrLf & vbCrLf & str)
    Connection.Close()
```

The DataReader class provides the `Read` method, which advances the current pointer to the next row in the result set. To read the individual columns of the current row, you use the `Item` property, which allows you to specify the column by name and returns an object variable. It's your responsibility to cast the object returned by the `Item` property to the appropriate type. Initially, the DataReader is positioned in front of the first line in the result set, and you must call its `Read` method to advance to the first row. If the query returns no rows, the `Read` method will return False and the `While` loop won't be executed at all. In the preceding sample code, the fields of each row are concatenated to form the *`str`* string, which is printed in the Immediate window; it looks something like this:

```
Davolio Nancy     Sales Representative    5467  (206) 555-9857
Fuller Andrew     Vice President, Sales   3457  (206) 555-9482
Leverling Janet   Sales Representative    3355  (206) 555-3412
```

Using Commands with Parameters

Most SQL statements and stored procedures accept parameters, and you should pass values for each parameter before executing the query. Consider a simple statement that retrieves the customers from a specific country, whose name is passed as an argument:

```
SELECT * FROM Customers WHERE Country = @country
```

The *`@country`* parameter must be set to a value, or an exception will be thrown as you attempt to execute this statement. Stored procedures also accept parameters. The `Sales By Year` stored procedure of the Northwind database, for example, expects two Date values and returns sales between the two dates. To accommodate the passing of parameters to a parameterized query or stored procedure, the Command object exposes the `Parameters` property, which is a collection of Parameter objects. To pass parameter values to a command, you must set up a Parameter object for each parameter; set its name, type, and value; and then add the Parameter object to the `Parameters` collection of the Command object. The following statements set up a Command object with a parameter of the `varchar` type with a maximum size of 15 characters:

```
Dim Command As New SqlCommand
Command.CommandText = "SELECT * FROM Customers WHERE Country = @country"
Command.Parameters.Add("@country", SqlDbType.VarChar, 15)
Command.Parameters("@country").Value = "Italy"
```

At this point, you're ready to execute the `SELECT` statement with the `ExecuteReader` method and retrieve the customers from Italy. You can also configure the Parameter object in its constructor:

```
Dim param As New SqlParameter(paramName, paramType, paramSize)
```

Here's the constructor of the *@country* parameter of the preceding example:

```
Dim param As New SqlParameter("@country", SqlDbType.VarChar, 15)
param.Value = "Italy"
Command.Parameters.Add param
```

Finally, you can combine all these statements into a single one:

```
Command.Parameters.Add("@country", SqlDbType.VarChar, 15).Value = "Italy"
```

In the last statement, I initialize the parameter as I add it to the `Parameters` collection and then set its value to the string `Italy`. Oddly, there's no overloaded form of the `Add` method that allows you to specify the parameter's value, but there is an `AddWithValue` method, which adds a new parameter and sets its value. This method accepts two arguments: a string with the parameter's name, and an object with the parameter's value. The actual type of the value is determined by the type of the query or stored procedure's argument, and it's resolved at runtime. The simplest method of adding a new parameter to the `Command.Parameters` collection is the following:

```
Command.Parameters.Add("@country", "Italy")
```

After the parameter has been set up, you can call the `ExecuteReader` method to retrieve the customers from the country specified by the argument and then read the results through an instance of the DataReader class.

Retrieving Multiple Values from a Stored Procedure

Another property of the Parameter class is the `Direction` property, which determines whether the stored procedure can alter the value of the parameter. The `Direction` property's setting is a member of the `ParameterDirection` enumeration: *Input*, *Output*, *InputOutput*, and *ReturnValue*. A parameter that's set by the procedure should have its `Direction` property set to Output: the parameter's value is not going to be used by the procedure, but the procedure's code can set it to return information to the calling application. If the parameter is used to pass information to the procedure, as well as to pass information back to the calling application, its `Direction` property should be set to InputOutput.

Let's look at a stored procedure that returns the total of all orders, as well as the total number of items ordered by a specific customer. This stored procedure accepts as a parameter the ID of a customer, obviously, and it returns two values: the total of all orders placed by the specified customer and the number of items ordered. A procedure (be it a SQL Server stored procedure or a regular VB function) can't return two or more values. The only way to retrieve multiple results from a single stored procedure is to pass output parameters, so that the stored procedure can set their value. To make the stored procedure a little more interesting, we'll add a return value, which will be the number of orders placed by the customer. Listing 22.1 shows the implementation of the `CustomerTotals` stored procedure.

Listing 22.1: The *CustomerTotals* Stored Procedure

```
CREATE PROCEDURE CustomerTotals
@customerID varchar(5),
@customerTotal money OUTPUT,
@customerItems int OUTPUT
AS
SELECT @customerTotal = SUM(UnitPrice * Quantity * (1 - Discount))
FROM [Order Details] INNER JOIN Orders
ON [Order Details].OrderID = Orders.OrderID
WHERE Orders.CustomerID = @customerID
SELECT @customerItems = SUM(Quantity)
FROM [Order Details] INNER JOIN Orders
ON [Order Details].OrderID = Orders.OrderID
WHERE Orders.CustomerID = @customerID

DECLARE @customerOrders int
SELECT @customerOrders = COUNT(*) FROM Orders
WHERE Orders.CustomerID = @customerID

RETURN @customerOrders
```

To attach the `CustomerTotals` stored procedure to the database, create a new stored procedure, paste the preceding statements in the code window, and press F5 to execute it. The stored procedure calculates three totals for the specified customer and stores them to three local variables. The *`@customerTotal`* and *`@customerItems`* variables are output parameters, which the calling application can read after executing the stored procedure. The *`@customerOrders`* variable is the procedure's return value. We can return the number of orders for the customer through the stored procedure's return value, because this variable happens to be an integer, and the return value is always an integer. In more-complex stored procedures, we'd use output parameters for all the values we want to return to the calling application, and the procedure would return a value to indicate the execution status: 0 or 1 if the procedure completed its execution successfully, and a negative value to indicate the error, should the procedure fail to execute.

Before using the `CustomerTotals` stored procedure with our VB application, let's test it in the SQL Server Management Studio. We must declare a variable for each of the output parameters: the *`@Total`*, *`@Items`*, and *`@Orders`* variables. These three variables must be passed to the stored procedure with the `OUTPUT` attribute, as shown here:

```
DECLARE @Total money
DECLARE @Items int
DECLARE @Orders int
DECLARE @custID varchar(5)
SET @custID = 'BLAUS'
EXEC @orders = CustomerTotals @custID,
          @customerTotal OUTPUT, @customerItems OUTPUT
PRINT 'Customer ' + @custId + ' has placed a total of ' +
      CAST(@orders AS varchar(8)) + ' orders ' +
      ' totaling $' + CAST(ROUND(@customerTotal, 2) AS varchar(12)) +
      ' and ' + CAST(@customerItems AS varchar(4)) + ' items.'
```

Open a new query window in the Management Studio and enter the preceding statements. Press F5 to execute them and you will see the following message printed in the output window:

```
Customer BLAUS has placed a total of 8 orders totaling $10355.45 and 653 items.
```

The customer's ID is an `INPUT` parameter, and we could pass it to the procedure as a literal. You can omit the declaration of the *`@custID`* variable and call the stored procedure with the following statement:

```
EXEC @orders = _
          CustomerTotals 'BLAUS', @customerTotal OUTPUT, @customerItems OUTPUT
```

Now that we've tested our stored procedure and know how to call it, we'll do the same from within our sample application. To execute the `CustomerTotals` stored procedure, we must set up a Command object, create the appropriate Parameter objects (one Parameter object per stored procedure parameter plus another Parameter object for the stored procedure's return value), and then call the `Command.ExecuteNonQuery` method. Upon return, we'll read the values of the output parameters and the stored procedure's return value. Listing 22.2 shows the code that executes the stored procedure (see the SimpleQueries sample project).

LISTING 22.2: Executing a Stored Procedure with Output Parameters

```
Private Sub bttnExecSP_Click(...) Handles bttnExecSP.Click
    Dim customerID As String = InputBox("Please a customer's ID", _
          "CustomerTotals Stored Procedure", "ALFKI")
    If customerID.Trim.Length = 0 Then Exit Sub
    Dim CMD As New SqlCommand
    CMD.Connection = CN
    CMD.CommandText = "CustomerTotals"
    CMD.CommandType = CommandType.StoredProcedure
    CMD.Parameters.Add( _
          "@customerID", SqlDbType.VarChar, 5).Value = customerID
    CMD.Parameters.Add("@customerTotal", SqlDbType.Money)
    CMD.Parameters("@customerTotal").Direction = ParameterDirection.Output
    CMD.Parameters.Add("@customerItems", SqlDbType.Int)
    CMD.Parameters("@customerItems").Direction = ParameterDirection.Output
    CMD.Parameters.Add("@orders", SqlDbType.Int)
    CMD.Parameters("@orders").Direction = ParameterDirection.ReturnValue
    CN.Open()
    CMD.ExecuteNonQuery()
    CN.Close()
    Dim items As Integer
    Items = Convert.ToInt32(CMD.Parameters("@customerItems").Value
    Dim orders As Integer
```

```
        Orders = Convert.ToInt32(CMD.Parameters(          "@orders").Value
        Dim ordersTotal As Decimal
        ordersTotal = Convert.ToDouble( _
             CMD.Parameters("@customerTotal").Value
        MsgBox("Customer BLAUS has placed " & _
               orders.ToString & " orders " & _
               "totaling $" & Math.Round( _
               orderTotal, 2).ToString("#,###.00") & _
               " and " & items.ToString & " items")
    End Sub
```

In most applications, the same Command object will be reused again and again with different parameter values, so it's common to add the parameters to a Command object's `Parameters` collection and assign values to them every time we want to execute the command. Let's say you've designed a form with text boxes, where users can edit the values of the various fields; here's how you'd set the values of the *UPDATECMD* object's parameters:

```
UPDATECMD.Parameters("@CustomerID").Value = txtID.Text.Trim
UPDATECMD.Parameters("@CompanyName").Value = txtCompany.Text.Trim
```

After setting the values of all parameters, you can call the `ExecuteNonQuery` method to submit the changes to the database. To update another customer, just assign different values to the existing parameters and call the *UPDATECMD* object's `ExecuteNonQuery` method.

Real World Scenario

Security Issue: SQL Injection Attacks

You may be wondering why we have to go through the process of creating SQL statements with parameters and setting up Parameter objects, instead of generating straightforward SQL statements on-the-fly. A statement that picks some user-supplied values and embeds them in a SQL statement can be exploited by a malicious user as a Trojan horse to execute any SQL statement against your database. Can you guess what will happen when the user enters a SQL statement in one of the TextBox controls on the form and your code uses this string to build a SQL statement on-the-fly? The code will pick it up, insert it into the larger statement, and then execute it against the database. This technique is known as *SQL injection,* and I'll show you how it works with a simple example. It's a known issue with any DBMS that can execute multiple SQL statements in a batch mode, and SQL Server is certainly vulnerable to SQL injection attacks.

How would you validate the users of an application? You'd most likely store the usernames and passwords in a table and execute a few statements to locate the row with the specified ID and password, right? If you didn't know any better, you'd probably write some code to extract the values from two TextBox controls and build the following SQL statement:

```
Dim CMD As New SqlClient.SqlCommand( _
"SELECT COUNT(*) FROM Customers WHERE CompanyName = "' & _
txtName.Text & "' AND CustomerID = "' & txtPsswd.Text & ""')
```

Then you'd execute this statement and examine the number of rows that match the specified criteria:

```
Dim count As Integer
count = Convert.ToInt32(CMD.ExecuteScalar)
```

If the value of the `count` variable is 1, the user can log in. (I've assumed that the `CompanyName` field is the username and that the password for each user is the `CustomerID` field of the Customers table.) If the values in the two text boxes are **`Frankenversan`** and **`FRANK`**, the following statement will be executed against the database and will return the numeric value 1:

```
SELECT COUNT(*) FROM Customers
WHERE CompanyName = 'Frankenversan' AND CustomerID = 'FRANK'
```

A malicious user might enter the following username (and no password at all):

```
xxx' ; DROP TABLE [Orders] --
```

If you examine the value of the `Command.CommandText` property before this statement is executed, you'll see the following SQL statement:

```
SELECT COUNT(*) FROM Customers
WHERE CompanyName = 'xxx' ;
DROP TABLE [Orders] --' AND CustomerID = ''
```

This statement contains two SQL statements and a comment: the `SELECT` statement, followed by another statement that drops the Orders table! The comments symbol (the two dashes) is required to disable the last part of the original statement, which otherwise would cause a syntax error. The Orders table can't be dropped, because it contains related rows in the Order Details table, but nothing will stop you from dropping the Order Details or the Employees table.

If you want to demonstrate to someone that their software isn't secure, you can replace the `DROP TABLE` statement with the `SHUTDOWN` statement. The following statement shuts down the server immediately:

```
SHUTDOWN WITH NOWAIT
```

Handling Special Characters

Another problem you will avoid with parameterized queries and stored procedures is that of handling single quotes, which are used to delimit literals in T-SQL. Consider the following `UPDATE` statement, which picks up the company name from a TextBox control and updates a single row in the Customers table:

```
CMD.CommandText = _
       "UPDATE Customers SET CompanyName = "' & _
       txtCompany.Text & ""' & _
       "WHERE CustomerID = "' & txtID.Text & ""'
```

If the user enters a company name that contains a single quote, such as *B's Beverages*, the command will become the following:

```
UPDATE Customers SET CompanyName = 'B's Beverages' WHERE CustomerID = 'BSBEV'
```

If you attempt to execute this statement, SQL Server will reject it because it contains a syntax error (you should be able to figure out the error easily by now). The exact error message is as follows:

```
Msg 102, Level 15, State 1, Line 1
Incorrect syntax near 's'.
Msg 105, Level 15, State 1, Line 1
Unclosed quotation mark after the character string ".
```

The single quote is used to delimit literals, and there should be an even number of single quotes in the statement. The compiler determines that there's an unclosed quotation mark in the statement and doesn't execute it. If the same statement was written as a parameterized query, such as the following, you could pass the same company name to the statement:

```
CMD.CommandText = _
    "UPDATE Customers SET CompanyName = @CompanyName " & _
    "WHERE CustomerID = @ID"
CMD.Parameters.Add("@CompanyName", _
         SqlDbType.VarChar, 40).Value = "B's Beverages"
CMD.Parameters.Add("@ID", _
         SqlDbType.Char, 5).Value = "BSBEV"
CMD.ExecuteNonQuery
```

The same is true for other special characters, such as the percentage symbol. It's possible to escape the special symbols; you can replace the single-quote mark with two consecutive single quotes, but the most elegant method of handling special characters, such as quotation marks, percent signs, and so on, is to use parameterized queries or stored procedures. You just assign a string to the parameter and don't have to worry about escaping any characters; the Command object will take care of all necessary substitutions.

Empty Strings versus Null Values

The values you assign to the arguments of a query or stored procedure usually come from controls on a Windows form, and in most cases from TextBox controls. The following statement reads the text in the `txtFax` TextBox control and assigns it to the `@FAX` parameter of a Command object:

```
Command.Parameters("@FAX").Value = txtFax.Text
```

But what if the user has left the *txtFax* TextBox blank? Should we pass to the INSERT statement an empty string or a Null value? If you collect the values from various controls on a form and use them as parameter values, you'll never send Null values to the database. If you want to treat empty strings as Null values, you must pass a Null value to the appropriate parameter explicitly. Let's say that the *txtFax* TextBox control on the form corresponds to the *@FAX* parameter. You can use the IIf() statement of Visual Basic to assign the proper value to the corresponding parameter as follows:

```
UPDATECommand.Parameters("@FAX").Value = _
     IIf(txtFax.Text.Trim.Length = 0, _
             System.DBNull.Value, txtFax.Text)
```

This is a lengthy statement, but here's how it works. The IIf() function evaluates the specified expression. If the length of the text in the *txtFax* control is zero, it returns the value specified by its second argument, which is the Null value. If not — in other words, if the TextBox control isn't empty — it returns the text on the control. This value is then assigned to the *@FAX* parameter of the *UPDATECommand* object.

The DataReader Class

To read the rows returned by a selection query, you must call the Command object's ExecuteReader method, which returns a DataReader object (a SqlDataReader object for queries executed against SQL Server). The DataReader is a stream to the data retrieved by the query, and it provides many methods for reading the data sent to the client by the database. The underlying Connection object remains open while you read the data off the DataReader, so you must read it as quickly as possible, store it at the client, and close the connection as soon as possible.

To read a set of rows with the DataReader, you must call its Read method, which advances the pointer to the next row in the set. Initially, the pointer is in front of the first row, so you must call the Read method before accessing the first row. Despite its name, the Read method doesn't actually fetch any data; to read individual fields, you must use the various Get methods of the DataReader object, described next (GetDecimal, GetString, and so on). After reading the fields of the current row, call the Read method again to advance to the next row. There's no method to move to a previous row, so make sure you've read all the data of the current row before moving to the next one. The basic properties and methods of the DataReader object are explained here:

HasRows This is a Boolean property that specifies whether there's a result set to read data from. If the query selected no rows at all, the HasRows property will return False.

FieldCount This property returns the number of columns in the current result set. Note that the DataReader object doesn't know the number of rows returned by the query. Because it reads the rows in a forward-only fashion, you must iterate through the entire result set to find out the number of rows returned by the query.

Read This method moves the pointer in front of the next row in the result set. Use this method to read the rows of the result set, usually from within a While loop.

Get<type> There are many versions of the Get method with different names, depending on the type of column you want to read. To read a Decimal value, use the GetDecimal method; to retrieve a string, use the GetString method; to retrieve an integer, call one of the GetInt16, GetInt32, or GetSqlInt64 methods; and so on. To specify the column you want to read, use an integer index value that represents the column's ordinal, such as Reader.GetString(2). The index of the first column in the result set is zero.

GetSql<type> There are many versions of the `GetSql` method with different names, depending on the SQL type of the column you want to read. To read a Decimal value, use the `GetSqlDecimal` method; to retrieve a string, use the `GetSqlString` method; to retrieve an integer, call one of the `GetSqlInt16`, `GetSqlInt32`, or `GetSqlInt64` methods; and so on. To specify the column you want to read, use an integer index value that represents the column's ordinal, such as `Reader.GetSqlString(2)`. The index of the first column in the result set is zero.

CLR Types versus SQL Types

The Get<*Type*> methods return data types recognized by the Common Language Runtime (CLR), whereas the GetSql<*Type*> methods return data types recognized by SQL Server. There's a one-to-one correspondence between most types, but not always. In most cases, we use the Get<*Type*> methods and store the values in VB variables, but you may wish to store the value of a field in its native format.

GetValue If you can't be sure about the type of a column, use the `GetValue` method, which returns a value of the Object type. This method accepts as an argument the ordinal of the column you wish to read.

GetValues This method reads all the columns of the current row and stores them into an array of objects, which is passed to the method as an argument. This method returns an integer value, which is the number of columns read from the current row.

GetName Use this method to retrieve the name of a column, which must be specified by its order in the result set. To retrieve the name of the first column, use the expression `Reader.GetName(0)`. The column's name in the result set is the original column name, unless the `SELECT` statement used an alias to return a column with a different name.

GetOrdinal This is the counterpart of the `GetName` method, and it returns the ordinal of a specific column from its name. To retrieve the ordinal of the *`CompanyName`* column, use the expression `Reader.GetName("CompanyName")`.

IsDbNull This method returns True if the column specified by its ordinal in the current row is Null. If you attempt to assign a Null column to a variable, a runtime exception will be thrown, so you should use this method to determine whether a column has a value and handle the Null values from within your code.

Note that you can't reset the DataReader object and reread the same result set. To go through the same rows, you must execute the query again. However, there's no guarantee that the same query executed a few moments later will return the same result set. Rows may have been added to, or removed from, the database, so your best bet is to go through the result set once and store all the data to a structure at the client computer's memory. Moreover, while you're using the DataReader, the connection to the server remains open. This means that you shouldn't process the data as you read it, unless it is a trivial form of processing, such as keeping track of sums and counts. If you need to perform some substantial processing on your data, read the data into an ArrayList or other structure in the client computer's memory, close the connection, and then access the data in the ArrayList. Later in this chapter, you'll learn about the DataSet object, which was designed to maintain relational data at the client. The DataSet is a great structure for storing relational data at the client; it's almost like a small database that resides in the client computer's memory. However, the DataSet is not ideal for all situations.

Listing 22.3 shows the code that retrieves all products along with category names and supplier names and populates a ListView control. The ListView control's columns aren't specified at design time; the code adds the appropriate columns at runtime (as long as the `View` property has been set to *`Details`*). The code goes through the columns of the result set and adds a new column to the ListView control for each data column. Then it reads the rows returned by the query and displays them on the control. The statements in Listing 22.3 are part of the SimpleQueries sample project.

LISTING 22.3: Displaying Product Information on a ListView Control

```
Dim Command As New SqlCommand
Command.Connection = CN
' a simple SELECT query
Command.CommandText = _
   "SELECT ProductName AS Product, " & _
   "CategoryName AS Category, " & _
   "CompanyName AS Supplier, UnitPrice AS Price " & _
   "FROM Products LEFT JOIN Categories " & _
   "ON Products.CategoryID = Categories.CategoryID " & _
   "LEFT JOIN Suppliers ON Products.SupplierID = Suppliers.SupplierID"
Connection.Open()
Dim count As Integer = 0
Dim Reader As SqlDataReader
Reader = Command.ExecuteReader
ListView1.Clear()
Dim i As Integer
' setup ListView control to display the headers
' of the columns read from the database
For i = 0 To Reader.FieldCount - 1
    ListView1.Columns.Add(Reader.GetName(i), 130)
Next
While Reader.Read
    Dim LI As New ListViewItem
    LI.Text = Convert.ToString(Reader.Item("Product"))
    LI.SubItems.Add(Convert.ToString( _
                     Reader.Item("Category")))
    LI.SubItems.Add(Convert.ToString( _
                     Reader.Item("Supplier")))
    LI.SubItems.Add(Convert.ToString( _
                     Reader.Item("Price")))
    ListView1.Items.Add(LI)
    Count += 1
End While
MsgBox("Read " & count.ToString & " Product rows")
Connection.Close()
```

Reading Multiple Result Sets

Another interesting aspect of the DataReader object is that you can use it to read multiple result sets, such as the ones returned by multiple queries. You can execute a batch query such as the following with a single Command object:

```
Command.CommandText = "SELECT * FROM Customers; SELECT * FROM Employees"
Dim Reader As SqlDataReader = Command.ExecuteReader
```

We'll use the same DataReader object to read the rows of both tables, but we need to know when we're finished with the first result set (the customers) and start reading the second result set. The `NextResult` property of the DataReader does exactly that: After exhausting the first result set (by iterating through its rows with the `Read` method), we can request the `NextResult` property to find out whether the DataReader contains additional result sets. If so, we can start reading the next result set with the `Read` method. Here's the outline of the code for reading two result sets from the same DataReader:

```
While Reader.Read
    ' read the fields of the current row in the 1st result set
End While
If Reader.NextResult
    While Reader.Read
        ' read the fields of the current row in the 2nd result set
    End While
End If
```

Storing Data in DataSets

The process of building data-driven applications isn't complicated and to a large extent is abstracted by the Connection, Command, and DataReader classes. You have seen a few interesting examples of these classes and should be ready to use them in your applications. The problem with these classes is that they don't offer a consistent method for storing the data at the client. The approach of converting the data into business objects and working with classes is fine, but you must come up with a data-storage mechanism at the client. You can store the data in a ListBox control, as we have done in some examples. You can also create an ArrayList of custom objects. The issue of storing data at the client isn't pressing when the client application is connected to the database and all updates take place in real time. As soon as the user edits a row, the row is submitted to the database and no work is lost.

In some situations, however, the client isn't connected at all times. There's actually a class of applications called *occasionally connected* or *disconnected*, and the techniques presented so far do not address the needs of these applications. Disconnected applications read some data when a connection is available and then they disconnect from the server. Users are allowed to interact with the data at the client, but they work with a local cache of the data; they can insert new rows, edit existing ones, and delete selected rows. The changes, however, are not submitted immediately to the server. It's imperative that the data is persisted at the client. (We don't want users to lose their edits because their notebook ran out of batteries or because of a bug in the application.) When a connection becomes available again, the application should be able to figure out the rows that have been edited and submit the appropriate changes to the server. To simplify the storage of data at the client, ADO.NET offers a powerful mechanism, the DataSet.

You can think of the DataSet as a small database that lives in memory. It's not actually a database, but it's made up of related tables that have the same structure as database tables. The similarities end there, however, because the DataSet doesn't impose all types of constraints and you can't exploit its data with SQL statements. It's made up of DataTable objects, and each DataTable in the DataSet corresponds to a separate query. Like database tables, the DataTable objects are made up of DataColumn and DataRow objects. The DataColumn objects specify the structure of the table, and the DataRow objects contain the rows of the table. You can also establish relations between the tables in the DataSet, and these relations are represented with DataRelation objects. As you realize, the DataSet lets you copy a small section of the database at the client, work with it, and then submit the changes made at the client back to the database.

The real power of the DataSet is that it keeps track of the changes made to its data. It knows which rows have been modified, added, or deleted, and it provides a mechanism for submitting the changes automatically. Actually, it's not the DataSet that submits the changes, but a class that's used in tandem with the DataSet: the DataAdapter class. Moreover, the DataSet class provides the `WriteXml` and `ReadXml` methods, which allow you to save its data to a local file. Note that these methods save the data to a local file so you can reload the DataSet later, but they do not submit the data to the database.

Filling DataSets

DataSets are filled with DataAdapters, and there are two ways to create a DataSet: You can use the visual tools of Visual Studio or create a DataSet entirely from within your code. DataSets created at runtime are not typed, because the compiler doesn't know what type of information you're going to store in them. DataSets created at design time with the visual tools are strongly typed, because the compiler knows what type of information will be stored in them.

The following statements demonstrate the difference between untyped and typed DataSets. To access the *`ProductName`* column of the first row in the Products table in an untyped DataSet, you'd use an expression like the following:

```
Products1.Products.Rows(0).Item("ProductName")
```

If the *`Products1`* DataSet is typed, you can create an object of the `Products.ProductsRow` type with the following statement:

```
Dim productRow As Products.ProductsRow = Products1.Products.Rows(0)
```

Then use the *`productRow`* variable to access the columns of the corresponding row:

```
productRow.ProductName
productRow.UnitPrice
```

As you can understand, the visual tools generate a number of classes on-the-fly, such as the ProductsRow class, and expose them to your code. As soon as you enter the string *`productRow`* and the following period in the code window, you will see the members of the ProductsRow class, which include the names of the columns in the corresponding table. In this chapter, I discuss untyped DataSets. In the following chapter, I'll discuss in detail typed DataSets and how to use them in building data-bound applications.

The DataAdapter Class

To use DataSets in your application, you must first create a DataAdapter object, which is the preferred technique of populating the DataSet. The DataAdapter is nothing more than a collection of Command objects that are needed to execute the various commands against the database. As you recall from our previous discussion, we interact with the database by using four different Command objects: one to select the data and load them to the client computer with the help of a DataReader object (a Command object with the `SELECT` statement), and three more to submit to the database the new rows (a Command object with the `INSERT` statement), update existing rows (a Command object with the `UPDATE` statement) and delete existing rows (a Command object with the `DELETE` statement). A DataAdapter is a container for Connection and Command objects. If you declare a SqlDataAdapter object with a statement like the following:

```
Dim DA As New SqlDataAdapter
```

you'll see that it exposes the following properties:

InsertCommand	A Command object that's executed to insert a new row
UpdateCommand	A Command object that's executed to update a row
DeleteCommand	A Command object that's executed to delete a row
SelectCommand	A Command object that's executed to retrieve selected rows

Each of these properties is an object and has its own Connection property, because each may not act on the same database (as unlikely as it may be). These properties also expose their own `Parameters` collection, which you must populate accordingly before executing a command.

The DataAdapter class performs the two basic tasks of a data-driven application: It retrieves data from the database to populate a DataSet, and submits the changes to the database. To populate a DataSet, use the `Fill` method, which fills a specific DataTable object. There's one DataAdapter per DataTable object in the DataSet, and you must call the corresponding `Fill` method to populate each DataTable. To submit the changes to the database, use the `Update` method of the appropriate DataAdapter object. The `Update` method is overloaded, and you can use it to submit a single row to the database or all edited rows in a DataTable. The `Update` method uses the appropriate Command object to interact with the database.

Passing Parameters through the DataAdapter

Let's build a DataSet in our code to demonstrate the use of the DataAdapter objects. Start by declaring a DataSet variable:

```
Dim DS As New DataSet
```

Then create the various commands that will interact with the database:

```
Dim cmdSelectCustomers As String = "SELECT * FROM Customers " & _
             "WHERE Customers.Country=@country"
Dim cmdDeleteCustomer As String = "DELETE Customers WHERE CustomerID=@CustomerID"
Dim cmdEditCustomer As String = "UPDATE Customers " & _
             "SET CustomerID = @CustomerID, CompanyName = @CompanyName, " & _
             "ContactName = @ContactName, ContactTitle = @ContactTitle " & _
             "WHERE CustomerID = @CustID"
Dim cmdInsertCustomer As String = "INSERT Customers " & _
             "(CustomerID, CompanyName, ContactName, ContactTitle) " & _
             "VALUES(@CustomerID, @CompanyName, @ContactName, @ContactTitle) "
```

I've included only a few columns in the examples to keep the statements reasonably short. The various commands use parameterized queries to interact with the database, and we must add the appropriate parameters to each Command object. After the SQL statements are in place, we can build the four Command properties of the DataAdapter object. Start by declaring a DataAdapter object:

```
Dim DACustomers As New SqlDataAdapter()
```

Because all Command properties of the DataAdapter object will act on the same database, we can create a Connection object and reuse it as needed:

```
Dim CN As New SqlConnection(ConnString)
```

The *`ConnString`* variable is a string with the proper connection string. Now we can create the four Command properties of the *`DACustomers`* DataAdapter object.

Let's start with the `SelectCommand` property of the DataAdapter object. The following statements create a new Command object based on the preceding `SELECT` statement and then set up a Parameter object for the *`@country`* parameter of the `SELECT` statement:

```
DACustomers.SelectCommand = New SqlClient.SqlCommand(cmdSelectCustomers)
DACustomers.SelectCommand.Connection = CN
Dim param As New SqlParameter
param.ParameterName = "@Country"
param.SqlDbType = SqlDbType.VarChar
param.Size = 15
param.Direction = ParameterDirection.Input
param.IsNullable = False
param.Value = "Germany"
DACustomers.SelectCommand.Parameters.Add(param)
```

This is the easier, but rather verbose, method of specifying a Parameter object. You are familiar with the Parameter object's properties and already know how to configure and add parameters to a Command object via a single statement. As a reminder, an overloaded form of the `Add` method allows you to configure and attach a Parameter object to a Command object's `Parameters` collection with a single, but lengthy, statement:

```
DA.SelectCommand.Parameters.Add( _
     New System.Data.SqlClient.qlParameter( _
     paramName, paramType, paramSize, paramDirection, _
     paramNullable, paramPrecision, paramScale, _
     columnName, rowVersion, paramValue)
```

The *paramPrecsion* and *paramScale* arguments apply to numeric parameters, and you can set them to 0 for string parameters. The *paramNullable* argument determines whether the parameter can assume a Null value. The *columnName* argument is the name of the table column to which the parameter will be matched. (We need this information for the INSERT and UPDATE commands.) The *rowVersion* argument determines which version of the field in the DataSet will be used — in other words, whether the DataAdapter will pass to the parameter object the current version (*DataRowVersion.Current*) or the original version (*DataRowVersion.Original*) of the field. The last argument, *paramValue*, is the parameter's value. You can specify a value as we did in the example of the SelectCommand, or set this argument to Nothing and let the DataAdapter object assign the proper value to each parameter. (You'll see in a moment how this argument is used with the INSERT and UPDATE commands.)

Finally, you can open the connection to the database and then call the DataAdapter's Fill method to populate a DataTable in the DataSet:

```
CN.Open
DACustomers.Fill(DS, "Customers")
CN.Close
```

The Fill method accepts as arguments a DataSet object and the name of the DataTable it will populate. The *DACustomers* DataAdapter is associated with a single DataTable and knows how to populate it, as well as how to submit the changes back to the database. The DataTable's name is arbitrary and need not match the name of the database table where the data originates. The four basic operations of the DataAdapter (which are no other than the four basic data-access operations of a client application) are also known as CRUD operations: Create/Retrieve/Update/Delete.

The CommandBuilder Class

Each DataAdapter object that you set up in your code is associated with a single SELECT query, which may select data from one or multiple joined tables. The INSERT/UPDATE/DELETE queries of the DataAdapter can submit data to a single table. So far you've seen how to manually set up each Command object in a DataAdapter object. There's a simpler method to specify the queries: You start with the SELECT statement, which selects data from a single table, and then let a Command-Builder object infer the other three statements from the SELECT statement. Let's see this technique in action.

Declare a new SqlCommandBuilder object by passing the name of the adapter for which you want to generate the statements:

```
Dim CustomersCB As SqlCommandBuilder = _
                New SqlCommandBuilder(DA)
```

This statement is all it takes to generate the InsertCommand, UpdateCommand, and Delete-Command objects of the *DACustomers* SqlDataAdapter object. When the compiler runs into the previous statement, it will generate the appropriate Command objects and attach them to the *DACustomers* SqlDataAdapter. Here are the SQL statements generated by the CommandBuilder object for the Products table of the Northwind database:

***UPDATE* Command**

```
UPDATE [Products] SET [ProductName] = @p1,
       [CategoryID] = @p2, [UnitPrice] = @p3,
       [UnitsInStock] = @p4, [UnitsOnOrder] = @p5
WHERE  (([ProductID] = @p6))
```

INSERT Command

```
INSERT INTO [Products]
         ([ProductName], [CategoryID],
          [UnitPrice], [UnitsInStock],
          [UnitsOnOrder])
          VALUES (@p1, @p2, @p3, @p4, @p5)
```

DELETE Command

```
DELETE FROM [Products] WHERE (([ProductID] = @p1))
```

These statements are based on the SELECT statement and are quite simple. You may notice that the UPDATE statement simply overrides the current values in the Products table. The CommandBuilder can generate a more elaborate statement that takes into consideration concurrency. It can generate a statement that compares the values read into the DataSet to the values stored in the database. If these values are different, which means that another user has edited the same row since the row was read into the DataSet, it doesn't perform the update. To specify the type of UPDATE statement you want to create with the CommandBuilder object, set its ConflictOption property, whose value is a member of the ConflictOption enumeration: *CompareAllSearchValues* (compares the values of all columns specified in the SELECT statement), *CompareRowVersion* (compares the original and current versions of the row), and *OverwriteChanges* (simply overwrites the fields of the current row in the database).

The *OverwriteChanges* option generates a simple statement that locates the row to be updated with its ID and overwrites the current field values unconditionally. If you set the ConflictOption property to *CompareAllSearchValues*, the CommandBuilder will generate the following UPDATE statement:

```
UPDATE [Products]
SET    [ProductName] = @p1, [CategoryID] = @p2,
       [UnitPrice] = @p3, [UnitsInStock] = @p4,
       [UnitsOnOrder] = @p5
WHERE  (([ProductID] = @p6) AND ([ProductName] = @p7)
       AND ((@p8 = 1 AND [CategoryID] IS NULL) OR
       ([CategoryID] = @p9)) AND
       ((@p10 = 1 AND [UnitPrice] IS NULL) OR
       ([UnitPrice] = @p11)) AND
       ((@p12 = 1 AND [UnitsInStock] IS NULL) OR
       ([UnitsInStock] = @p13)) AND
       ((@p14 = 1 AND [UnitsOnOrder] IS NULL) OR
       ([UnitsOnOrder] = @p15)))
```

This is a lengthy statement indeed. The row to be updated is identified by its ID, but the operation doesn't take place if any of the other fields don't match the value read into the DataSet. This statement will fail to update the corresponding row in the Products table if it has already been edited by another user.

The last member of the `ConflictOption` enumeration, the *`CompareRowVersion`* option, works with tables that have a `TimeStamp` column, which is automatically set to the time of the update. If the row has a time stamp that's later than the value read when the DataSet was populated, it means that the row has been updated already by another user and the `UPDATE` statement will fail.

The SimpleDataSet sample project, which is discussed later in this chapter and demonstrates the basic DataSet operations, generates the `UPDATE/INSERT/DELETE` statements for the Categories and Products tables with the help of the CommandBuilder class and displays them on the form when the application starts. Open the project to examine the code and change the setting of the `ConflictOption` property to see how it affects the autogenerated SQL statements.

Accessing the DataSet's Tables

The DataSet is made up of tables, which are represented by the DataTable class. Each DataTable in the DataSet may correspond to a table in the database or a view. When you execute a query that retrieves fields from multiple tables, all selected columns will end up in a single DataTable of the DataSet. You can select any DataTable in the DataSet by its index or its name:

```
DS.Tables(0)
DS.Tables("Customers")
```

Each table contains columns, which you can access through the `Columns` collection. The `Columns` collection is made up of DataColumn objects, one DataColumn object for each column in the corresponding table. The `Columns` collection is the schema of the DataTable object, and the DataColumn class exposes properties that describe a column. `ColumnName` is the column's name, `DataType` is the column's type, `MaxLength` is the maximum size of text columns, and so on. The `AutoIncrement` property is True for Identity columns, and the `AllowDBNull` property determines whether the column allows Null values. In short, all the properties you can set visually as you design a table are also available to your code through the `Columns` collection of the DataTable object. You can use the DataColumn class's properties to find out the structure of the table or to create a new table. To add a table to a DataSet, you can create a new DataTable object. Then create a DataColumn object for each column, set its properties, and add the DataColumn objects to the DataTable's `Columns` collection. Finally, add the DataTable to the DataSet. The process is described in detail in the online documentation, so I won't repeat it here.

Working with Rows

As far as data are concerned, each DataTable is made up of DataRow objects. All DataRow objects of a DataTable have the same structure and can be accessed through an index, which is the row's order in the table. To access the rows of the Customers table, use an expression like the following:

```
DS.Customers.Rows(iRow)
```

where *`iRow`* is an integer value from zero (the first row in the table) up to `DS.Customers.Rows.Count – 1` (the last row in the table). To access the individual fields of a DataRow object, use the `Item` property. This property returns the value of a column in the current row by either its index or its name:

```
DS.Customers.Rows(0).Item(0)
```

or

```
DS.Customers.Rows(0).Item("CustomerID")
```

To iterate through the rows of a DataSet, you can set up a `For...Next` loop like the following:

```
Dim iRow As Integer
For iRow = 0 To DSProducts1.Products.Rows.Count - 1
    ' process row: DSProducts.Products.Rows(iRow)
Next
```

Alternatively, you can use a `For Each...Next` loop to iterate through the rows of the DataTable:

```
Dim product As DataRow
For Each product In DSProducts1.Products.Rows
    ' process prodRow row:
    ' product.Item("ProductName"),
    ' product.Item("UnitPrice"), and so on
Next
```

To edit a specific row, simply assign new values to its columns. To change the value of the *`ContactName`* column of a specific row in a DataTable that holds the customers of the Northwind database, use a statement like the following:

```
DS.Customers(3).Item("ContactName") = "new contact name"
```

The new values are usually entered by a user on the appropriate interface, and in your code you'll most likely assign a control's property to a row's column with statements like the following:

```
If txtName.Text.Trim <> "" Then
    DS.Customers(3).Item("ContactName") = txtName.Text
Else
    DS.Customers(3).Item("ContactName") = DBNull.Value
End If
```

The code segment assumes that when the user doesn't supply a value for a column, this column is set to Null (if the column is nullable, of course). If the control contains a value, this value is assigned to the *`ContactName`* column of the fourth row in the Customers DataTable of the *`DS`* DataSet.

Handling Null Values

An important (and quite often tricky) issue in coding data-driven applications is the handling of Null values. Null values are special, in the sense that you can't assign them to control properties or use them in other expressions. Every expression that involves Null values will throw a runtime exception. The DataRow object provides the `IsNull` method, which returns True if the column specified by its argument is a Null value:

```
If customerRow.IsNull("ContactName") Then
    ' handle Null value
```

```
Else
    ' process value
End If
```

In a typed DataSet, DataRow objects provide a separate method to determine whether a specific column has a Null value. If the *customerRow* DataRow belongs to a typed DataSet, you can use the `IsContactNameNull` method instead:

```
If customerRow.IsContactNameNull Then
    ' handle Null value for the ContactName
Else
    ' process value: customerRow.ContactName
End If
```

If you need to map Null columns to specific values, you can do so with the `ISNULL()` function of T-SQL, as you retrieve the data from the database. In many applications, we want to display an empty string or a zero value in place of a Null field. We can avoid all the comparisons in our code by retrieving the corresponding field with the `ISNULL()` function in our SQL statement. Where the column name would appear in the `SELECT` statement, use an expression like the following:

```
ISNULL(customerBalance, 0.00)
```

If the *customerBalance* column is Null for a specific row, SQL Server will return the numeric value zero. This value can be used in reports or other calculations in your code. Notice that the customer's balance shouldn't be Null. A customer always has a balance, even if it's zero. When a product's price is Null, it means that we don't know the price of the product (and therefore can't sell it). In this case, a Null value can't be substituted with a zero value. You must always carefully handle Null columns in your code, and how you'll handle them depends on the nature of the data they represent.

Adding and Deleting Rows

To add a new row to a DataTable, you must first create a DataRow object, set its column values, and then call the `Add` method of the `Rows` collection of the DataTable to which the new row belongs, passing the new row as an argument. If the *DS* DataSet contains the Customers DataTable, the following statements will add a new row for the Customers table:

```
Dim newRow As New DataRow = dataTable.NewRow
newRow.Item("CompanyName") = "new company name"
newRow.Item("CustomerName") = "new customer name"
newRow.Item("ContactName") = "new contact name"
DS.Customers.Rows.Add(newRow)
```

Notice that we need not set the *CustomerID* column. This Identity column is assigned a new value automatically by the DataSet. Of course, when the row is submitted to the database, the ID assigned to the new customer by the DataSet may already be taken. SQL Server will assign a new unique value to this column when it inserts it into the table. It's recommended that you set the `AutoIncrementSeed` property of an Identity column to 0, and the `AutoIncrement` to −1, so that new rows are assigned consecutive negative IDs in the DataSet. Presumably, the corresponding columns in the database have a positive AutoIncrement setting, so when these rows are submitted to the database, they're assigned the next Identity value automatically. If you're designing a new database, use globally unique identifiers (GUIDs) instead of identity values. A GUID can be created at the client and is unique. The same GUID that will be generated by the client will also be inserted in the table, when the row is committed. To create GUIDs, call the NewGuid method of the Guid class:

```
newRow.Item("CustomerID") = Guid.NewGuid
```

To delete a row, you can either remove it from the Rows collection via the Remove or the RemoveAt methods of the Rows collection, or call the Delete method of the DataRow object that represents the row. The Remove method accepts as an argument a DataRow object and removes it from the collection:

```
Dim customerRow As DS.CustomerRow
customerRow = DS.Customers.Rows(2)
DS.Customers.Remove(customerRow)
```

The RemoveAt method accepts as an argument the index of the row you want to delete in the Rows collection. Finally, the Delete method is a method of the DataRow class, and you must apply it to a DataRow object that represents the row to be deleted:

```
customerRow.Delete
```

Deleting versus Removing Rows

The Remove method removes a row from the DataSet as if it were never read when the DataSet was filled. Deleted rows are not always removed from the DataSet, because the DataSet maintains its state. If the row you've deleted exists in the underlying table (in other words, if it's a row that was read into the DataSet when you filled it), the row will be marked as deleted but will not be removed from the DataSet. If it's a row that was added to the DataSet after it was read from the database, the deleted row is actually removed from the Rows collection.

You can physically remove deleted rows from the DataSet by calling the DataSet's AcceptChanges method. However, after you've accepted the changes in the DataSet, you can no longer submit any updates to the database. If you call the DataSet's RejectChanges method, the deleted rows will be restored in the DataSet.

Navigating through a DataSet

The DataTables making up a DataSet may be related, and they usually are. There are methods that allow you to navigate from table to table following the relations between their rows. For example, you can start with a row in the Customers DataTable, retrieve its child rows in the Orders DataTable (the orders placed by the selected customer), and then drill down to the details of each of the selected orders.

The relations of a DataSet are DataRelation objects and are stored in the Relations property of the DataSet. Each relation is identified by a name, the two tables it relates to, and the fields of the tables on which the relation is based. It's possible to create relations in your code, and the process is really quite simple. Let's consider a DataSet that contains the Categories and Products tables. To establish a relation between the two tables, create two instances of the DataTable object to reference the two tables:

```
Dim tblCategories As DataTable = DS.Categories
Dim tblProducts As DataTable = DS.Products
```

Then create two DataColumn objects to reference the columns on which the relation is based. They're the *`CategoryID`* columns of both tables:

```
Dim colCatCategoryID As DataColumn = _
                  tblCategories.Columns("CategoryID")
Dim colProdCategoryID As DataColumn = _
                  tblProducts.Columns("CategoryID")
```

And finally, create a new DataRelation object and add it to the DataSet:

```
Dim DR As DataRelation
DR = New DataRelation("Categories2Products", _
                colCatCategoryID, colProdCategoryID)
```

Notice that we need to specify only the columns involved in the relation, and not the tables to be related. The information about the tables is derived from the DataColumn objects. The first argument of the DataRelation constructor is the relation's name. If the relation involves multiple columns, the second and third arguments of the constructor become arrays of DataColumn objects.

To navigate through related tables, the DataRow object provides the `GetChildRows` method, which returns the current row's child rows as an array of DataRow objects, and the `GetParent-Row`/`GetParentRows` methods, which return the current row's parent row(s). `GetParentRow` returns a single DataRow object, and `GetParentRows` returns an array of DataRow objects. Because a DataTable may be related to multiple DataTables, you must also specify the name of the relation. Consider a DataSet with the Products, Categories, and Suppliers tables. Each row of the Products table can have two parent rows, depending on which relation you want to follow. To retrieve the product's category, use a statement like the following:

```
DS.Products(iRow).GetParentRow("CategoriesProducts")
```

The product's supplier is given by the following expression:

```
DS.Products(iRow).GetParentRow("SuppliersProducts")
```

If you start with a category, you can find out the related products with the `GetChildRows` method, which accepts as an argument the name of a Relation object:

```
DS.Categories(iRow).GetChildRows("CategoriesProducts")
```

To iterate through the products of a specific category (in other words, the rows of the Products table that belong to a category), set up a loop like the following:

```
Dim product As DataRow
For Each product In DS.Categories(iRow). _
                    GetChildRows("CategoriesProducts")
' process product
Next
```

Row States and Versions

Each row in the DataSet has a `State` property. This property indicates the row's state and its value if a member of the `DataRowState` enumeration, whose members are the following:

Added The row has been added to the DataTable, and the `AcceptChanges` method has not been called.

Deleted The row was deleted from the DataTable, and the `AcceptChanges` method has not been called.

Detached The row has been created with its constructor but has not yet been added to a DataTable.

Modified The row has been edited, and the `AcceptChanges` method has not been called.

Unchanged The row has not been edited or deleted since it was read from the database or the `AcceptChanges` was last called. (In other words, the row's fields are identical to the values read from the database.)

You can use the `GetChanges` method to find out the rows that must be added to the underlying table in the database, the rows to be updated, and the rows to be removed from the underlying table.

If you want to update all rows of a DataTable, call an overloaded form of the DataAdapter's `Update` method, which accepts as an argument a DataTable and submits its rows to the database. The edited rows are submitted through the UpdateCommand object of the appropriate DataAdapter, the new rows are submitted through the InsertCommand object, and the deleted rows are submitted through the DeleteCommand object. Instead of submitting the entire table, however, you can create a subset of a DataTable that contains only the rows that have been edited, inserted, or deleted. The `GetChanges` method of the DataTable object retrieves a subset of rows, depending on the argument you pass to it, and this argument is a member of the `DataRowState` enumeration:

```
Dim DT As New DataTable = _
    Products1.Products.GetChanges(DataRowState.Deleted)
```

This statement retrieves the rows of the Customers table that were deleted and stores them into a new DataTable. The new DataTable has the same structure as the one from which the rows were copied, and you can access its rows and their columns as you would access any DataTable of a DataSet. You can even pass this DataTable as an argument to the appropriate DataAdapter's

`Update` method. This form of the `Update` method allows you to submit selected changes to the database.

In addition to a state, rows have a version. What makes the DataSet such a powerful tool for disconnected applications is that it maintains not only data, but also the changes in its data. The `Rows` property of the DataTable object is usually called with the index of the desired row, but it accepts a second argument, which determines the version of the row you want to read:

```
DS.Tables(0).Rows(i, version)
```

This argument is a member of the `DataRowVersion` enumeration, whose values are the following:

Current Returns the row's current values (the fields as they were edited in the DataSet).

Default Returns the default values for the row. For added, edited, and current rows, the default version is the same as the current version. For deleted rows, the default versions are the same as the original versions. If the row doesn't belong to a DataTable, the default version is the same as the proposed version.

Original Returns the row's original values (the values read from the database).

Proposed Returns the row's proposed value (the values assigned to a row that doesn't yet belong to a DataTable).

If you attempt to submit an edited row to the database and the operation fails, you can give the user the option to edit the row's current version or to restore the row's original values. To retrieve the original version of a row, use an expression like the following:

```
DS.Tables(0).Row(i, DataRowVersion.Original)
```

Although you can't manipulate the version of a row directly, you can use the `AcceptChanges` and `RejectChanges` methods to either accept the changes or reject them. These two methods are exposed by the DataSet, DataTable, and DataRow classes. The difference is the scope: Applying `RejectChanges` to the DataSet restores all changes made to the DataSet (not a very practical operation), whereas applying `RejectChanges` to a DataTable object restores the changes made to the specific table's rows; applying the same method to the DataRow object restores the changes made to a single row.

The `AcceptChanges` method sets the original value of the affected row(s) to the proposed value. Deleted rows are physically removed. The `RejectChanges` method removes the proposed version of the affected row(s). You can call the `RejectChanges` method when the user wants to get rid of all changes in the DataSet. Notice that after you call the `AcceptChanges` method, you can no longer update the underlying tables in the database, because the DataSet no longer knows which rows were edited, inserted, or deleted. Call the `AcceptChanges` method only for DataSets you plan to persist on disk and not submit to the database.

Update Operations

One of the most important topics in database programming is how to submit changes back to the database. There are basically two modes of operation: single updates and multiple updates. A client application running on a local area network as the database server can (and should) submit changes as soon as they occur. If the client application is not connected to the database server at all times, changes may accumulate at the client and can be submitted in batch mode when a connection to the server is available.

From a developer's point of view, the difference between the two modes is how you handle update errors. If you submit individual rows to the database and the update operation fails, you can display a warning and let the user edit the data again. You can write code to restore the row to its original state, or not. In any case, it's fairly easy to handle isolated errors. If the application submits a few dozen rows to the database, several of these rows may fail to update the underlying table and you'll have to handle the update errors from within your code. At the very least, you must validate the data as best as you can at the client before submitting them to the database. No matter how thoroughly you validate your data, however, you can't be sure that they will be inserted into the database successfully.

Another factor you should consider is the nature of the data you work with. Let's consider an application that maintains a database of books and an application that takes orders. The book maintenance application handles publishers, authors, translators, and other data. If two dozen users are entering and correcting titles, they will all work with the same authors. If you allow them to work in disconnected mode, the same author name may be entered several times, because no user can see the changes made by any other user. This application should be connected: Every time a user adds a new author, the table with the author names in the database must be updated, so that other users can see the new author. The same goes for publishers, translators, topics, and so on. A disconnected application of this type should also include utilities to consolidate multiple author and publisher names.

An order-taking application can safely work in a disconnected mode, because orders entered by one user are not aware of and don't interfere with the orders entered by another user. You can install the client application on the notebooks of several salespersons so they can take orders on the go and upload them after establishing a connection between the notebook and the database server (which may even happen when the salespersons return to the company's offices).

Updating the Database with the DataAdapter

The simplest method of submitting changes to the database is to use each DataAdapter's `Update` method. The DataTable object provides the members you need to retrieve the rows that failed to update the database, as well as the messages returned by the database server, and you'll see how these members are used in this section. The `Update` method may not have updated all the rows in the underlying tables. If a product was removed in the meantime from the Products table in the database, the DataAdapter's `UpdateCommand` will not be able to submit the changes made to the specific product. A product with a negative value may very well exist at the client, but the database will reject this row, because it violates one of the constraints of the Products table.

If the database returned any errors during the update process, the `HasErrors` property of the DataSet object will be set to True. You can retrieve the rows in error from each table with the `GetErrors` method of the DataTable class. This method returns an array of DataRow objects, and you can process them in any way you see fit. The code shown in Listing 22.4 iterates through the rows of the Categories table that are in error and prints the description of the error in the Output window.

LISTING 22.4: Retrieving and Displaying the Update Errors

```
If Products1.HasErrors Then
    If Products1.Categories.GetErrors.Length = 0 Then
```

```
            Console.WriteLine("Errors in the Categories DataTable")
                Else
            Dim RowsInError() As Products.CategoriesRow
            RowsInError = Products1.Categories.GetErrors
            Dim row As Products.CategoriesRow
            Console.WriteLine("Errors in the Categories table")
            For Each row In RowsInError
                Console.WriteLine(vbTab & row.CategoryID & vbTab & _
                            row.RowError)
            Next
        End If
    Endif
```

The DataRow object exposes the `RowError` property, which is a description of the error that prevented the update for the specific row. It's possible that the same row has more than a single error. To retrieve all columns in error, call the DataRow object's `GetColumnsInError` method, which returns an array of DataColumn objects that are the columns in error.

Handling Identity Columns

An issue that deserves special attention in coding data-driven applications is the handling of Identity columns. Identity columns are used as primary keys, and each row is guaranteed to have a unique Identity value because this value is assigned by the database the moment the row is inserted into its table. The client application can't generate unique values. When new rows are added to a DataSet, they're assigned Identity values, but these values are unique in the context of the local DataSet. When a row is submitted to the database, any Identity column will be assigned its final value by the database. The temporary Identity value assigned by the DataSet is also used as a foreign key value by the related rows, and we must make sure that every time an Identity value is changed, the change will propagate to the related tables.

Handling Identity values is an important topic, and here's why: Consider an application for entering orders or invoices. Each order has a header and a number of detail lines, which are related to a header row with the *`OrderID`* column. This column is the primary key in the Orders table and is the foreign key in the Order Details table. If the primary key of a header is changed, the foreign keys of the related rows must change also.

The trick in handling Identity columns is to make sure that the values generated by the DataSet will be replaced by the database. We do so by specifying that the Identity column's starting value is −1 and its autoincrement is −1. The first ID generated by the DataSet will be −1, the second one will be −2, and so on. Negative Identity values will be rejected by the database, because the `AutoIncrement` properties in the database schema are positive. By submitting negative Identity values to SQL Server, we make sure that new, positive values will be generated and used by SQL Server.

We must also make sure that the new values will replace the old ones in the related rows. In other words, we want these values to propagate to all related rows. The DataSet allows you to specify that changes in the primary key will propagate through the related rows with the `UpdateRule` property of the `Relation.ChildKeyConstraint` property. Each relation exposes the `ChildKeyConstraint` property, which determines how changes in the primary key of a relation affect the child rows. This property is an object that exposes a few properties of its own. The two properties we're interested in are `UpdateRule` and `DeleteRule` (what happens to the child rows when the parent row's primary key is changed, or when the primary key is deleted). You can use one of the following rules:

Cascade Foreign keys in related rows change every time the primary key changes value, so that they'll always remain related to their parent row.

None The foreign key in the related row(s) is not affected.

SetDefault The foreign key in the related row(s) is set to the `DefaultValue` property for the same column.

SetNull The foreign key in the related rows is set to Null.

As you can understand, setting the `UpdateRule` property to anything other than *`Cascade`* will break the relation. If the database doesn't enforce the relation, you may be able to break it. If the relation is enforced, however, `UpdateRule` must be set to *`Rule.Cascade`*, or the database will not accept changes that violate its referential integrity.

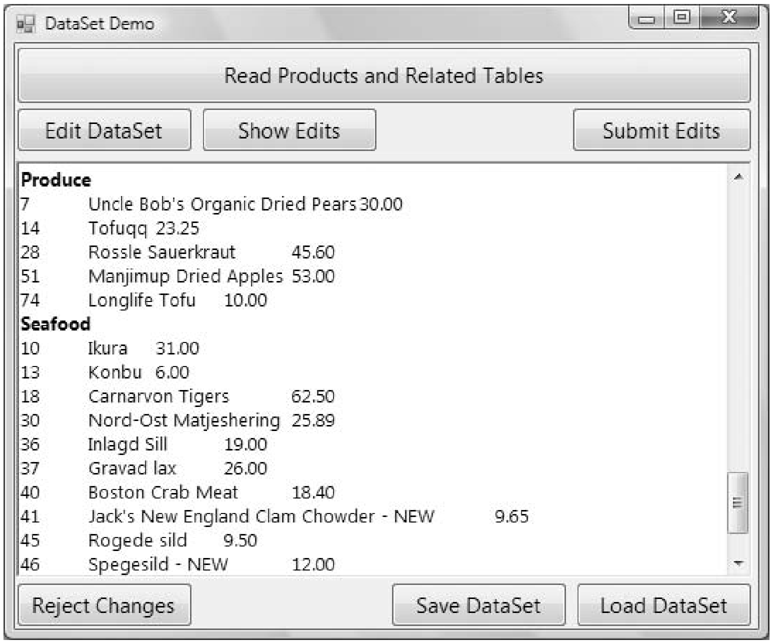
If you set `UpdateRule` to *`None`*, you may be able to submit the order to the database. However, the detail rows may refer to a different order. This will happen when the ID of the header is changed because the temporary value is already taken. The detail rows will be inserted with the temporary key and added to the details of another order. Notice that no runtime exception will be thrown, and the only way to catch this type of error is by examining the data inserted into the database by your application. By using negative values at the DataSet, we make sure that the ID of both the header and all detail rows will be rejected by the database.

VB 2008 at Work: The SimpleDataSet Project

Let's put together the topics discussed so far to build an application that uses a DataSet to store and edit data at the client. The sample application is called SimpleDataSet, and its interface is shown in Figure 22.1.

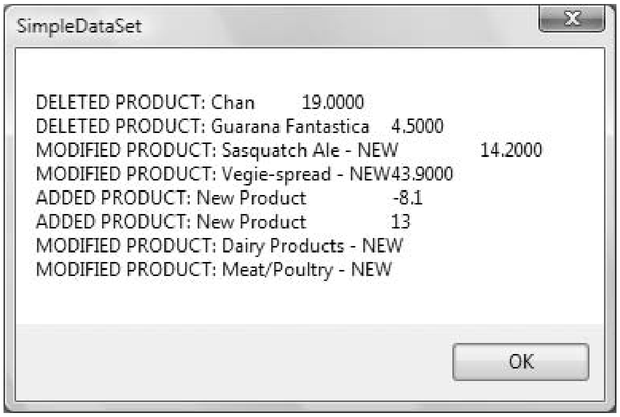
Click the large Read Products and Related Tables button at the top to populate a DataSet with the rows of the Products and Categories tables of the Northwind database. The application displays the categories and the products in each category in a RichTextBox control. Instead of displaying all the columns in a ListView control, I've chosen to display only a few columns of the Products table. The Edit DataSet button edits a few rows of both tables. The code behind this button changes the name and price of a couple of products in random, deletes a row, and adds a new row. It actually sets the price of the edited products to a random value in the range from −10 to 40 (negative prices are invalid and they will be rejected by the database). The DataSet keeps track of the changes, and you can review them at any time by clicking the Show Edits button, which displays the changes in the DataSet in a message box, like the one shown in Figure 22.2.

You can undo the changes and reset the DataSet to its original state by clicking the Reject Changes button, which calls the `RejectChanges` method of the DataSet class to reject the edits. It removes the new rows, restores the deleted ones, and undoes the edits in the modified rows.



```
Produce
7       Uncle Bob's Organic Dried Pears 30.00
14      Tofuqq  23.25
28      Rossle Sauerkraut       45.60
51      Manjimup Dried Apples   53.00
74      Longlife Tofu   10.00
Seafood
10      Ikura   31.00
13      Konbu   6.00
18      Carnarvon Tigers        62.50
30      Nord-Ost Matjeshering   25.89
36      Inlagd Sill     19.00
37      Gravad lax      26.00
40      Boston Crab Meat        18.40
41      Jack's New England Clam Chowder - NEW          9.65
45      Rogede sild     9.50
46      Spegesild - NEW         12.00
```

FIGURE 22.1 The SimpleDataSet project populates a DataSet at the client with categories and products.



```
DELETED PRODUCT: Chan        19.0000
DELETED PRODUCT: Guarana Fantastica    4.5000
MODIFIED PRODUCT: Sasquatch Ale - NEW            14.2000
MODIFIED PRODUCT: Vegie-spread - NEW43.9000
ADDED PRODUCT: New Product             -8.1
ADDED PRODUCT: New Product             13
MODIFIED PRODUCT: Dairy Products - NEW
MODIFIED PRODUCT: Meat/Poultry - NEW
```

FIGURE 22.2 Viewing the changes in the client DataSet

The Save DataSet and Load DataSet buttons persist the DataSet at the client, so that you can reload it later without having to access the database. The code shown in Listing 22.5 calls the `WriteXml` and `ReadXml` methods and uses a hard-coded filename. `WriteXml` and `ReadXml` save the data only, and you can't create a DataSet by calling the `ReadXml` method; this method will populate an existing DataSet. To actually load a DataSet, you must first specify its structure. Fortunately, the DataSet exposes the `WriteXmlSchema` and `ReadXmlSchema` methods, which store and read the schema of the DataSet. `WriteXmlSchema` saves the schema of the DataSet, so you can regenerate an identical DataSet with the `ReadXmlSchema` method, which reads an existing schema and structures the DataSet accordingly. The code behind the Save DataSet and Load DataSet buttons first calls these two methods to take care of the DataSet's schema, and then calls the `WriteXml` and `ReadXml` methods to save/load the data.

LISTING 22.5: Saving and Loading the DataSet

```
Private Sub bttnSave_Click(...) Handles bttnSave.Click
    Try
        DS.WriteXmlSchema("DataSetSchema.xml")
        DS.WriteXml("DataSetData.xml", XmlWriteMode.DiffGram)
    Catch ex As Exception
        MsgBox("Failed to save DataSet" & vbCrLf & ex.Message)
        Exit Sub
    End Try
    MsgBox("DataSet saved successfully")
End Sub

Private Sub bttnLoad_Click(...) Handles bttnLoad.Click
    Try
        DS.ReadXmlSchema("DataSetSchema.xml")
        DS.ReadXml("DataSetData.xml", XmlReadMode.DiffGram)
    Catch ex As Exception
        MsgBox("Failed to load DataSet" & vbCrLf & ex.Message)
        Exit Sub
    End Try
    ShowDataSet()
End Sub
```

The Submit Edits button, finally, submits the changes to the database. The code attempts to submit all edited rows, but some of them may fail to update the database. The local DataSet doesn't enforce any check constraints, so when the application attempts to submit a product row with a negative price to the database, the database will reject the update operation. The DataSet rows that failed to update the underlying tables are shown in a message box like the one shown in Figure 22.3. You can review the values of the rows that failed to update the database and the description of the error returned by the database, and edit them further. The rows that failed to update the underlying table(s) in the database remain in the DataSet. Of course, you can always call the `RejectChanges` method for each row that failed to update the database, to undo the changes of the invalid rows. As is, the application doesn't reject any changes on its own. If you click the Show Edits button after an update operation, you will see the rows that failed to update the database, because they're marked as inserted/modified/deleted in the DataSet.

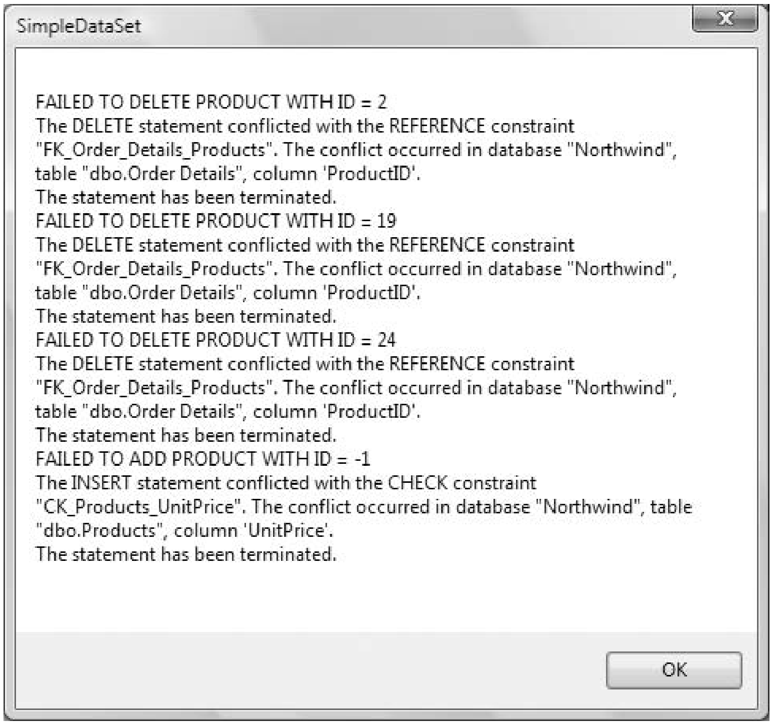


FIGURE 22.3
Viewing the rows that failed to update the database and the error message returned by the DBMS

```
SimpleDataSet

FAILED TO DELETE PRODUCT WITH ID = 2
The DELETE statement conflicted with the REFERENCE constraint
"FK_Order_Details_Products". The conflict occurred in database "Northwind",
table "dbo.Order Details", column 'ProductID'.
The statement has been terminated.
FAILED TO DELETE PRODUCT WITH ID = 19
The DELETE statement conflicted with the REFERENCE constraint
"FK_Order_Details_Products". The conflict occurred in database "Northwind",
table "dbo.Order Details", column 'ProductID'.
The statement has been terminated.
FAILED TO DELETE PRODUCT WITH ID = 24
The DELETE statement conflicted with the REFERENCE constraint
"FK_Order_Details_Products". The conflict occurred in database "Northwind",
table "dbo.Order Details", column 'ProductID'.
The statement has been terminated.
FAILED TO ADD PRODUCT WITH ID = -1
The INSERT statement conflicted with the CHECK constraint
"CK_Products_UnitPrice". The conflict occurred in database "Northwind", table
"dbo.Products", column 'UnitPrice'.
The statement has been terminated.

OK
```

Let's start with the code that loads the DataSet. When the form is loaded, the code initializes two DataAdapter objects, which load the rows of the Categories and Products tables. The names of the two DataAdapters are *DACategories* and *DAProducts*. They're initialized to the CN connection object and a simple SELECT statement, as shown in Listing 22.6.

LISTING 22.6: Setting Up the DataAdapters for the Categories and Products Tables

```
Private Sub Form1_Load(...) Handles MyBase.Load
    Dim CN As New SqlClient.SqlConnection( _
           "data source=localhost;initial catalog=northwind; " & _
           "Integrated Security=True")
    DACategories.SelectCommand = New SqlClient.SqlCommand( _
           "SELECT CategoryID, CategoryName, Description FROM Categories")
    DACategories.SelectCommand.Connection = CN
    Dim CategoriesCB As SqlCommandBuilder = New SqlCommandBuilder(DACategories)
    CategoriesCB.ConflictOption = ConflictOption.OverwriteChanges
    DAProducts.SelectCommand = New SqlClient.SqlCommand( _
           "SELECT ProductID, ProductName, " & _
           "CategoryID, UnitPrice, UnitsInStock, " & _
           "UnitsOnOrder FROM Products ")
    DAProducts.SelectCommand.Connection = CN
```

```
        DAProducts.ContinueUpdateOnError = True
        Dim ProductsCB As SqlCommandBuilder = New SqlCommandBuilder(DAProducts)
        ProductsCB.ConflictOption = ConflictOption.CompareAllSearchableValues
    End Sub
```

I've specified the SELECT statements in the constructors of the two DataAdapter objects and let the CommandBuilder objects generate the update statement. You can change the value of the `ConflictOption` property to experiment with the different styles of update statements that the CommandBuilder will generate. When the form is loaded, all the SQL statements generated for the DataAdapters are shown in the RichTextBox control. (The corresponding statements are not shown in the listing, but you can open the project in Visual Studio to examine the code.)

The Read Products and Related Tables button populates the DataSet and then displays the categories and products in the RichTextBox control by calling the `ShowDataSet()` subroutine, as shown in Listing 22.7.

LISTING 22.7: Populating and Displaying the DataSet

```
    Private Sub bttnCreateDataSet_Click(...) _
                 Handles bttnCreateDataSet.Click
        DS.Clear()
        DACategories.Fill(DS, "Categories")
        DAProducts.Fill(DS, "Products")
        DS.Relations.Clear()
        DS.Relations.Add(New Data.DataRelation("CategoriesProducts", _
             DS.Tables("Categories").Columns("CategoryID"), _
             DS.Tables("Products").Columns("CategoryID")))
        ShowDataSet()
    End Sub

    Private Sub ShowDataSet()
        RichTextBox1.Clear()
        Dim category As DataRow
        For Each category In DS.Tables("Categories").Rows
            RichTextBox1.AppendText( _
                 category.Item("CategoryName") & vbCrLf)
            Dim product As DataRow
            For Each product In category.GetChildRows( "CategoriesProducts")
                    RichTextBox1.AppendText( _
                        product.Item("ProductID") & vbTab & _
                        product.Item("ProductName") & vbTab)
                    If product.IsNull("UnitPrice") Then
                        RichTextBox1.AppendText(" *** " & vbCrLf)
                    Else
                        RichTextBox1.AppendText( _
                        Convert.ToDecimal( _
                        product.Item("UnitPrice")) _
                        .ToString("#.00") & vbCrLf)
                End If
            Next
        Next
    End Sub
```

After calling the `Fill` method to populate the two DataTables, the code sets up a DataRelation object to link the products to their categories through the *`CategoryID`* column and then displays the categories and the corresponding products under each category. Notice the statement that prints the products. Because the *`UnitPrice`* column may be Null, the code calls the `IsNull` method of the `product` variable to find out whether the current product's price is Null. If so, it doesn't attempt to call the `product.Item("UnitPrice")` expression, which would result in a runtime exception, and prints three asterisks in its place.

The Edit DataSet button modifies a few rows in the DataSet. Here's the statement that changes the name of a product selected at random (it appends the string `NEW` to the product's name):

```
DS.Tables("Products").Rows( _
     RND.Next(1, 78)).Item("ProductName") &= " - NEW"
```

The same button randomly deletes a product and sets the price of another row to a random value in the range from −10 to 40, and inserts a new row with a price in the same range. If you click the Edit DataSet button a few times, you'll very likely get a few invalid rows. The Show Edits button retrieves the edited rows of both tables and displays them. It uses the `DataRowState` property to discover the state of the row (whether it's new, modified, or deleted) and displays the row's ID and a couple of additional columns. Notice that you can retrieve the proposed and original versions of the edited rows (except for the deleted rows, which have no proposed version) and display the row's fields before and after the editing on a more elaborate interface. Listing 22.8 shows the code behind the Show Edits button.

LISTING 22.8: Viewing the Edited Rows

```
Private Sub bttnShow_Click(...)Handles bttnShow.Click
    Dim product As DataRow
    Dim msg As String = ""
    For Each product In DS.Tables("Products").Rows
        If product.RowState = DataRowState.Added Then
            msg &= "ADDED PRODUCT: " & _
               product.Item("ProductName") & vbTab & _
               product.Item("UnitPrice").ToString & vbCrLf
        End If
        If product.RowState = DataRowState.Modified Then
            msg &= "MODIFIED PRODUCT: " & _
               product.Item("ProductName") & vbTab & _
               product.Item("UnitPrice").ToString & vbCrLf
        End If
        If product.RowState = DataRowState.Deleted Then
```

```
                msg &= "DELETED PRODUCT: " & _
                   product.Item("ProductName", _
                   DataRowVersion.Original) & vbTab & _
                   product.Item("UnitPrice", _
                   DataRowVersion.Original).ToString & vbCrLf
            End If
        Next
        If msg.Length > 0 Then
            MsgBox(msg)
        Else
            MsgBox("There are no changes in the dataset")
        End If
    End Sub
```

I'm showing only the statements that print the edited rows of the Products DataTable in the listing. Notice that the code retrieves the proposed versions of the modified and added rows, but the original version of the deleted rows.

The Submit Edits button submits the changes to the two DataTables to the database by calling the `Update` method of the *`DAProducts`* DataAdapter and then the `Update` method of the *`DACategories`* DataAdapter. After that, it retrieves the rows in error with the `GetErrors` method and displays the error message returned by the DBMS with statements similar to the ones shown in Listing 22.5.

The Bottom Line

Create and populate DataSets. DataSets are data containers that reside at the client and are populated with database data. The DataSet is made up of DataTables, which correspond to database tables, and you can establish relationships between DataTables, just like relating tables in the database. DataTables, in turn, are made up of DataRow objects.

Master It How do we populate DataSets and then submit the changes made at the client back to the database?

Establish relations between tables in the DataSet. The DataSet can be thought of as a small database that resides at the client, because it's made up of tables and relationships between them. The relations in a DataSet are DataRelation objects, which are stored in the `Relations` property of the DataSet. Each relation is identified by a name, the two tables it relates, and the fields of the tables on which the relation is based.

Mater It How do we navigate through the related rows of two tables?

Submit changes in the DataSet back to the database. The DataSet maintains not only data at the client, but their states and versions too. It knows which rows were added, deleted, or modified (the `DataRowState` property) and it also knows the version of each row read from the database and the current version (the `DataRowVersion` property).

Master It How will you submit the changes made to a disconnected DataSet to the database?

Chapter 23

Building Data-Bound Applications

In the previous chapter, you learned about the two basic classes for interacting with databases: The Connection class provides the members you need to connect to the database, and the Command class provides the members you need to execute commands against the database. A data-driven application also needs to store data at the client, and you know how to use a DataReader to grab data from the database, and a DataAdapter to populate a DataSet at the client.

In addition to the DataSets we've created in the previous chapter, Visual Studio also allows you to create typed DataSets. A typed DataSet is designed with visual tools at design time and its structure is known to the compiler, which can generate very efficient code for the specific type of data you're manipulating in your application. Another advantage of typed DataSets is that they can be bound to Windows controls on a form. When a field is bound to a Windows control, every time you move to another row in the table, the control is updated to reflect the current value of the field. When the user edits the control on the form, the new value replaces the original value of the field in the DataSet. The form is said to be *bound* to the data source, which is usually a DataSet.

Data binding is a process for building data-driven applications with visual tools: You map the columns of a table to controls on the form, and Visual Studio generates the code for displaying the data on the bound controls, as well as updating the DataSet when the user edits the value of a bound control. These applications are called *data bound*, and they're similar to the applications you designed in the previous chapter. The difference is that Visual Studio generates the necessary code for you.

In this chapter, you'll learn how to do the following:

- ◆ Design and use typed DataSets
- ◆ Bind Windows forms to typed DataSets

Working with Typed DataSets

The DataSets you explored in the preceding chapter were untyped: Their exact structure was determined at runtime, when they were populated through the appropriate DataAdapters. In this chapter, I'll discuss in detail the typed DataSets. A *typed DataSet* is a DataSet with known structure, because it's created at design time with visual tools. The compiler knows the structure of the DataSet (that is, the DataTables it contains and the structure of each DataTable) and it generates code that's specific to the data at hand.

The most important characteristic of typed DataSets is that they allow you to write strongly typed code and practically eliminate the chances of exceptions due to syntax errors. Whereas in an untyped DataSet you had to access its DataTables by name with an expression such as `DS.Tables("Products")`, the equivalent typed DataSet exposes the name of the table as a property: `DS.Products`. To find out whether a specific field of an untyped DataSet is Null, you must use an expression like the following:

```
DS.Tables ("Products").Rows(0).Item("UnitPrice").IsNull
```

With a typed DataSet, you can declare a variable that represents a row of the Products table like this:

```
Dim prod As NorthwindDataSet.ProductsRow = _
             DS.Products.Rows(0)
```

You can then access the fields of the row as properties, with expressions like this: `prod.ProductName`, `prod.UnitPrice`, and so on. To find out whether the *`UnitPrice`* field is Null, call the method `prod.IsUnitPriceNull`. You can also set a field to Null with a method call: `prod.SetUnitPriceNull`. As you can guess, after the structure of the DataSet is known, the editor can generate a class with many members that will enormously simplify the coding of the application using the typed DataSet. The typed DataSet is a class that's generated by a wizard on-the-fly, and it becomes part of your solution.

Let's start by looking at the process of generating typed DataSets with visual data tools. Then you'll see how to bind Windows controls to typed DataSets and generate functional interfaces with point-and-click operations.

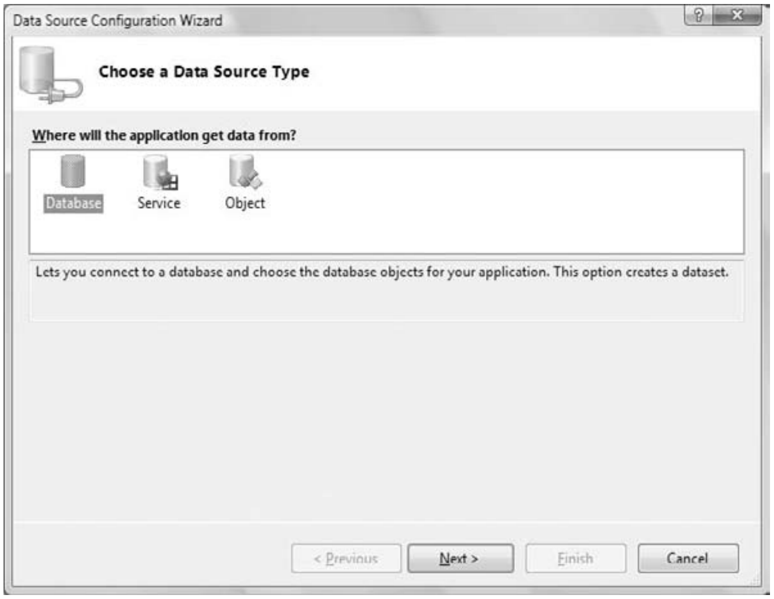
Generating a Typed DataSet

In this section, we'll create a typed DataSet with the three basic tables of the Northwind database: Products, Categories, and Suppliers. Create a new project, the DataSetOperations project. (This is the name of the sample project included with the chapter.) Then open the Data menu and choose the Add Data Source command. You will see the Data Source Configuration Wizard, which will take you through the steps of building a DataSet at design time. In the first dialog box of the wizard, you'll be asked to select the data source type, which can be a database, a service (such as a web service), or an object, as in Figure 23.1. Select the Database icon and click the Next button.

The Service option in the dialog box of Figure 23.1 will create a DataSet that retrieves its data from a service (usually a web service). The Object option allows you to create a DataSet from a collection of custom objects. You'll find an example of binding a form to a collection of custom objects in Chapter 24, ''Advanced DataSet Operations.''

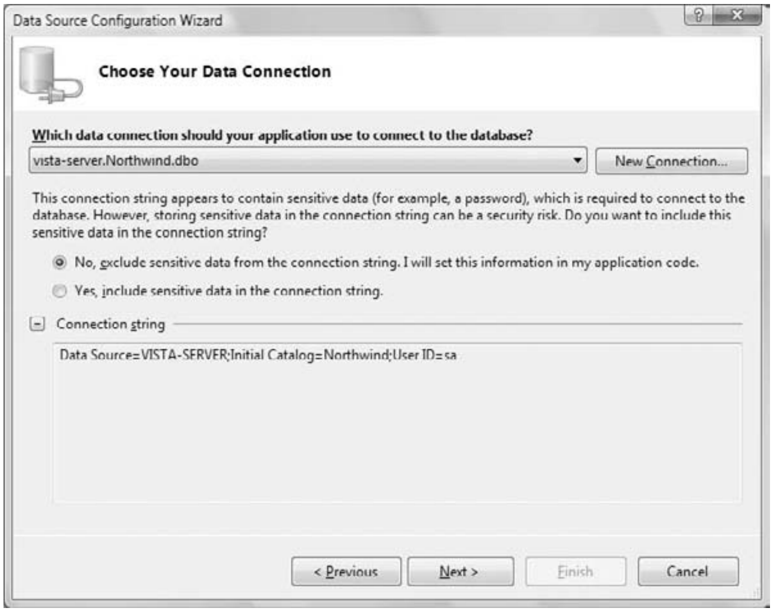
In the next dialog box of the wizard, the Choose Your Data Connection dialog box shown in Figure 23.2, you must specify a connection string for the database you want to use. Click the New Connection button to create a new connection only if there's no connection for the database you want to use. If you've experimented already with the visual tools of Visual Basic, you may already have an existing connection, in which case you simply select it from the drop-down list.

To create a new connection, you must specify your credentials: whether you'll connect with a username and password or use Windows authentication. Once connected to the server, you can select the desired database from a ComboBox control. If you click the New Connection button to create a new connection to the Northwind database, you will see the Add Connection dialog box, as shown in Figure 23.3. This dialog box is not new to you; it's the same dialog box you use to create a new data connection in the Server Explorer. If you haven't created a connection to the Northwind database yet, do it now. Otherwise, select the existing connection.


Data Source Configuration Wizard
Choose a Data Source Type
Where will the application get data from?
Database
Service
Object
< Previous
Next >
Finish
Cancel

Lets you connect to a database and choose the database objects for your application. This option creates a dataset.

FIGURE 23.1
Choosing the data source type in the Data Source Configuration Wizard


Data Source Configuration Wizard
Choose Your Data Connection
Which data connection should your application use to connect to the database?
New Connection...
Connection string
< Previous
Next >
Finish
Cancel

vista-server.Northwind.dbo

This connection string appears to contain sensitive data (for example, a password), which is required to connect to the database. However, storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

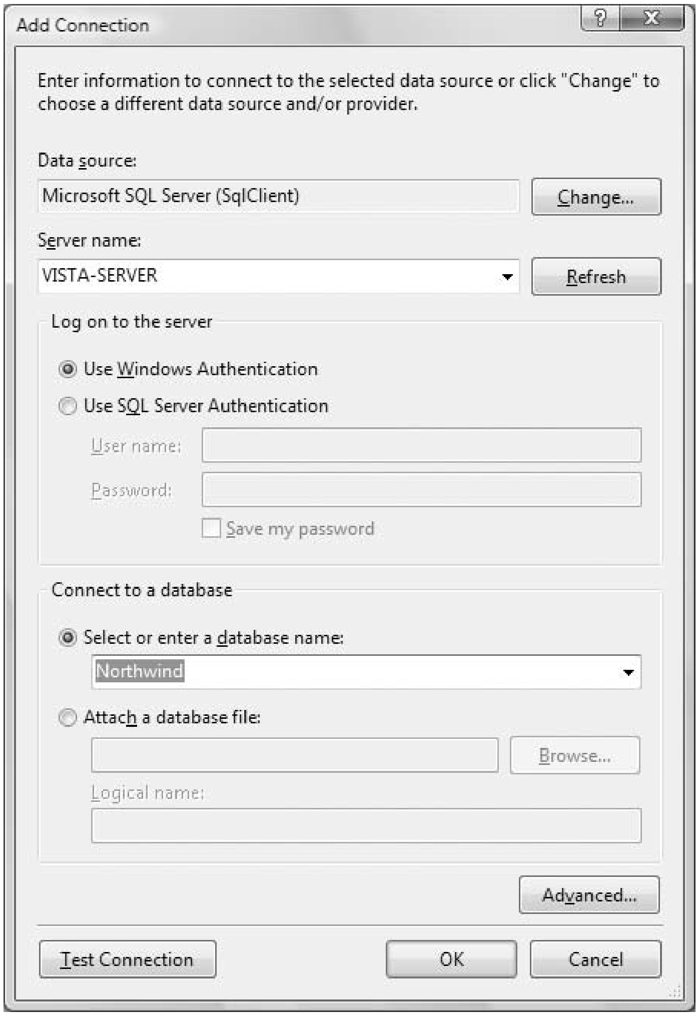
[✓ No, exclude sensitive data from the connection string. I will set this information in my application code.]

Yes, include sensitive data in the connection string.

```
Data Source=VISTA-SERVER;Initial Catalog=Northwind;User ID=sa
```

FIGURE 23.2
The Choose Your Data Connection dialog box of the Data Source Configuration Wizard

Figure 23.3
Use the Add Connection dialog box to specify a new connection to the Northwind database


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:
VISTA-SERVER

Log on to the server
(●) Use Windows Authentication
( ) Use SQL Server Authentication
User name:
Password:
[ ] Save my password

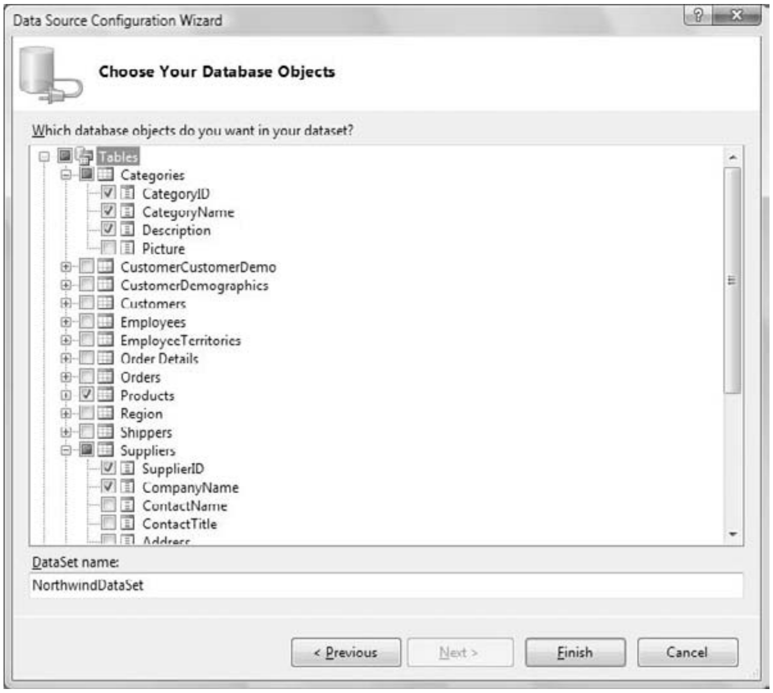
Connect to a database
(●) Select or enter a database name:
Northwind
( ) Attach a database file:
Logical name:

It's recommended that you use Windows authentication to connect to the database. If this isn't possible, because the database server is running on a remote computer, you must specify a username and password in the boxes shown in Figure 23.3. In this case, the wizard will ask whether you want to store sensitive information (the account's password) to the connection string. You can choose to either include the password in the connection string (not a very safe approach) or supply it from within your code. You can always set the `Password` property of a Connection object in your code. To secure the password, you can prompt the user for a password when the application starts and save it to an application variable. The password isn't stored anywhere. Alternatively, you can store an encrypted version of the password and decrypt and use it from within your code. The best approach for a local network is to use Windows authentication.

Click the OK button to close the Add Connection dialog box and then click Next again and you will see a dialog box with the default connection name: NorthwindConnectionString. This is the name of a new application setting that will store the connection information. You can edit this, as well as the other application settings, in the Settings tab of the project's Properties pages. (To see the project's Properties pages, choose Project Properties from the Project menu.)

Click Next again and you will see the Choose Your Database Objects dialog box, shown in Figure 23.4, where you can select the tables and columns you want to load to the DataSet. Notice that you can't use a `SELECT` statement to select the data you want from a table: You must select the entire table. Of course, you can write a stored procedure to limit your selection and then select it in the dialog box. If you select multiple tables and they're related, the relationship between them will also be imported for you (no need to create a DataRelation object for each relationship between the DataSet's tables). For this example, select the Categories, Suppliers, and Products tables of the Northwind database. Select all columns of the Products and Categories tables, except for the `Picture` column of the Categories table. From the Suppliers table, select the `SupplierID` and `CompanyName` columns.

FIGURE 23.4
Selecting the tables and columns you want to include in your typed DataSet



At the bottom of the dialog box of Figure 23.4, you can specify the name of the DataSet that will be generated for you. I'll use the default name, NorthwindDataSet. Click Finish to close the wizard and create the DataSet, which will be added automatically to the Solution Explorer window.

In the Data Sources window, shown in Figure 23.5, you will see a tree that represents the tables in the DataSet. The DataSet contains three DataTables, and each DataTable is made of the columns you selected in the wizard. This DataSet is typed, because it knows the structure of the data we're going to store in it. The interesting part of this tree is that it contains the Products table twice: in the first level along with the Categories table, and once again under the Categories and Suppliers table. (You must expand either table to see it, as in Figure 23.5.) Whereas the Products table on the first level represents the entire table, the nested ones represent the products linked to their categories and their suppliers respectively. You will see later in the chapter how to use the multiple Products DataTables.

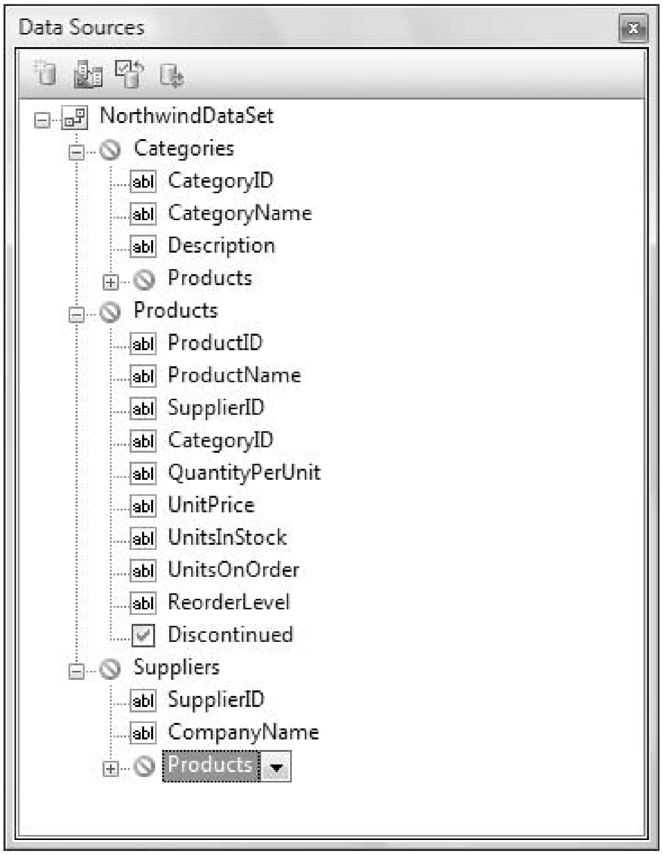


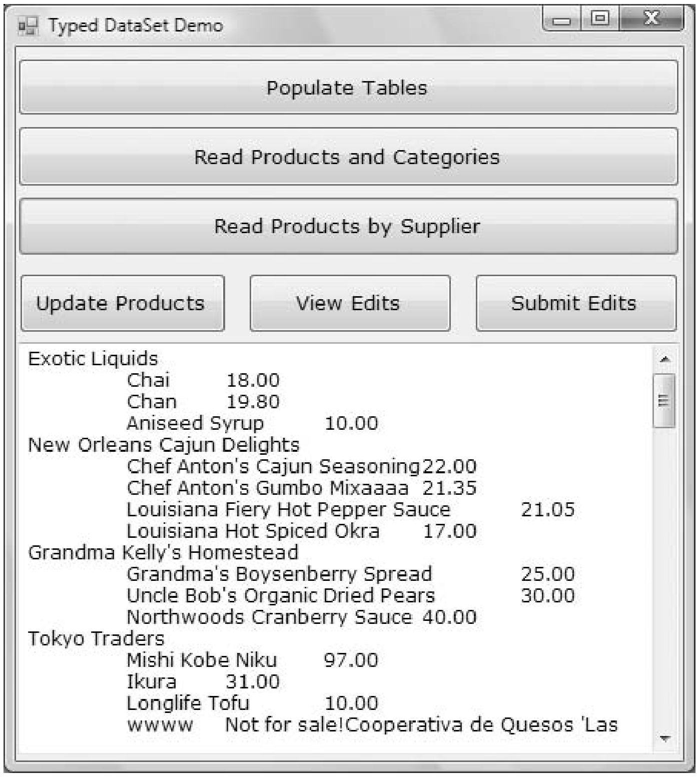
FIGURE 23.5
The Northwind DataSet created by the Data Source Configuration Wizard

The typed DataSet is actually a class, which is generated on-the-fly. It's no longer a generic DataSet we will populate at runtime with any table we wish through a DataAdapter; it's a specific object that can be populated only with the tables we specified in its design. If you want to see the code of the class generated by the wizard, click the Show All Files button in the Solution Explorer and double-click the `NorthwindDataSet.Designer.vb` item under the Northwind DataSet. You shouldn't edit the code, because if you decide to edit the DataSet (you'll see how you can edit it with visual tools), the wizard will create a new class and your changes will be lost. If you want to add some custom members to the Northwind class, create a new Partial class with the custom members and name it `NorthwindDataSet.vb`.

Exploring the Typed DataSet

Let's exercise the members of the typed DataSet a little and see how it differs from the equivalent untyped DataSet. The operations we'll perform are similar to the ones we performed in the preceding chapter with an untyped DataSet; you should focus on the different syntax. The code shown in this section belongs to the DataSetOperations sample project. This project contains three forms, and you will have to change the project's Startup object to view each one.

Figure 23.6 shows `Form1` of the project, which demonstrates the basic operations on a typed DataSet: how to populate it, edit some of its tables, and submit the changes back to the database. They're basically the same operations you'd perform with an untyped DataSet, but you will see that it's much simpler to work with typed DataSets.



```
Exotic Liquids
          Chai      18.00
          Chan      19.80
          Aniseed Syrup       10.00
New Orleans Cajun Delights
          Chef Anton's Cajun Seasoning22.00
          Chef Anton's Gumbo Mixaaaa  21.35
          Louisiana Fiery Hot Pepper Sauce        21.05
          Louisiana Hot Spiced Okra      17.00
Grandma Kelly's Homestead
          Grandma's Boysenberry Spread            25.00
          Uncle Bob's Organic Dried Pears         30.00
          Northwoods Cranberry Sauce  40.00
Tokyo Traders
          Mishi Kobe Niku      97.00
          Ikura      31.00
          Longlife Tofu        10.00
          wwww     Not for sale!Cooperativa de Quesos 'Las
```

FIGURE 23.6
`Form1` of the DataSetOperations project demonstrates the basic operations on a typed DataSet

To populate the three DataTables of the DataSet, we need three DataAdapter objects. Instead of the generic DataAdapter, the class generated by the wizard has created a TableAdapter class for each DataTable: the CategoriesTableAdapter, SuppliersTableAdapter, and ProductsTableAdapter classes. Declare three objects of the corresponding type at the form's level:

```
Dim CategoriesTA As New _
  NorthwindDataSetTableAdapters.CategoriesTableAdapter
Dim SuppliersTA As New _
  NorthwindDataSetTableAdapters.SuppliersTableAdapter
Dim ProductsTA As New _
  NorthwindDataSetTableAdapters.ProductsTableAdapter
```

The classes create the three objects that will retrieve the data from the database and submit the edited rows back to the database from the `SELECT` statements you specified with point-and-click operations. These objects derive from the TableAdapter class, which in turn is based on the DataAdapter class. If you examine the code of the Northwind class, you will find the code that creates the SQL statements for querying and updating the three tables and how these statements are used to create a DataAdapter object. The code is similar to the code we used in the preceding chapter to create DataAdapters from within our code.

We must also create a DataSet object to store the data. This time, however, we can use a specific type that describes the structure of the data we plan to store at the client, and not a generic DataSet. Insert the following declaration at the form's level:

```
Dim DS As New NorthwindDataSet
```

Now place the Populate Tables button on the form and insert the code shown in Listing 23.1 in its `Click` event handler.

Listing 23.1: Populating a Typed DataSet with the Proper TableAdapters

```
Private Sub bttnPopulate_Click(...) Handles bttnPopulate.Click
    Dim categories As Integer = CategoriesTA.Fill(DS.Categories)
    Dim suppliers As Integer = SuppliersTA.Fill(DS.Suppliers)
    Dim products As Integer = ProductsTA.Fill(DS.Products)
```

As you can see, the `Fill` method doesn't accept any DataTable as an argument; instead, the type of its argument is determined by the TableAdapter object to which it's applied. The `Fill` method of the *`ProductsTA`* TableAdapter accepts as an argument an object of the ProductsDataTable type. The event handler of the sample project includes a few more statements that print the count of rows in each of the three tables.

To go through the rows of the Products table, write a simple loop like the following:

```
Dim prod As NorthwindDataSet.ProductsRow
For Each prod In DS.Products.Rows
    TextBox1.AppendText(prod.ProductName & vbTab & _
         prod.UnitPrice.ToString("#,###.00") & vbCrLf)
Next
```

As you can see, the names of the fields are properties of the ProductsRow class. Some products may have no price (a Null value in the database). If you attempt to access the `UnitPrice` property of the ProductsRow class, a `NullReferenceException` exception will be thrown. To prevent it, you can make sure that the field is not Null from within your code, with an `If` structure like the following:

```
If prod.IsUnitPriceNull Then
    TextBox1.AppendText("Not for sale!")
Else
    TextBox1.AppendText(prod.UnitPrice.ToString("#,###.00"))
End If
```

To read data from linked tables in a hierarchical way, you don't have to specify the relationship between the tables as you did with untyped DataSets, because the typed DataTables expose the appropriate methods.

Now place another button on your form, the Read Products By Supplier button, and in its `Click` handler insert the code shown in Listing 23.2 to iterate through suppliers and related products. Notice that the SuppliersRow class exposes the `GetProductsRows` method, which retrieves the Products rows that are associated with the current supplier. The `GetProductsRows` method is equivalent to the `GetChildRows` of an untyped DataSet, only with the latter you have to supply a relationship name as an argument. Moreover, the `GetProductsRows` method returns an array of ProductsRow objects, not generic DataRow objects.

LISTING 23.2: Iterating through Linked DataTables

```
Private Sub bttnSuppliersProducts_Click(...) _
            Handles bttnSuppliersProducts.Click
    TextBox1.Clear()
    Dim supp As NorthwindDataSet.SuppliersRow
    For Each supp In DS.Suppliers.Rows
        TextBox1.AppendText(supp.CompanyName & vbCrLf)
        Dim prod As NorthwindDataSet.ProductsRow
        For Each prod In supp.GetProductsRows
            TextBox1.AppendText(vbTab & _
                  prod.ProductName & vbTab & _
                  prod.UnitPrice.ToString("#,###.00") & vbCrLf)
        Next
    Next
End Sub
```

The ProductsRow object exposes the `SuppliersRow` and `CategoriesRow` methods, which return the current product's parent rows in the Suppliers and Categories DataTables.

The most useful method of the typed DataTable is the `FindByID` method, which locates a row by its ID in the DataTable. To locate a product by its ID, call the `FindByProductID` method passing a product ID as argument. The method returns a ProductsRow object that represents the matching product. The method's return value is not a copy of the found row, but a reference to the actual row in the DataTable, and you can edit it. The code behind the Update Products button, which is shown in Listing 23.3, selects a product at random by its ID and prompts the user for a new price. Then it sets the `UnitPrice` field to the user-supplied value.

LISTING 23.3: Updating a Row of a Typed DataTable

```
Private Sub bttnUpdate_Click(...) Handles bttnUpdate.Click
    Dim selProduct As NorthwindDataSet.ProductsRow
    Dim RND As New System.Random
    selProduct = DS.Products.FindByProductID(RND.Next(1, 77))
    Dim newPrice As Decimal
```

```
        newPrice = Convert.ToDecimal(InputBox( _
                      "Enter product's new price", _
                       selProduct.ProductName, _
                       selProduct.UnitPrice.ToString))
        selProduct.UnitPrice = newPrice
    End Sub
```

As you can see, manipulating the rows of typed DataTables is much simpler than the equivalent operations with untyped DataSets, because the fields are exposed as properties of the appropriate class (the ProductsRow class for rows of the Products DataTable, the CategoriesRow class for rows of the Categories DataTable, and so on).

Let's look at the code for updating the database. The first step is to retrieve the edited rows with the `GetChanges` method, which returns a typed DataTable object, depending on the DataTable to which it was applied. To retrieve the modified rows of the Products DataTable, use the following statements:

```
Dim DT As NorthwindDataSet.ProductsDataTable
DT = DS.Products.GetChanges
```

You can pass an argument of the DataRowState type to the `GetChanges` method to retrieve the inserted, modified, or deleted rows. Because this is a typed DataSet, you can write a `For Each` loop to iterate through its rows (they're all of the ProductsRow type) and find out the edits. One feature you'd expect to find in a typed DataTable is a method for retrieving the original versions of a row by name. Unfortunately, the class generated by the wizard doesn't include such a method; you must use the `Item` property, passing as an argument the name of the row. A row's original field versions are given by the expression:

```
prod.Item("UnitPrice", DataRowVersion.Original)
```

To submit the edited rows to the database, you can call the appropriate TableAdapter's `Update` method. The code behind the Submit Edits button does exactly that, and it's shown in Listing 23.4.

LISTING 23.4: Submitting the Edited Rows of a Typed DataTable to the Database

```
Private Sub bttnSubmit_Click(...) Handles bttnSubmit.Click
    If DS.HasChanges Then
        Dim DT As NorthwindDataSet.ProductsDataTable
        DT = DS.Products.GetChanges
        If DT IsNot Nothing Then
            Try
                ProductsTA.Update(DT)
            Catch ex As Exception
                MsgBox(ex.Message)
                Exit Sub
            End Try
            MsgBox(DT.Rows.Count.ToString & _
                   " rows updated successfully.")
        End If
    End If
End Sub
```

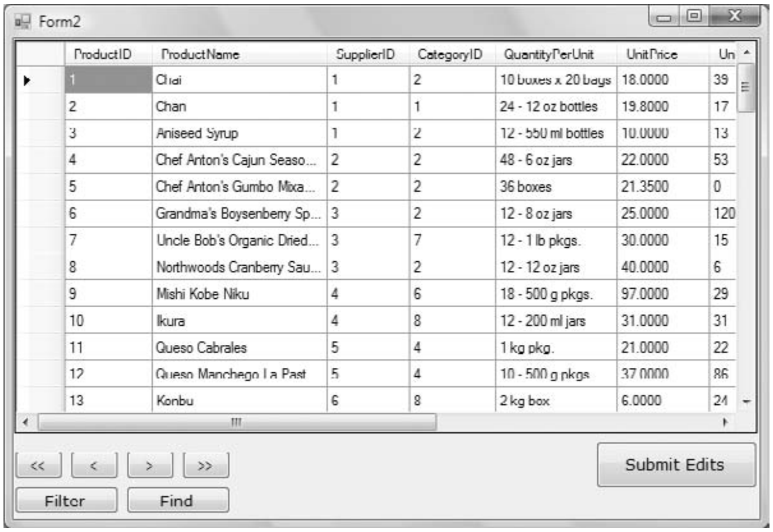
Typed DataSets are quite convenient when it comes to coding. The real advantage of typed DataSets is that they can simplify enormously the generation of data-bound forms, which is the main topic of this chapter.

Data Binding

Data binding is the process of linking the contents of a field to a control on the form. Every time the application modifies the field's value, the control is updated automatically. Likewise, every time the user edits the control's value on the form, the underlying field in the DataSet is also updated. The DataSet keeps track of the changes (the modified, added, and deleted rows), regardless of how they were changed. In short, data binding relieves you from having to map field values to controls on the form when a row is selected and moving values from the controls back to the DataSet when a row is edited.

In addition to binding simple controls such as TextBox controls to a single field, you can bind an entire column of a DataTable to a list control, such as the ListBox or the ComboBox control. And of course, you can bind an entire DataTable to a special control, the DataGridView control. You can build a data-browsing and data-editing application by binding the Products DataTable to a DataGridView control without a single line of code.

To explore the basics of data binding, add a second form to the project and make it the project's startup object. The new form of the DataSetOperations, `Form2`, is shown in Figure 23.7.

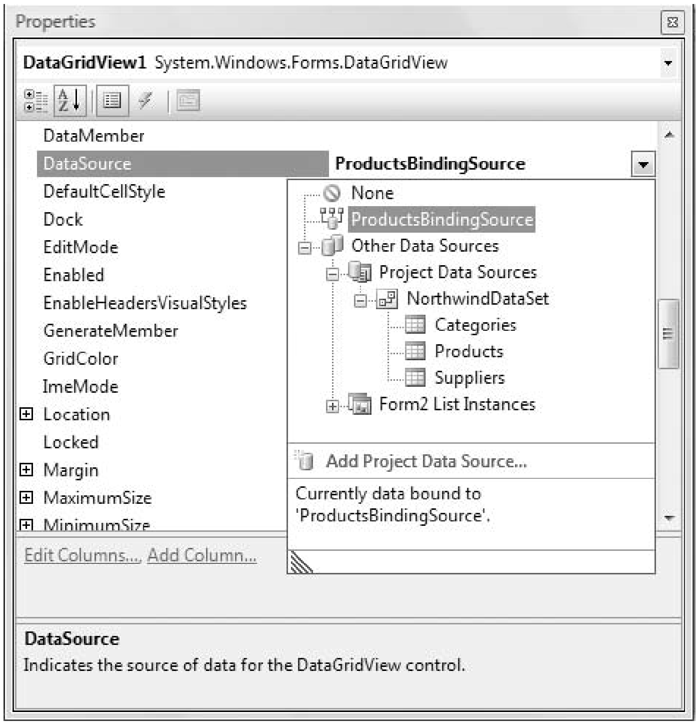


ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	Un
1	Chai	1	2	10 boxes x 20 bags	18.0000	39
2	Chan	1	1	24 - 12 oz bottles	19.8000	17
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10.0000	13
4	Chef Anton's Cajun Seaso...	2	2	48 - 6 oz jars	22.0000	53
5	Chef Anton's Gumbo Mixa...	2	2	36 boxes	21.3500	0
6	Grandma's Boysenberry Sp...	3	2	12 - 8 oz jars	25.0000	120
7	Uncle Bob's Organic Dried...	3	7	12 - 1 lb pkgs.	30.0000	15
8	Northwoods Cranberry Sau...	3	2	12 - 12 oz jars	40.0000	6
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97.0000	29
10	Ikura	4	8	12 - 200 ml jars	31.0000	31
11	Queso Cabrales	5	4	1 kg pkg.	21.0000	22
12	Queso Manchego La Past	5	4	10 - 500 g pkgs.	37.0000	86
13	Konbu	6	8	2 kg box	6.0000	24

FIGURE 23.7
Viewing an entire DataTable on a data-bound DataGridView control

Then drop a DataGridView control on the form and set the control's `DataSource` property to bind it to the Products DataTable. Select the DataGridView control on the form and locate its

`DataSource` property in the Properties window. Expand the `DataSource` property and you will see the project's data sources. The form contains no data source for the time being, so all data sources are listed under Other Data Sources. Expand this item of the tree to see the Project Data Sources item, which in turn contains the NorthwindDataSet data source (or whatever you have named the typed DataSet). Expand this item and you will see the names of the DataTables in the DataSet, as shown in Figure 23.8. Select the Products DataTable.



Currently data bound to 'ProductsBindingSource'.

DataSource
Indicates the source of data for the DataGridView control.

FIGURE 23.8
Binding the DataGridView control to a DataTable

The editor will populate the DataGridView control with the table's columns: it will map each column in the Products DataTable to a column in the DataGridView control. All columns have the same width and are displayed as text boxes, except for the `Discontinued` column, which is mapped to a CheckBox control. (This is the last column of the controls, and you will see it at runtime, because you can't scroll the control at design time.) The control's columns were named after the DataTable's columns, but we'll change the appearance of the grid shortly. Press F5 to run the application, and the form will come up populated with the Products rows! Obviously, the editor has generated some code for us to populate the control. The code generated by the editor is a single statement in the form's `Load` event handler:

```
Me.ProductsTableAdapter.Fill(Me.NorthwindDataSet.Products)
```

As far as browsing the data, we're all set. All we have to do is adjust the appearance of the DataGridView control with point-and-click operations. You can also edit the rows, but there's no code to submit the edits to the database. Submitting the changes to the database shouldn't be a problem for you; just copy the corresponding code statement from `Form1` of the project. Place the Submit Edits button on the form, and in its `Click` handler insert the following statements:

```
If NorthwindDataSet.HasChanges Then
    Dim DT As NorthwindDataSet.ProductsDataTable
    DT = NorthwindDataSet.Products.GetChanges
    If DT IsNot Nothing Then
        Try
            ProductsTableAdapter.Update(DT)
        Catch ex As Exception
            MsgBox(ex.Message)
            Exit Sub
        End Try
        MsgBox(DT.Rows.Count.ToString & _
              " rows updated successfully.")
    End If
End If
```

I changed the name of the DataSet from *`DS`* to *`NorthwindDataSet`* and the name of the TableAdapter from *`ProductsTA`* to *`ProductsTableAdapter.`* And where did these names come from? If you switch to the form's Designer, you'll see that while you were setting the DataGridView control's properties, three items were added to the Components tray of the form: the NorthwindDataSet component (which is the typed DataSet), the ProductsTableAdapter (which is responsible for populating the control and submitting the edited rows to the database), and the ProductsBindingSource (which is the liaison between the DataGridView control and the DataSet). The ProductsBindingSource is basically a data source, and it's discussed in the following section.

Using the BindingSource Class

To understand the functionality of the BindingSource class, look up its members. Enter its name and the following period in the code window and you will see a list of members. The `Position` property reads (or sets) the current item's index in the underlying DataTable. The DataGridView control doesn't maintain the order of the rows in the underlying table; besides, you can sort the DataGridView control's rows in any way you like, but the DataTable's rows won't be sorted. Use the `Position` property to find out the index of the selected row in the DataTable. The `MoveFirst`, `MovePrevious`, `MoveNext`, and `MoveLast` methods are simple navigational tools provided by the BindingSource class. You can place four buttons on the form and insert a call to these methods to move to the first, previous, next, and last rows, respectively. The four navigational buttons at the lower-left corner of the form shown in Figure 23.7 call these methods to select another row on the grid.

The two most interesting members of the BindingSource class are the `Find` method and the `Filter` property. The `Filter` property is set to an expression similar to the `WHERE` clause of an SQL statement to filter the data on the grid. Place a new button on the form, set its caption to **Filter** and its name to **`bttnFilter`**, and insert the following statements in its `Click` even handler to filter the rows of the grid with their product name:

```
Private Sub bttnFilter_Click(...) Handles bttnFilter.Click
    Dim filter As String
    filter = InputBox("Enter product name, or part of it")
```

```
        ProductsBindingSource.Filter = _
             "ProductName LIKE '%" & filter.Trim & "%'"
    End Sub
```

Run the application, and click the Filter button to limit the rows displayed on the grid by their product name. If you're searching for products that contain the string `sauce` in their name, the `Filter` property limits the selection as if you had requested products with the following `WHERE` clause (the percent sign is an SQL wildcard that matches any string):

```
WHERE ProductName LIKE '%sauce%'
```

To restore the original selection, set the filter expression to a blank string. You can design an auxiliary form on which users can enter multiple criteria and filter products by their price or stock, their supplier, and so on. With a bit of programming effort, you can apply multiple criteria, such as products of a specific category that are on order, out-of-stock items from a specific supplier, and so on.

The `Find` method searches a value in a specific column. Both the column name and search argument are specified as arguments to the method, and the return value is the row's position in the DataTable. To select the row, set the BindingSource object's `Position` property to the value returned by the `Find` method. The code behind the Find button in the sample project is the following:

```
Dim search As String
search = InputBox("Enter product name, or part of it")
Dim idx As Integer = _
      ProductsBindingSource.Find("ProductName", search)
ProductsBindingSource.Position = idx
```

The `Find` method is not the most convenient search tool, because you have to specify the exact value of the field you're looking for. To retrieve the current row in the DataTable mapped to the BindingSource, use the `Current` property; to determine the number of rows in the same DataTable, read the value of the `Count` property. The `Current` property returns an object, which you must cast to the DataRowView type and call its `Row` property:

```
CType(ProductsBindingSource.Current, DataRowView).Row
```

This expression returns a DataRow object, which you can cast to a ProductsRow type. You will see examples of using the `Current` property of the BindingSource class to access the underlying row in the DataTable later in this chapter.

Handling Identity Columns

If you attempt to add a row to the DataGridView control, the new row's ID will be –1 (or another negative value if you have added multiple rows). This is a valid value for an Identity column, as long as its `AutoIncrement` property is set to –1. But the `ProductID` column in the database has an `AutoIncrement` value of 1 — why is it different in the DataSet? When the editor created the DataSet, it changed this setting to avoid conflicts during the updates. If new products were assigned valid IDs (positive values following the last ID in the DataSet) at the client, consider what might happen when the edits were submitted to the database. The IDs provided by the DataSet might be taken in the database, and the Insert operation would fail. To avoid this conflict, the DataSet uses negative identity values. When these rows are submitted to the database, they're assigned a new ID by the database, which is a positive value.

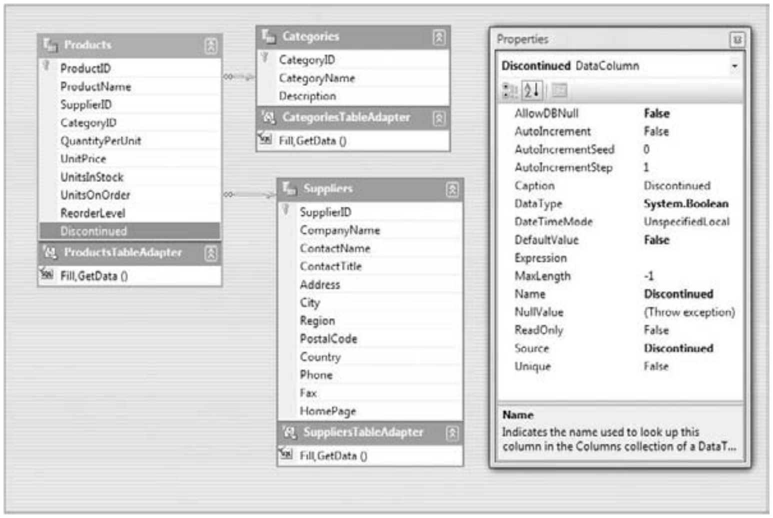
However, a problem remains. The new ID isn't transferred back to the client, and the DataSet displays negative IDs. One solution is to populate the DataSet again; however, there's a lot more to learn about submitting edited rows to the database, and we'll return to this topic in the section ''Binding Hierarchical Tables,'' later in this chapter.

You can experiment with this form of the DataSetOperations project by editing the products, adding new ones, and deleting rows. If you attempt to add a new row, you'll get an error message indicating that the `Discontinued` column doesn't accept Nulls. The default value of the check box on the DataGrid control is neither True nor False (it's Null), and we must validate its value. The simplest solution to the problem is to apply a default value to the `Discontinued` column, and the following section describes how to edit the properties of the DataSet.

Adjusting the DataSet

To adjust the properties of the DataSet, right-click the DataSet in the Data Sources window and choose Edit DataSet With Designer from the context menu. The DataSet's Designer window will appear, shown in Figure 23.9.

Figure 23.9
Editing the DataSet with visual tools



Discontinued DataColumn

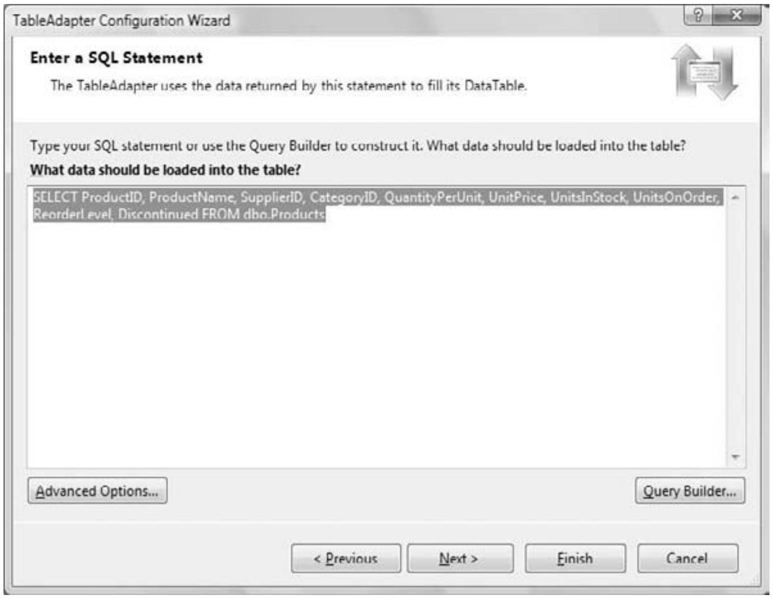
AllowDBNull	False
AutoIncrement	False
AutoIncrementSeed	0
AutoIncrementStep	1
Caption	Discontinued
DataType	System.Boolean
DateTimeMode	UnspecifiedLocal
DefaultValue	False
Expression	
MaxLength	-1
Name	Discontinued
NullValue	(Throw exception)
ReadOnly	False
Source	Discontinued
Unique	False

Name
Indicates the name used to look up this column in the Columns collection of a DataT...

Right-click the header of the `Discontinued` column in the Products table and choose Properties to see the properties of the DataColumn. One of them is the `DefaultValue` property, which is set by default to Null. Change it to 0 or False to impose a default value for this column. On this designer, you can examine the data types of the columns of each table, drop or create new relations between tables, and set other interesting properties, such as the `Caption` property of a column, which will be used to name the column of the bound DataGridView control, or the `NullValue` property, which determines how the DataSet will handle Null values. The default value of the `NullValue` property is Throw Exception. Every time the application requests the value of a Null field, a runtime exception is thrown. You can set it to Empty (in which case an empty string is returned) or Nothing (in which case a Nothing value is returned). You can also set the autoincrement values of Identity columns here. If you select the `ProductID` column in the Products table, you'll see that the wizard has set the column's `AutoIncrementSeed` and `AutoIncrementStep` to −1, for the reasons explained already.

While you're in the DataSet Designer, right-click a DataTable and choose Configure. This starts the TableAdapter Configuration Wizard. The first dialog box of the wizard, shown in Figure 23.10, shows the SQL statement that the Data Source Configuration Wizard generated while you were selecting the columns you wanted to include in the DataSet. You can edit this statement by adding more columns, a `WHERE` clause to limit the number of rows to be selected, and an `ORDER BY` clause. To edit the `SELECT` statement, you modify it right in the dialog box of Figure 23.10, or click the Query Builder button to view the Query Builder dialog box that lets you specify complicated queries with visual tools.

FIGURE 23.10
Editing the SELECT statement that populates the Products DataTable with the TableAdapter Configuration Wizard



Enter a SQL Statement

The TableAdapter uses the data returned by this statement to fill its DataTable.

Type your SQL statement or use the Query Builder to construct it. What data should be loaded into the table?

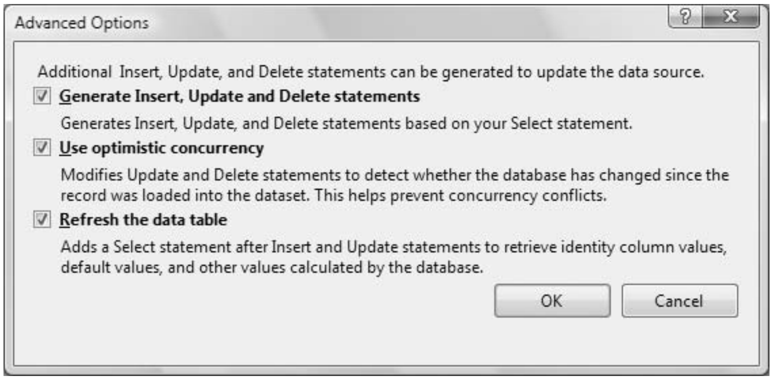
What data should be loaded into the table?

```
SELECT ProductID, ProductName, SupplierID, CategoryID, QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
ReorderLevel, Discontinued FROM dbo.Products
```

If you click the Advanced Options button, you will see the Advanced Options dialog box, shown in Figure 23.11. Here you must specify which statements should be generated by the wizard. If you're developing a browser application, deselect the first check box: Generate Insert, Update, and Delete Statements. If you clear this option, the other two options will be disabled.

The Use Optimistic Concurrency option affects the `UPDATE` and `DELETE` statements generated by the wizard. If this check box is selected, the two statements will not update or delete a row if it has been edited by another user since it was read. The wizard will generate two long statements that take into consideration the values read from the database into the DataSet at the client (the row's original values), and if any of the row's columns in the database are different from the original version of the same row in the DataSet, it won't update or delete the row. By using optimistic concurrency, you're assuming that it's rather unlikely that two users will update the same row at the same time. If the row being updated has already been modified by another user, the update operation fails. If you clear this option, the `UPDATE/DELETE` statements take into consideration the row's primary key and are executed, even if the row has been modified since it was read. In effect, the last user to update a row overwrites the changes made by other users.

FIGURE 23.11
The Advanced Options dialog box lets you know how the TableAdapter will submit the updates to the underlying tables


Advanced Options

Additional Insert, Update, and Delete statements can be generated to update the data source.

☑ Generate Insert, Update and Delete statements

Generates Insert, Update, and Delete statements based on your Select statement.

☑ Use optimistic concurrency

Modifies Update and Delete statements to detect whether the database has changed since the record was loaded into the dataset. This helps prevent concurrency conflicts.

☑ Refresh the data table

Adds a Select statement after Insert and Update statements to retrieve identity column values, default values, and other values calculated by the database.

The last option in the Advanced Options dialog box specifies whether the TableAdapter reads back the inserted/updated rows. You should leave this check box selected, so that the identity values assigned by the database to new rows will be read back and update the DataSet.

Implementing Optimistic Concurrency

Curious about the statements that take into consideration the original values of the row being updated? Here's the DELETE statement for the Products row that uses optimistic concurrency:

```
DELETE FROM [dbo].[Products]
WHERE (([ProductID] = @Original_ProductID) AND
      ([ProductName] = @Original_ProductName) AND
      ((@IsNull_SupplierID = 1 AND
              [SupplierID] IS NULL) OR
             ([SupplierID] = @Original_SupplierID)) AND
      ((@IsNull_CategoryID = 1 AND
              [CategoryID] IS NULL) OR
             ([CategoryID] = @Original_CategoryID)) AND
      ((@IsNull_QuantityPerUnit = 1 AND
              [QuantityPerUnit] IS NULL) OR
             ([QuantityPerUnit] = @Original_QuantityPerUnit)) AND
      ((@IsNull_UnitPrice = 1 AND
              [UnitPrice] IS NULL) OR
             ([UnitPrice] = @Original_UnitPrice)) AND
      ((@IsNull_UnitsInStock = 1 AND
              [UnitsInStock] IS NULL) OR
             ([UnitsInStock] = @Original_UnitsInStock)) AND
      ((@IsNull_UnitsOnOrder = 1 AND
             [UnitsOnOrder] IS NULL) OR
            ([UnitsOnOrder] = @Original_UnitsOnOrder)) AND
```

```
        ((@IsNull_ReorderLevel = 1 AND
               [ReorderLevel] IS NULL) OR
              ([ReorderLevel] = @Original_ReorderLevel)) AND
        ([Discontinued] = @Original_Discontinued))
```

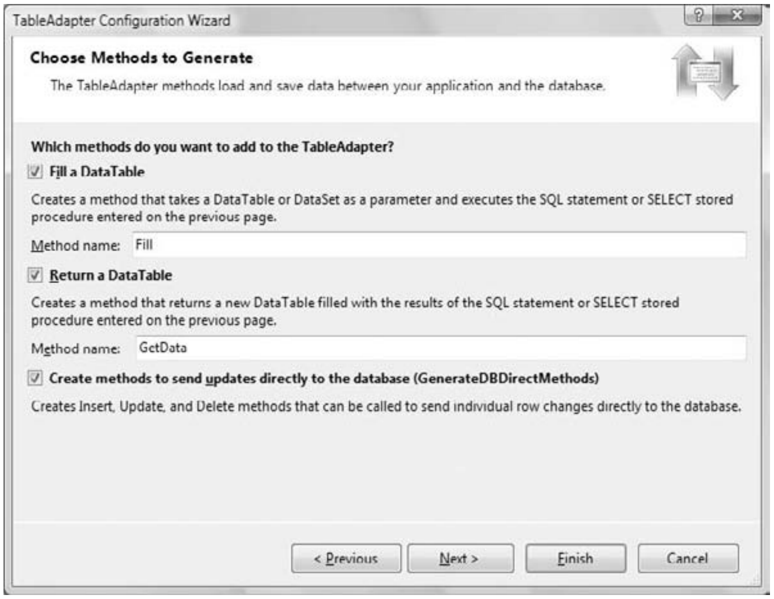
The same statement with the optimistic concurrency off is quite short:

```
DELETE FROM [dbo].[Products]
WHERE ([ProductID] = @Original_ProductID)
```

Examine the long statement to understand how optimistic concurrency is handled. The first term in the WHERE clause locates a single row based on the product ID, which is unique. This is the row that should be deleted (or updated by the UPDATE statement). However, the statement will not select the row if any of its fields have been edited since we read it from the database. If another user has changed the price of a specific product, the following term will evaluate to false and the WHERE clause will return no row to be deleted.

When you return to the configuration wizard, click Next and you will see the dialog box shown in Figure 23.12, where you can specify the methods that the wizard will generate for you. The `Fill` method populates a DataTable, and the `GetData` method returns a DataTable object with the same data; the last option in the dialog box specifies whether the DataSet will expose methods for inserting/updating/deleting rows directly against the database.

FIGURE 23.12
Selecting the methods to be generated by the TableAdapter Configuration Wizard


TableAdapter Configuration Wizard
Choose Methods to Generate
The TableAdapter methods load and save data between your application and the database.
< Previous
Next >
Finish
Cancel

Which methods do you want to add to the TableAdapter?

☑ Fill a DataTable

Creates a method that takes a DataTable or DataSet as a parameter and executes the SQL statement or SELECT stored procedure entered on the previous page.

Method name: Fill

☑ Return a DataTable

Creates a method that returns a new DataTable filled with the results of the SQL statement or SELECT stored procedure entered on the previous page.

Method name: GetData

☑ Create methods to send updates directly to the database (GenerateDBDirectMethods)

Creates Insert, Update, and Delete methods that can be called to send individual row changes directly to the database.

Click Next again, and the wizard will regenerate the NorthwindDataSet class, taking into consideration the options you specified in the steps of the wizard.

In the following section, we'll let the editor build simple data-driven applications for us. You're going to see how to bind other controls to typed DataSets and how to customize the DataGridView control.

Designing Data-Driven Interfaces the Easy Way

Instead of binding the DataGridView control through its properties, you can let Visual Studio perform the binding for you:

1. Add a third form to the DataSetOperations sample project, the `Form3` form, and make it the project's Startup object.
2. To display the rows of the Products table on a DataGridView control, open the Data Sources window and select the Products table. As soon as you select it, an arrow appears next to its name. Click this arrow to open a drop-down list with the binding options for the DataTable. The DataTable can be bound to the following:
    - ◆ A DataGridView control, which will display all rows and all columns of the table
    - ◆ A ListBox or ComboBox control, which will display a single column of all rows
    - ◆ A number of TextBox controls (the Details option), one for each column
3. Select the DataGridView option and then drop the Products DataTable on the form.

The editor will create a DataGridView control and bind it to the Products DataTable. In addition, it will create a toolbar at the top of the form with a few navigational and editing buttons, as shown in Figure 23.13 (shown at design time so you can see the components generated by the editor). Notice that the toolbar contains one button for deleting rows (the button with the X icon) and one button for submitting the edits to the database (the button with the disk icon). The Filter, Find, and Refresh Data buttons were not generated by the editor; I've added them to the toolbar and inserted the appropriate code in their `Click` event handlers. You've already seen the code that implements all three operations.

The designer will also generate the following components, which will appear in the Components tray:

NorthwindDataSet The typed DataSet for the data specified with the Data Source Configuration Wizard.

ProductBindingSource A BindingSource object for the Products table.

ProductsTableAdapter An enhanced DataAdapter that exposes the methods for reading data from the database and submitting the changes made at the client back to the database. The TableAdapter class differs from the DataAdapter class in that its `Fill` method accepts as an argument an object of the Products type, and not generic DataSet and DataTable objects. The methods of the TableAdapter object know how to handle rows of the specific type, and not any DataRow object.

TableAdapterManager This encapsulates the functionality of all TableAdapter objects on the form. If you drop additional tables on the form, the editor will create the corresponding TableAdapter for each one. The TableAdapterManager encapsulates the functionality of all individual TableAdapter objects and exposes the `UpdateAll` method, which submits the entire DataSet to the database. The `UpdateAll` method of the TableAdapterManager calls the `Update` method of each individual TableAdapter in the proper order.

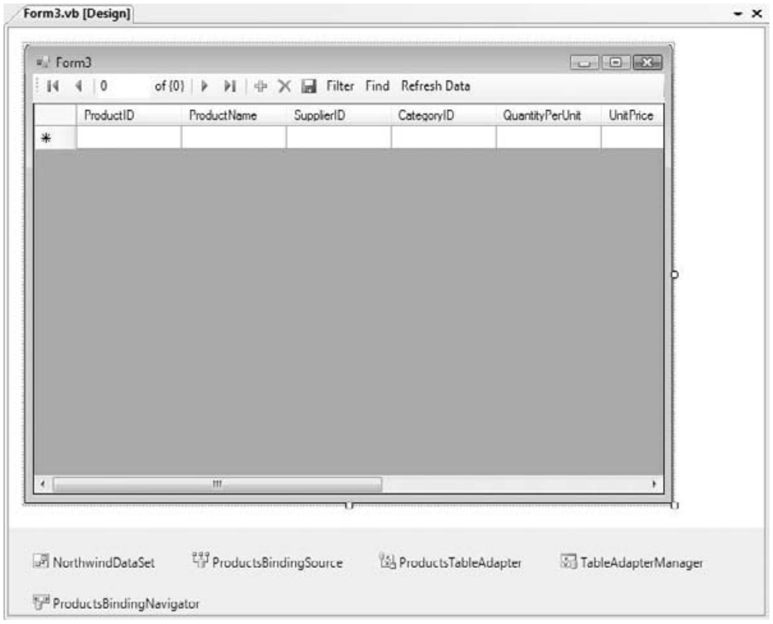


FIGURE 23.13
Binding a form to a DataTable

ProductsBindingNavigator This component represents the toolbar added to the form. The toolbar is a ToolStrip control with custom items and the appropriate code. The navigational tools generated by the editor are rather primitive, and you can remove them from the control. Just keep the code for the Save button, which you'll need if your application allows editing of the data.

As for the code generated by the editor, here it is:

```
Private Sub ProductsBindingNavigatorSaveItem_Click(...) _
            Handles ProductsBindingNavigatorSaveItem.Click
    Me.Validate()
    Me.ProductsBindingSource.EndEdit()
    Me.TableAdapterManager.UpdateAll(Me.DSProducts)
End Sub

Private Sub Form2_Load(...) Handles MyBase.Load
    Me.ProductsTableAdapter.Fill( Me.DSProducts.Products)
End Sub
```

In the form's `Load` event handler, the Products DataTable is filled with a call to the Products-TableAdapter class's `Fill` method. The other event handler corresponds to the `Click` event of the Save button on the toolbar, and it calls the TableAdapterManager class's `UpdateAll` method. This is all it takes to submit the changes made to the data at the client.

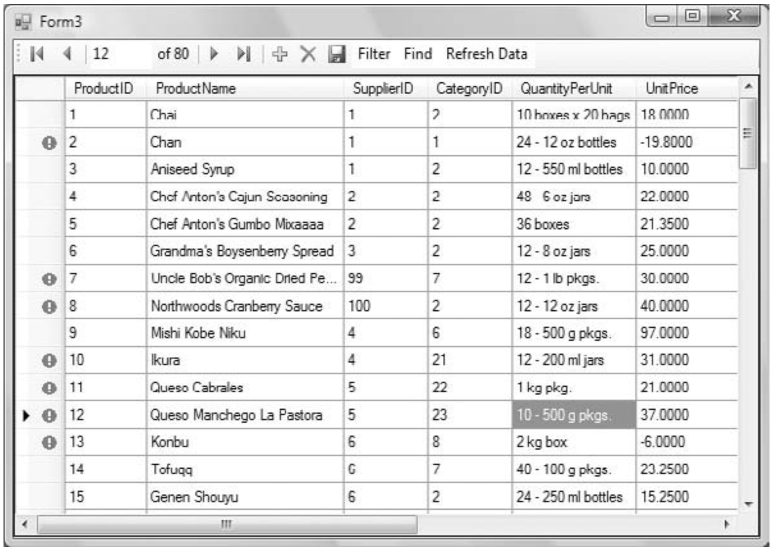
Let's see how far this autogenerated application will take us. Run the application and edit a few products. Change a few names, set a couple of prices to negative values, set a product's category to an invalid category ID (any value exceeding 7 is invalid, unless you have added new categories), add a couple of new products (they will be assigned negative IDs, as expected), and delete some products. As you can guess, you can't delete rows from the Products table, because they're all referenced by the Order Details table, but this table doesn't exist in the DataSet, so it's perfectly legal to delete products in the context of the DataSet. When the edits are submitted to the database, the deletions will be rejected, of course.

Let's see how the Save button on the toolbar handles the updates. Click the Save button on the toolbar and you will get an error message indicating that a row has violated a referential or check constraint, depending on the order in which the rows were submitted to the database. The `UpdateAll` method of the ProductsTableAdapter object will give up after the first failure.

As you recall, the DataAdapter class, on which the TableAdapter class is based, exposes the `ContinueUpdateOnError` property. Unfortunately, the TableAdapter class doesn't expose this property. However, you can access the underlying DataAdapter through the `Adapter` property and set the `ContinueUpdateOnError` property to True. Insert the following method in front of the statement that calls the `UpdateAll` method:

```
Me.TableAdapterManager.ProductsTableAdapter. _
          Adapter.ContinueUpdateOnError = True
```

Run the application again, edit the data on the grid, and submit the changes to the database. This time the application won't crash with an error message. Instead, the rows that failed to update the underlying table in the database will be marked with an exclamation mark icon in the row's header, as shown in Figure 23.14. We managed to submit all the rows to the database, regardless of whether they successfully updated the Products table, through the ProductsTableAdapter object. The `UpdateAll` method retrieved the error messages returned by the DBMS and displayed them on the control. To see the reason why each row failed to update the Products table, hover the pointer over the error icon and you will see the description of the error in a ToolTip box.



	ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice
	1	Chai	1	2	10 boxes x 20 bags	18.0000
❗	2	Chan	1	1	24 - 12 oz bottles	-19.8000
	3	Aniseed Syrup	1	2	12 - 550 ml bottles	10.0000
	4	Chef Anton's Cajun Seasoning	2	2	48 6 oz jars	22.0000
	5	Chef Anton's Gumbo Mixaaaa	2	2	36 boxes	21.3500
	6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars	25.0000
❗	7	Uncle Bob's Organic Dried Pe...	99	7	12 - 1 lb pkgs.	30.0000
❗	8	Northwoods Cranberry Sauce	100	2	12 - 12 oz jars	40.0000
	9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97.0000
❗	10	Ikura	4	21	12 - 200 ml jars	31.0000
❗	11	Queso Cabrales	5	22	1 kg pkg.	21.0000
▶ ❗	12	Queso Manchego La Pastora	5	23	10 - 500 g pkgs.	37.0000
❗	13	Konbu	6	8	2 kg box	-6.0000
	14	Tofuqq	6	7	40 - 100 g pkgs.	23.2500
	15	Genen Shouyu	6	2	24 - 250 ml bottles	15.2500

FIGURE 23.14
Viewing the update errors on the DataGridView control

You can also create a list of the rows that failed to update their underlying table along with the error message returned by the database. The code for iterating through a table's rows and examining the `RowError` property was presented in the preceding chapter. You can easily add an extra button on the toolbar and use it to display an auxiliary form with the update errors.

By the way, the error messages displayed on the DataGridView control are the ones returned by the DBMS (SQL Server in our case). If you want, you can set each row's `RowError` property to a different, more meaningful description.

Enhancing the Navigational Tools

The navigational tools on the BindingNavigator are quite primitive. Let's enhance the toolbar at the top of the form by adding two buttons, the Filter and Find buttons of the preceding section. Stop the application and open `Form3` in design mode.

To add a new element to the ToolBar control, expand the combo box that's displayed at design time after the existing elements. From the drop-down list, select the Button item to add a Button control to the toolbar. Select the newly added button and set its `DisplayStyle` property to Text. (Normally this property is set to Image, because the items on a toolbar are identified by an icon; you should find a few good icons and use them in your applications.) Set its `Text` property to **`Filter`** and its name to **`bttnFilter`**. Follow similar steps for the Find button as well. Then copy the code of the two buttons in `Form2` and paste it in the `Click` event handlers of the two ToolStrip buttons. You just added a filtering and search feature to your application.

The Find feature isn't very practical with product names, because users have to specify the full and exact product name. This feature should be used with fields such as IDs, book ISBNs, and email addresses. To find a product by name, most users would use the Filter button to limit the selection on the grid and then locate the desired product.

You can also use two TextBox controls in place of the two Button controls on the toolbar. If you'd rather allow users to enter their search and filter arguments on the toolbar, you must intercept the Enter keystroke from within the respective control's `KeyUp` event and call the same code to filter or search the rows of the grid.

Now add yet another button to the toolbar and set its caption to **Refresh Data**. This button will reload the data from the database by calling the `Fill` method of the TableAdapter. Before loading the data, however, we must make sure that the DataSet doesn't contain any changes by examining the `HasChanges` property. If it's True, we must prompt the user accordingly. Notice that if a row failed to update the database, the DataSet will contain changes, even though the edits were submitted to the database. Some of the changes can be undone, but not all of them. A deleted row, for example, is no longer visible on the control, and users can't restore it. The code behind the Refresh Data button is shown in Listing 23.5.

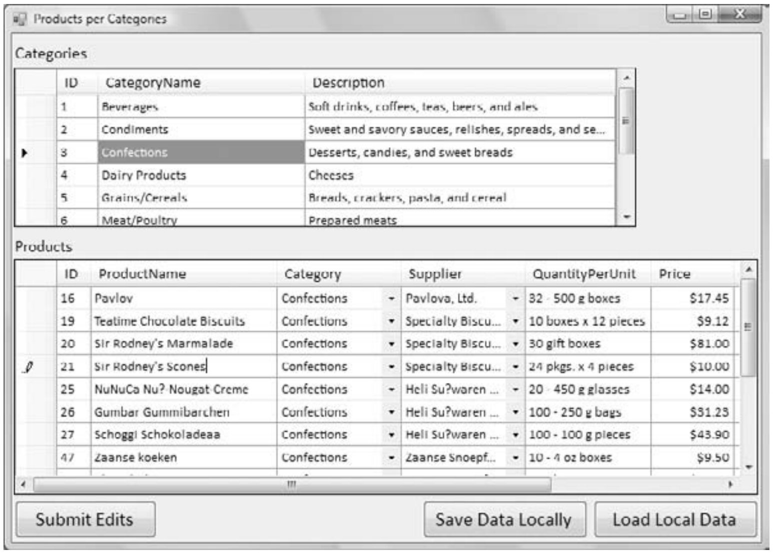
Listing 23.5: Refreshing the DataSet

```
Private Sub ToolStripButton1_Click(...) Handles ToolStripButton1.Click
    If DSCustomers.HasChanges Then
        Dim reply As MsgBoxResult = _
             MsgBox("The DataSet contains changes." & _
             vbCrLf & "Reload data anyway?", _
             MsgBoxStyle.YesNo Or MsgBoxStyle.Exclamation)
        If reply = MsgBoxResult.No Then Exit Sub
    End If
    Me.CustomersTableAdapter.Fill( _
                Me.DSCustomers.Customers)
End Sub
```

We developed a fairly functional application for browsing and editing one of the basic tables of the Northwind database, the Products table. The interface of the application is a bit rough around the edges (that's the least you can say about an interface that displays category and supplier IDs instead of category and supplier names), but we'll come back and adjust the interface of the application in a moment. First, I'd like to discuss another way of using the DataGridView control, namely how to bind related tables to two or more DataGridView controls. This arrangement is the most common one, because we rarely work with a single table.

Binding Hierarchical Tables

In this section, you'll build an interface to display categories and products on two DataGridView controls, as shown in Figure 23.15. The top DataGridView control is bound to the Categories DataTable and displays all the category rows. The lower DataGridView control displays the products of the category selected in the top control. In effect, we'll create two DataGridView controls linked together.



Categories

ID	CategoryName	Description
1	Beverages	Soft drinks, coffees, teas, beers, and ales
2	Condiments	Sweet and savory sauces, relishes, spreads, and se...
3	Confections	Desserts, candies, and sweet breads
4	Dairy Products	Cheeses
5	Grains/Cereals	Breads, crackers, pasta, and cereal
6	Meat/Poultry	Prepared meats

Products

ID	ProductName	Category	Supplier	QuantityPerUnit	Price
16	Pavlov	Confections	Pavlova, Ltd.	32 - 500 g boxes	$17.45
19	Teatime Chocolate Biscuits	Confections	Specialty Biscu...	10 boxes x 12 pieces	$9.12
20	Sir Rodney's Marmalade	Confections	Specialty Biscu...	30 gift boxes	$81.00
21	Sir Rodney's Scones	Confections	Specialty Biscu...	24 pkgs. x 4 pieces	$10.00
25	NuNuCa Nu? Nougat Creme	Confections	Heli Su?waren ...	20 - 450 g glasses	$14.00
26	Gumbar Gummibarchen	Confections	Heli Su?waren ...	100 - 250 g bags	$31.23
27	Schoggi Schokoladeaa	Confections	Heli Su?waren ...	100 - 100 g pieces	$43.90
47	Zaanse koeken	Confections	Zaanse Snoepf...	10 - 4 oz boxes	$9.50

FIGURE 23.15
Viewing related data on two DataGridView controls with the LinkedDataTables application

Follow these steps:

1. Start a new project (it's the LinkedDataTables project), and create a new DataSet that contains the Products, Categories, and Suppliers tables. Name it **DS**.
2. In the Data Sources window, select each table and set its binding option to DataGridView.
3. Drop the Categories table on the form. The editor will place an instance of the DataGridView control on the form and will bind it to the Categories table. It will also create a BindingNavigator object, which we don't really need, so you can delete it. When you drop multiple tables on a form, the editor generates a single toolbar. The navigational buttons apply to the first DataGridView control, but the Save button submits the changes made in all DataTables.

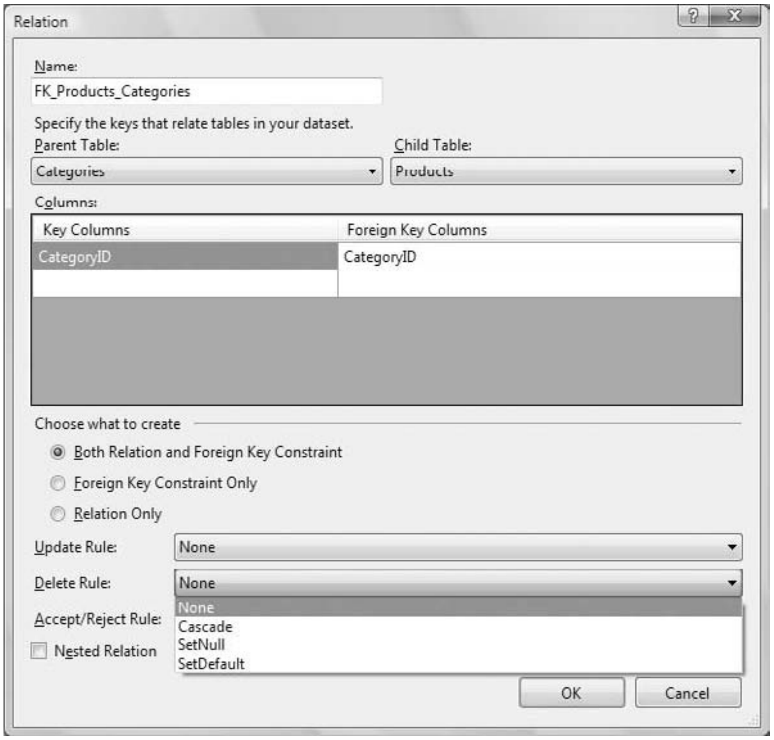
4. Locate the Products table under the Categories table in the Data Sources window and drop it onto the form. If you drop the Products table of the original DataSet onto the form, you'll end up with two grids that are independent of one another. For a more meaningful interface, you must link the two grids, so that when the user selects a category in the upper grid, the corresponding tables are shown automatically in the lower grid. The Products table under the Categories table in the data source represents the rows of the Products table that are related to each row of the Categories table.

Now you can run the application and see how it behaves. Every time you select a category, the selected category's products appear in the lower grid. If you change the `CategoryID` field of a product, it disappears from the grid, as expected. You must select its new category to see it.

Experiment with the new interface. Start editing the two tables on the form. Add new categories, and then add products that belong to these categories. If you attempt to delete a category, the DataGridView will happily remove the row from its table. But didn't the DataSet pick the relationships from the database's definition? Now that both tables are at the client as DataTables with a relationship between them, shouldn't the DataSet reject this operation?

Let's take a look at the properties of the relationship between the two tables. Right-click the *DS* DataSet in the Data Sources window and from the context menu select Edit DataSet With Designer to see the DataSet Designer window. Right-click the line that connects the Products and Categories tables (this line represents the relationship between the two tables) and select Edit Relation to open the Relation dialog box, shown in Figure 23.16.

FIGURE 23.16
Setting relation properties


Relation

Name: FK_Products_Categories

Specify the keys that relate tables in your dataset.

Parent Table: Categories

Child Table: Products

Columns:

Key Columns	Foreign Key Columns
CategoryID	CategoryID

Choose what to create

[✓ Both Relation and Foreign Key Constraint]
Foreign Key Constraint Only
Relation Only

Update Rule: None

Delete Rule: None

The `FK_Products_Categories` relation is marked as Relation Only. In the database, this is a relation and a foreign key constraint. The relation simply relates the two tables if their *`CategoryID`* fields match. Most importantly, the constraint won't let you insert a product that points to a nonexisting category, or delete a category that has related rows in the Products table. Check the radio button Both Relation And Foreign Key Constraint, and then close the dialog box.

Foreign key constraints are subject to three rules: the Update, Delete, and Accept/Reject rules, as shown in Figure 23.16. These rules determine what should happen when a parent row is removed from its table (the Delete rule), when a parent ID is modified (the Update rule), and when users accept or reject changes in the DataSet (the last rule). A rule can be set to None (no action is taken, which means that a runtime exception will be thrown), Cascade (the child rows are updated or deleted), SetNull (the foreign keys of the related rows are set to Null), and SetDefault (the foreign keys of the related rows are set to their default value).

We usually don't change the rules of a relationship in the DataSet, unless you've used rules in the database. Leave them set to None and run the application again. You should avoid setting the Delete and Update rules to Cascade, because this can lead to irrecoverable errors. If you delete a category, for example, it will take with it the related products, and each deleted product will take with it the related rows in the Order Details table. A simple error can ruin the database. There are other situations, which aren't as common, where the Cascade rule can be used safely. When you delete a book in the Pubs database, for example, you want the book's entries in the TitleAuthors table to be removed as well. No rows in the Authors table will be removed, because they're primary, and not foreign, keys in the relation between the TitleAuthors and Authors tables.

Let's return to our interface for editing products and categories. Attempt again to remove a category. This time you'll get a lengthy error message that ends with the following suggestion:

```
To replace this default dialog please handle the DataError event.
```

Let's do exactly that to avoid displaying a totally meaningless message to our users. Open the `DataError` event handler for both controls and insert the following statement:

```
MsgBox(e.Exception.Message)
```

Run the application again and delete a category. This time you'll see the following error message, and the program won't remove the category from the DataGridView, because it can't remove it from its DataSet:

```
Cannot delete this row because constraints are enforced
  on relation FL_Products_Categories, and deleting this
 row will strand child rows.
```

You can still delete products and set their prices to negative values. These two operations are invalid in the context of the database, but quite valid in the client DataSet.

Using the BindingSource as a Data Source

As you recall, binding a DataGridView control to a DataTable is possible by setting the control's `DataSource` property. Binding to two related tables is a bit more involved, so let's see how it's done (short of dropping the two tables on the form and letting the editor handle the details).

To link the two DataGridView controls, you must create a BindingSource object for each one. The BindingSource class encapsulates a data source and is itself a data source. Initialize an instance of this class by setting its `DataSource` and `DataMember` properties for the Categories table:

```
Dim categoriesBS As New BindingSource
categoriesBS.DataSource = DS
categoriesBS.DataMember = "Categories"
```

Then set the upper grid's `DataSource` property to the *`categoriesBS`* object. As for the lower grid, you must create a new BindingSource object and set its `DataSource` property not to the actual DataSet, but to the BindingSource object of the upper grid:

```
Dim productsBS As New BindingSource
productsBS.DataSource = categoriesBS
```

Now here's the tricky part: the `DataMember` property must be set to the name of the relationship between the two tables in the DataSet, so that it's linked to the products of the category selected in the upper control:

```
productsBS.DataMember = "FK_Categories_Products"
```

After the two BindingSource objects have been set up, assign them to the `DataSource` property of their corresponding controls:

```
DataGridView1.DataSource = categoriesBS
DataGridView2.DataSource = productsBS
```

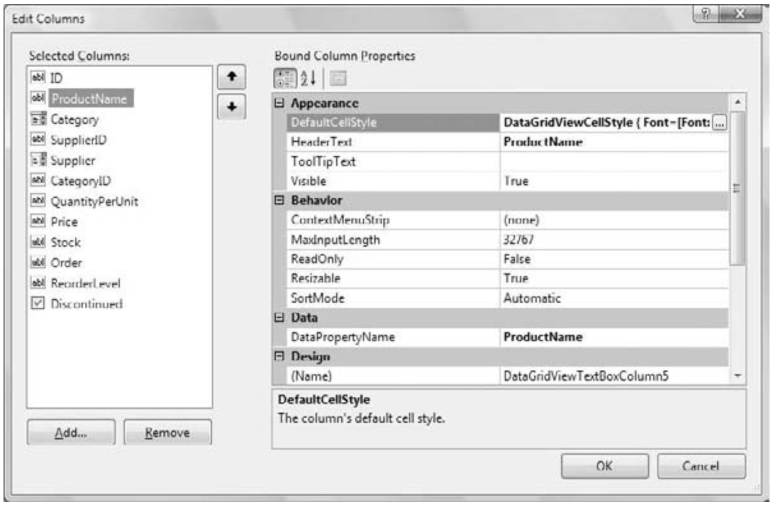
These actions were performed for you automatically, as you dropped the two tables on the form. If you want to bind two related tables of an untyped DataSet, you must set these properties from within your code.

Adjusting the Appearance of the DataGridView Control

The DataGridView control is bound to a single table as soon as you drop the table on the form, but its default appearance leaves a lot to be desired. To begin with, we must set the widths of the columns and hide certain columns. The product IDs, for example, need not be displayed. The numeric fields should be formatted properly and aligned to the right, and the foreign key fields should be replaced by the corresponding descriptions in the primary table. We also should display category and supplier names, instead of IDs.

The DataGridView control exposes a large number of properties, and you can experiment with them in the Properties window. They have obvious names, and you can see the effects of each property on the control as you edit it. Beyond the properties that apply to the entire control, you can also customize the individual columns through the Edit Columns dialog box.

To tweak the appearance of the columns of the DataGridView control, select the control with the mouse and open the Tasks menu by clicking the little arrow in the upper-right corner of the control. This menu contains four check boxes that allow you to specify whether the user is allowed to add/edit/delete rows and reorder columns. To adjust the appearance of the grid's columns, click the Edit Columns hyperlink in the menu and you will see the Edit Columns dialog box, shown in Figure 23.17, where you can set each column's properties.



Edit Columns

Bound Column Properties

Appearance	
DefaultCellStyle	DataGridViewCellStyle { Font=[Font:
HeaderText	ProductName
ToolTipText	
Visible	True
Behavior	
ContextMenuStrip	(none)
MaxInputLength	32767
ReadOnly	False
Resizable	True
SortMode	Automatic
Data	
DataPropertyName	ProductName
Design	
(Name)	DataGridViewTextBoxColumn5

DefaultCellStyle
The column's default cell style.

FIGURE 23.17
Use the Edit Columns dialog box to customize the appearance of the DataGridView control

In the Edit Columns dialog box, you can set each column's header and width, as well as the minimum width of the column. (Users won't be allowed to make the column narrower than its minimum width.) You can also set the `AutoSize` property to True to let the control decide about column widths, but this may result in a very wide control. You can lock certain columns during editing by setting their `ReadOnly` property, or make other columns invisible with the `Visible` property. The most interesting setting in this dialog box is the `ColumnType` property, which is the type of the column. By default, all columns are of the DataGridViewTextBoxColumn type, unless the corresponding field is a Boolean or Image type. A DataGridView column can be one of the following types:

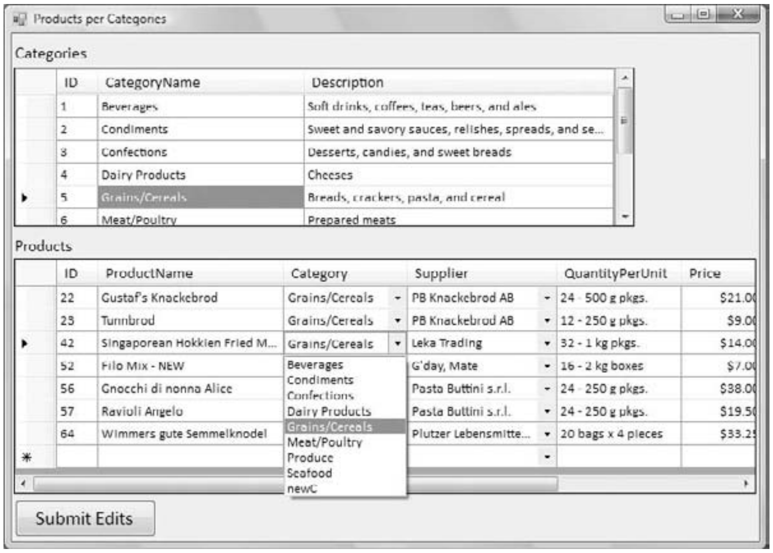
DataGridViewButtonColumn This displays a button whose caption is the bound field. Use buttons to indicate that users can click them to trigger an action. To program the `Click` event of a button column, insert the appropriate code in the control's `CellContentClick` event handler. Your code must detect whether a column with buttons was clicked and, if so, act accordingly. Change the column that displays the product names into a button column and then insert the following statements in the `CellContentClick` event handler of the DataGridView control:

```
Private Sub ProductsDataGridView_CellContentClick( _
         ByVal sender As System.Object, _
         ByVal e As System.Windows.Forms.DataGridViewCellEventArgs) _
   Handles ProductsDataGridView.CellContentClick
    If e.ColumnIndex = 1 Then
        MsgBox(ProductsDataGridView.Rows(e.RowIndex). _
               Cells(e.ColumnIndex).Value.ToString)
    End If
End Sub
```

The code shown here reads the caption of the button that was clicked. You can just as easily read the product's ID and use it to retrieve product details and display them on another form.

DataGridViewCheckBoxColumn This column type is used with True/False columns (the bit data type in SQL Server). The *`Discontinued`* column of the Products table, for example, is mapped automatically to a DataGridView column of this type.

DataGridViewComboBoxColumn This column type is used for foreign keys or lookup fields. You will shortly see how to change the *`CategoryID`* and *`SupplierID`* columns into ComboBox columns, so that users can see category and supplier names instead of IDs. When editing the table, users can expand the list and select another item, instead of having to enter the ID of the corresponding item. Figure 23.18 shows the DataGridView control for displaying products with a ComboBox column for categories and suppliers.



Categories

ID	CategoryName	Description
1	Beverages	Soft drinks, coffees, teas, beers, and ales
2	Condiments	Sweet and savory sauces, relishes, spreads, and se...
3	Confections	Desserts, candies, and sweet breads
4	Dairy Products	Cheeses
5	Grains/Cereals	Breads, crackers, pasta, and cereal
6	Meat/Poultry	Prepared meats

Products

ID	ProductName	Category	Supplier	QuantityPerUnit	Price
22	Gustaf's Knackebrod	Grains/Cereals	PB Knackebrod AB	24 - 500 g pkgs.	$21.0
23	Tunnbrod	Grains/Cereals	PB Knackebrod AB	12 - 250 g pkgs.	$9.0
42	Singaporean Hokkien Fried M...	Grains/Cereals	Leka Trading	32 - 1 kg pkgs.	$14.0
52	Filo Mix - NEW		G'day, Mate	16 - 2 kg boxes	$7.0
56	Gnocchi di nonna Alice		Pasta Buttini s.r.l.	24 - 250 g pkgs.	$38.0
57	Ravioli Angelo		Pasta Buttini s.r.l.	24 - 250 g pkgs.	$19.5
64	Wimmers gute Semmelknodel		Plutzer Lebensmitte...	20 bags x 4 pieces	$33.2

FIGURE 23.18 Displaying product categories and suppliers in a ComboBox control on the DataGridView control

DataGridViewLinkColumn This is similar to the DataGridViewButtonColumn type, only it displays a hyperlink instead of a button. Use the same technique outlined earlier for the Button columns to detect the click of a hyperlink.

DataGridViewImageColumn Use this column type to display images. In general, you shouldn't store images in your databases. Use separate files for your images and include only their paths in the database. Keep in mind that all rows of a DataGridView control have the same height, and if one of them contains an image, the remaining cells will contain a lot of white space.

DataGridViewTextBoxColumn This is the most common column type, and it displays the field in a text box.

Notice that as you change the style of a column, the Bound Column Properties pane of the Edit Columns dialog box is populated with the properties that apply to the specific column type. For a combo box column, for example, you can set the `DropDownWidth` and the `MaxDropDownItems` properties. You can even populate the combo box with a set of values through the `Items` property, just as you would with a regular combo box on the form.

There aren't any properties in the Edit Columns dialog box to adjust the appearance of the selected column. To change the appearance of a column, select the `DefaultCellStyle` property and click the button with the ellipses next to it to see the CellStyle Builder dialog box, which is shown in Figure 23.19.

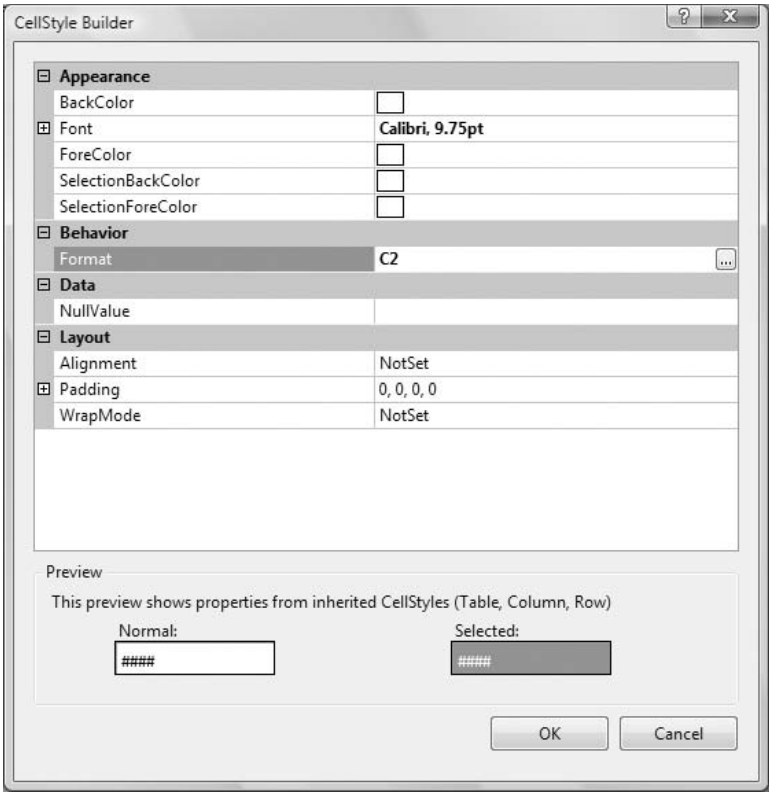


FIGURE 23.19
Use this dialog box to adjust the appearance of a column in a DataGridView control

CellStyle Builder

Appearance	
BackColor	
Font	**Calibri, 9.75pt**
ForeColor	
SelectionBackColor	
SelectionForeColor	
Behavior	
Format	C2
Data	
NullValue	
Layout	
Alignment	NotSet
Padding	0, 0, 0, 0
WrapMode	NotSet

Preview
This preview shows properties from inherited CellStyles (Table, Column, Row)
Normal: ####
Selected: ####

In the CellStyle Builder dialog box, you can set the column's font, set its foreground and background colors, specify whether the text should wrap in its cell, and determine the alignment of the cell's contents. Turning on the wrap mode with the `WrapMode` property doesn't cause the rows to be resized automatically to the height of the tallest cell. To have rows resized automatically, you must set the control's `AutoSizeRowsMode` property to *`All`*. The other possible settings for this property are *`None`*, *`AllHeaders`*, *`AllCellsExceptHeaders`*, *`DisplayedHeaders`*, *`Displayed-CellsExceptHeaders`*, and *`DisplayedCells`*. Finally, you must set the `Format` property for all numeric and date fields, and size the columns according to the data that will be displayed on the control.

This feast of customization techniques is possible because the DataGridView control is bound to a typed DataSet. If you want to use the techniques of the previous chapter to bind a DataGridView control to an untyped DataSet, you can still use the Edit Columns dialog box to add and customize the control's columns, but the process isn't nearly as convenient. You must also remember to set the control's `AutoGenerateColumns` property to False and each column's `DataPropertyName` property to the name of the database column that will be mapped to the corresponding grid column. If the `AutoGenerateColumns` property is left to its default value, which is True, the control will generate a new column for each data column in its data source.

Displaying Lookup Columns in a DataGridView Control

In this section, we're going to customize the *`CategoryID`* and *`SupplierID`* columns, which display the product's category and supplier ID in a text box. Instead of IDs, we must display category and supplier names. Moreover, we will display each product's category and supplier in a ComboBox control, so that users can quickly select another value when editing the product. Let's return to the LinkedDataTables project and set up the bottom GridView control.

Select the DataGridView control with the products, and from the Task menu choose Edit Columns. In the Edit Columns dialog box, select the *`CategoryID`* column and make it invisible by setting its `Visible` property to False. Then click the Add button to add a new column. In the Add Column dialog box, which is shown in Figure 23.20, click the Unbound radio button, set the column's `Name` property to **`colCategory`** and its `HeaderText` property to **`Category`**. Click Add to add the column to the DataGridView control and then Close to close the dialog box.

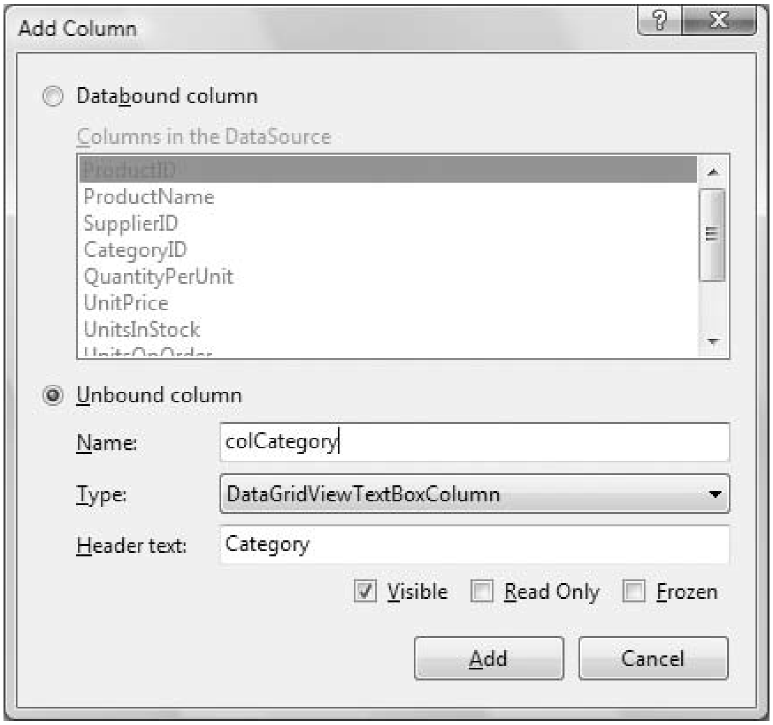


Figure 23.20 Adding a new column to the DataGridView control

Name: colCategory
Type: DataGridViewTextBoxColumn
Header text: Category
[✓ Visible] Read Only Frozen

Back in the Edit Columns dialog box, move the new column to the desired position by using the arrow buttons. You must now set up the new column so that it displays the name of the category that corresponds to the *`CategoryID`* column of the selected product. Locate the

`DataSource` property and click the arrow to expand the data sources. Select the `CategoriesBindingSource` entry. Then set the `DisplayName` property to **`CategoryName`** and the `ValueMember` to **`CategoryID`**. Click OK to close the dialog box. If you run the application now, you'll see that the *`CategoryID`* column has been replaced by the *`Category`* column, which displays a ComboBox with the list of all categories. However, the product's category isn't automatically selected; you have to drop down the items in the combo box to see the categories, and you still don't know the selected product's category. To link the appropriate category name to the selected product's `CategoryID` value, you must set yet another property, the `DataPropertyName` property. If you expand the list of available columns for this property, you'll see the columns of the Products table. Select the *`CategoryID`* column, so that the combo box will display the category name that corresponds to the category ID of the selected product row. Now you have a much better interface for editing the products — you no longer need to enter IDs; you see the name of the selected product's category and you can select a product's category from a drop-down list by using the mouse.

Of course, you must do the same with the Suppliers table. Right-click the NorthwindDataSet object in the Data Sources window and choose Edit DataSet With Designer from the context menu to open the DataSet in design mode.

If the table with the lookup values isn't part of your DataSet, you can easily add a new DataTable to the DataSet. Right-click somewhere on the DataSet Designer's surface and choose Add ➢ Query from the context menu. A new DataTable will be added to the DataSet, and you'll be prompted with a wizard similar to the Data Source Configuration Wizard to select the rows that will be used to populate the DataTable. Select the *`SupplierID`* and *`CompanyName`* columns of the Suppliers table. The wizard will add the Suppliers table to the DataSet, and you're ready to hide the *`SupplierID`* column and replace it with a ComboBox column that contains the names of the suppliers. Select the Edit Column command from the second grid's Tasks menu and set the properties as follows:

`ColumnType`	*`DataGridViewComboBoxColumn`*
`DataSource`	*`Suppliers`*
`DisplayMember`	*`CompanyName`*
`ValueMember`	*`SupplierID`*
`PropertyName`	*`SupplierID`*

Notice that the `ValueMember` property is the *`SupplierID`* column of the Suppliers table, but the *`PropertyName`* property's value is the *`SupplierID`* column of the Products DataTable, because the control is bound to the Products table. The designer will replace the name of the Suppliers DataTable with the `SupplierBindingSource1` object. Run the application now, edit a few products, and then save the edits. You can also edit the categories (names and descriptions only). As a reminder, even the DataSet that was generated by the wizard doesn't enforce check constraints, and you can set the price of a product to a negative value. When you attempt to submit the changes to the database, a runtime exception will be thrown.

The code behind the Submit Edits button isn't new to you. The code sets the `ContinueUpdateOnError` property of the underlying DataAdapter objects to True and then calls the `UpdateAll` method of the TableAdapterManager to submit the changes made to all tables.

The rows that will fail to update the underlying tables in the database will be marked as errors. The DataGridView control marks the rows in error with an icon of an exclamation mark in the column's header, as shown earlier in Figure 23.14. If you hover the mouse pointer over this icon, you'll see a description of the error in a Tooltip box.

Updating hierarchical DataSets isn't as simple as calling the `Update` or `UpdateAll` method. In the LinkedDataTables sample project, I've called the `UpdateAll` method of the TableAdapter-Manager class, which submits first the changes in the parent table (the Categories table) and then the changes in the related table(s). Unless you commit the new rows to the Categories table first, the database will refuse to insert any products that use the IDs of the categories that do not exist in the Categories table. But even if you update the Categories table first and then the Products table, it's not guaranteed that all updates will take place. The order of updates in a hierarchical DataSet is very important, and here's why: Let's say you've deleted all products of a specific category and then the category itself. As soon as you attempt to remove the specific row from the Categories table in the database, the database will return an error indicating that you can't delete a category that has related rows, because the relevant products haven't been removed from the Products table in the database yet.

The proper update order is to submit the deleted rows of the Products table, then perform all updates in the Categories table, and finally submit the insertions and modifications in the Products table. You'll see shortly how to retrieve deleted/modified/new rows from a DataTable at the client (we'll use each row's `DataRowVersion` property) and then how to pass these rows to the `Update` method.

In the meantime, you can experiment with the LinkedDataTables project and perform a few updates at a time. Let's say you want to delete all products in a specific category and then the category itself. To submit the changes without violating the primary/foreign key relationship between the two tables, you must first delete the products and then update the database by clicking the Submit Edits button. After the Products table is updated, you can delete the category from the Categories table and then submit the changes to the database again.

In this section, we've built a functional interface for editing the categories and products of the Northwind database. We started by creating a DataSet with the three tables, and then dropped them on the form to instantiate two DataGridView controls and tweaked the appearance of the two controls with point-and-click operations. We even managed to display combo boxes right in the DataGridView control without a single line of code. The DataGridView control provides the basic editing features, and we were able to put together an interface for our data without any code. We did have to write a bit of code to submit the changes to the database, because the code generated by the wizard couldn't handle anything but the best case scenario. Real-world applications must take into consideration all possible scenarios and handle them gracefully.

Data binding allows you to write data-driven applications quickly, mostly with point-and-click operations, but the default interfaces generated by the wizards are not perfect. You'll see shortly how to use data binding to produce more-elegant interfaces, but they'll require a bit of code. Another problem with data binding is that in most cases you'll end up filling large DataSets at the client — and this is not the best practice with data-driven applications. If your application is going to be used by many users against a single server, you must retrieve a relatively small number of rows from the database and submit the edits as soon as possible. If you keep too much data at the client and postpone the submission of edited rows to the database, you're increasing the chances of concurrency errors. Other users may already have changed the same rows that your application is attempting to update. Of course, you can disable optimistic concurrency and overwrite the changes made by other users, if the nature of the data allows it.

The most common approach is to design a form with search criteria, on which users will specify as best as they can the rows they need. Then you can populate the client DataSet with the rows that meet the specified criteria. If you're writing a connected application, submit the changes to the database as soon as they occur. If you want the users to control when the updates are submitted to the database, display the number of modified/inserted/deleted rows in the form's status bar. You can even pop up a message when the number of edited rows exceeds a limit.

Building More-Functional Interfaces

Editing large tables on a grid isn't the most flexible method. In this section, we'll build an alternate interface for editing the Products table, as shown in Figure 23.21. This is the form of the Products sample project, which uses a data-bound ListBox control to display product names and text boxes for the individual fields. The toolbar at the top allows you to add new rows, delete existing ones, and submit the changes to the database. All the controls on the form are data bound, and the application contains very little code.

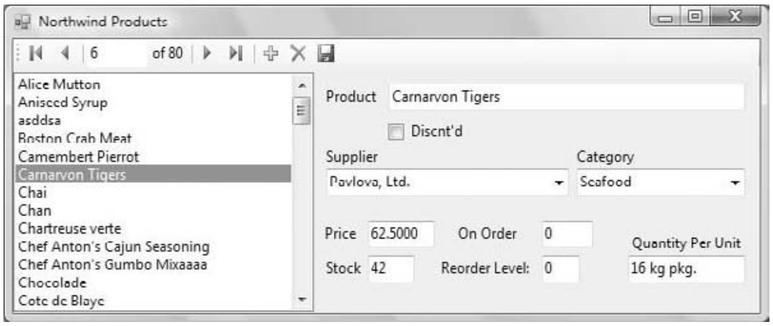


FIGURE 23.21
An alternate interface for editing the Products table

Here's how to get started:

1. Start a new project, the Products project, and create a new DataSet, the *NorthwindDataSet*, with three usual tables: Products, Categories, and Suppliers.
2. Change the binding option for the Products DataTable to ListBox and drop the Products DataTable onto the form. In the Properties window, you must make sure that the ListBox control is properly bound: Its `DataSource` property has been set to the ProductsBinding-Source object that was generated by the editor when you dropped the DataTable on the form, its `DisplayMember` property to ProductName (because it's the first text column in the table), and its `ValueMember` property to ProductID (because it's the table's key). If you want to display a different field to identify the products, change the `DisplayMember` property.
3. Now drop all the fields of the Products DataTable onto the form. The editor will create the appropriate text boxes and bind them to their fields.
4. Rearrange the controls on the form and delete the text boxes that correspond to the *CategoryID* and *SupplierID* columns. Place two ComboBox controls in their place. By the way, even if you set the binding option for the *CategoryID* and *SupplierID* columns to ComboBox, you'll end up displaying IDs in the corresponding controls, not category and supplier names. You can't use any data-binding techniques to automatically set up two lookup fields.
5. Drop two instances of the ComboBox control on the form and name them **`cbCategoryName`** and **`cbCompanyName`**. Select the first one and set the data-binding properties as follows:
   - The `DataSource` is where the control will get its data and it should be Categories-BindingSource, because we want to populate the control with all category names.
   - The `DisplayMember` property is the column we want to view on the control and must be set to the CategoryName column of the Categories DataTable.
   - The `ValueMember` property is the column that will bind the contents of the combo box control to the related table and must be set to the *CategoryID* column of the Categories DataTable.

- Finally, you must set the `SelectedValue` property, in the DataBindings section of the Properties window, to the matching column of the child table, which is the *`CategoryID`* column of the ProductsBindingSource. The control will automatically select the row of the Categories table whose category ID matches the *`CategoryID`* column of the Products DataTable.

6. Perform similar steps with the other combo box, which displays the *`CompanyName`* column of the Suppliers DataTable and is bound to the *`SupplierID`* column of the ProductsTable.

Run the application and see how it behaves. Every time you select a product in the list, the product's details appear in the data-bound text boxes, and its category and supplier are displayed in the two combo box controls. Use the list to navigate through the products, the Add button to add a new product, and the Delete button to delete a product. You can edit the product's fields on the form, and the edits will be written to the DataSet as soon as you move to another row.

You have created a functional application for selecting products and viewing their details. If all you need is a browsing application for the products, you can set the `ReadOnly` property of all TextBox controls on the form to True.

If you attempt to enter a new product and leave its *`Discontinued`* column to Null, a runtime exception will be raised. This problem is easy to fix by specifying a default value for the *`Discontinued`* column. To do so, open the DataSet in the designer, locate the *`Discontinued`* column in the Products table, and select Properties from the context menu. The `DefaultValue` property's value is *`DBNull`*. Set it to False, so that unspecified fields will be automatically set to False.

You'll also get an error message if you attempt to submit the edits and the DataSet contains a product with a negative price. Such trivial errors can be caught and handled from within your code — no need to send the rows to the database and get back an error message. We'll write a few statements to detect trivial errors, such as negative prices (or negative stocks, for that matter). First we must decide how to handle these errors. Do we pop up a message box every time we detect an error condition? This will drive users crazy. Do we reject changes until the user enters a valid value? It's a better approach, but remember: Data-entry operators don't look at the monitor. They expect that the Tab (or Enter) key will take them to the next field. The best approach is to do what the DataGridView control does: display an error icon next to the control in error.

Add an instance of the ErrorProvider control on the form. This control displays the exclamation mark icon next to a control. To display the error icon, you must call the control's `SetError` method, passing as arguments the control in error and the message for the error. (The message will be displayed in a ToolTip box when users hover the pointer over the icon.)

To detect errors from within your code, you need to insert some code in the `CurrentItemChanged` event handler of the ProductsBindingSource. Insert the statements shown in Listing 23.6 in this event handler.

Listing 23.6: Catching Data-Entry Errors in Your Code

```
    Private Sub ProductsBindingSource_CurrentItemChanged( _
                ByVal sender As Object, _
                ByVal e As System.EventArgs) Handles _
                ProductsBindingSource.CurrentItemChanged
        ErrorProvider1.Clear()
        Dim product As DSProducts.ProductsRow
        product = CType(CType( _
                  ProductsBindingSource.Current, _
                  DataRowView).Row, DSProducts.ProductsRow)
        If Not product.IsUnitPriceNull AndAlso _
             Convert.ToDecimal(product.UnitPrice) < 0 Then
            ErrorProvider1.SetError(UnitPriceTextBox, _
                      "PRICE CAN'T BE NEGATIVE!")
        End If
        If ProductNameTextBox.Text.Trim.Length = 0 Then
            If CType(ProductsBindingSource.Current, _
               DataRowView).Row.RowState <> _
               DataRowState.Detached Then
                    ErrorProvider1.SetError( _
                         ProductNameTextBox, _
                         "PRODUCT NAME CAN'T BE BLANK!")
            End If
        End If
    End Sub
```

This code segment requires some explanation. The `CurrentItemChanged` event is fired every time the user selects another row or column on the control. The code in this event handler retrieves the current row with the `Current` property of the ProductBindingSource object. This property returns an object, which is a DataRowView object. This is why I cast it to the DataRowView type, then retrieve its `Row` property, and finally cast it to the ProductsRowtype. The *`product`* variable represents the currently selected row in the DataSet. This is a typed variable, and I can access the columns of the current row as properties. If the *`UnitPrice`* column has a negative value, the code sets an ErrorProvider control to display the error next to the corresponding text box.

Viewing the Deleted Rows

One unique aspect of this interface is that it provides a link to display the deleted rows. These rows exist in the DataSet, but they're not shown on the interface. The inserted and modified rows are on the ListBox control, and users can review them. You may even provide a button to display the old and new versions of the edited rows. But users have no way of reviewing the deleted rows.

The Show Deleted Rows link opens an auxiliary form, like the one shown in Figure 23.22, which displays the deleted rows in a CheckedListBox control. Users are allowed to select some of the deleted rows and restore them.

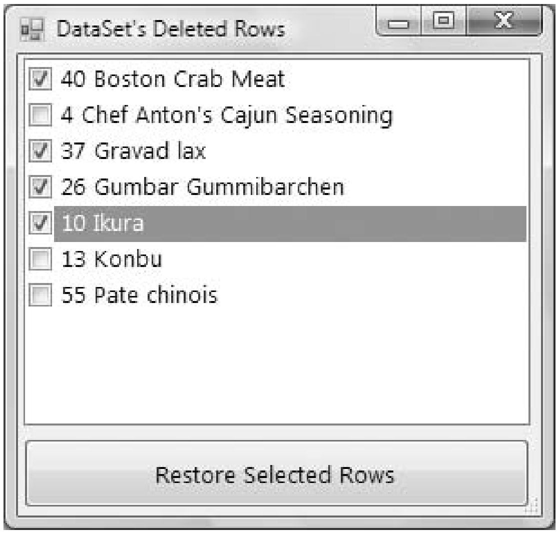
Listing 23.7 shows the code that retrieves the deleted rows and displays them on the auxiliary form.

Listing 23.7: Retrieving and Displaying the Deleted Rows

```
    Private Sub LinkLabel1_LinkClicked(...) _
                       Handles LinkLabel1.LinkClicked
        Form2.CheckedListBox1.Items.Clear()
        For Each row As DataRow In DSProducts.Products.Rows
            If row.RowState = DataRowState.Deleted Then
```

```
                Form2.CheckedListBox1.Items.Add( _
                row.Item("ProductID", _
                DataRowVersion.Original) & " " & _
                row.Item("ProductName", _
                DataRowVersion.Original))
        End If
    Next
    Form2.ShowDialog()
    Dim SelectedIDs As New ArrayList
    For Each itm As String In Form2.CheckedListBox1.CheckedItems
            SelectedIDs.Add(Convert.ToInt32( _
               itm.Substring(0, itm.IndexOf(" ") + 1)))
    Next
    Dim cust As DSProducts.ProductsRow
    For Each cust In DSProducts.Products
        If cust.RowState = DataRowState.Deleted _
            AndAlso SelectedIDs.Contains(cust.Item( _
            "ProductID", DataRowVersion.Original)) Then
            cust.RejectChanges()
        End If
    Next
End Sub
```

FIGURE 23.22
Reviewing the deleted rows in the DataSet


DataSet's Deleted Rows

- [x] 40 Boston Crab Meat
- [ ] 4 Chef Anton's Cajun Seasoning
- [x] 37 Gravad lax
- [x] 26 Gumbar Gummibarchen
- [x] 10 Ikura
- [ ] 13 Konbu
- [ ] 55 Pate chinois

The code goes through the `Rows` collection of the Products DataTable and examines the `RowState` property of each row. If its value is RowState.Deleted, it adds the row's ***`ProductID`*** and *`ProductName`* fields to the CheckedListBox control of the auxiliary form. Then it displays the form modally, and when the user closes it, the code retrieves the IDs of the selected rows into the *SelectedIDs* ArrayList. The last step is to restore the selected rows. The code goes through all rows again, examines their RowState property, and if a row is deleted and its ID is in the *SelectedIDs* ArrayList, it calls the RejectChanges method to restore the row. The restored rows are automatically displayed in the ListBox control because this control is bound to the DataSet.

Real World Scenario

Hand-Crafting an Application's Interface

A practical feature you can add to the interface of a disconnected application is the ability to review the modifications. You can display the original and proposed versions of the inserted/modified rows, as well as the original versions of the deleted rows on an auxiliary form, as we have done in the preceding example.

If you don't mind writing a bit of code, you can display the original values of the edited rows in the same controls on your form. Because the controls are data bound, you can't display any different values on them; if you do, they'll be stored in the DataSet as well. But you can place nonbound controls, such as Label controls, in the place of the data-bound control. The Labels normally will be invisible, but when the user presses a function key, they can become visible, hiding the regular editable controls. When the user requests the original versions of the fields, you must populate the Labels from within your code and display them by toggling their Visible property. You must also hide the regular controls on the form.

Another approach would be to suspend data binding momentarily by calling the SuspendBinding method of the BindingSource class. After data binding has been suspended, you can populate the text boxes on the form at will, because the values you display on them won't propagate to the DataSet. To display the bound values again, call the ResumeBinding method.

Another interesting feature you can add to a data-driven application is to display the state of the current row in the form's status bar. You can also give users a chance to undo the changes by pressing a button on the status bar. All you have to do in this button's Click event handler is to call the RejectChanges method on the current row. Use the Current property of the ProductsBindingSource to retrieve the current row and cast it to the ProductsRow type, and then call the RejectChanges method.

You can select one of the data-editing applications presented in this chapter, perhaps the Products application, and add as many professional features as you can to it. Start by adding a status bar to the form and display on it the state of the current row. For modified rows, display a button on the toolbar that allows users to view and/or undo the changes to the current row. Program the form's KeyUp event handler, as explained in Chapter 4, ''Graphical Interface Design,'' so that the Enter key behaves like the Tab key. Users should be able to move to the next field by pressing Enter. Finally, you can display an error message for the current row on a label in the form's status bar. Or display a message such as *Row has errors* as a hyperlink and show the actual error message when users click the hyperlink. Test the application thoroughly and insert error handlers for all types of errors that can be caught at the client. Finally, make the edited rows a different color in the two DataGridView controls on the form of the LinkedDataTables sample project. To do so, you must insert some code in the control's `RowValidated` event, which is fired after the validation of the row's data. You'll need to access the same row in the DataSet and examine its `RowState` property by retrieving the `Current` property of the ProductsBindingSource object, as shown in Listing 23.8.

Listing 23.8: Coloring the Edited and Inserted Rows on the DataGridView Control

```
Private Sub ProductsDataGridView_RowValidated(...) _
            Handles ProductsDataGridView.RowValidated
    Dim row As DS.ProductsRow
        row = CType(CType(ProductsBindingSource.Current, _
              DataRowView).Row, DS.ProductsRow)
    If row.RowState = DataRowState.Modified Then
        ProductsDataGridView.Rows(e.RowIndex). _
             DefaultCellStyle.ForeColor = Color.Green
    Else
        ProductsDataGridView.Rows(e.RowIndex). _
             DefaultCellStyle.ForeColor = Color.Black
    End If
    If row.RowState = DataRowState.Added Then
        ProductsDataGridView.Rows(e.RowIndex). _
             DefaultCellStyle.ForeColor = Color.Blue
    Else
        ProductsDataGridView.Rows(e.RowIndex). _
             DefaultCellStyle.ForeColor = Color.Black
    End If
End Sub
```

The code in the listing sets the foreground color of modified rows to green and the foreground color of inserted rows to blue. From within the same event's code, you can set the text of a Label on the form's status bar to the row's error description. If you run the application now, you'll see that it paints modified and inserted cells differently, but only while you're working with products of the same category. If you select another category and then return to the one whose products you were editing, they're no longer colored differently, because the `RowValidated` event is no longer fired. To draw rows differently, you must duplicate the same code in the control's `RowPostPaint` event as well.

If you carefully test the revised application, you'll realize that the DataGridView control doesn't keep track of modified rows very intelligently. If you append a character to the existing product name and then delete it (without switching to another cell between the two operations), the DataGridView control considers the row modified, even though the original and proposed versions are identical. As a result, the application will render the row in green. Moreover, the `UpdateAll` method will submit the row to the database.

For a truly disconnected application, you should give users a chance to store the data locally at the client. The LinkedDataTables application's main form contains two more buttons: the Save Data Locally and Load Local Data buttons. The first one saves the DataSet to a local file via the `WriteXml` method of the DataSet, and the second button loads the DataSet via the `ReadXml` method. The application uses the `tmpData.#@#` filename in the application's folder to store the data. It also uses an overloaded form of the two methods to accept an additional argument that stores not only the data, but the changes as well. Here's the code behind the two buttons:

```
Private Sub Button1_Click(...) Handles Button1.Click
    DS.WriteXml("tmpData.#@#", XmlWriteMode.DiffGram)
End Sub

Private Sub Button2_Click(...) Handles Button2.Click
    ProductsBindingSource.SuspendBinding()
    DS.ReadXml("tmpData.#@#", XmlReadMode.DiffGram)
    ProductsBindingSource.ResumeBinding()
End Sub
```

The Bottom Line

Design and use typed DataSets. Typed DataSets are created with visual tools at design time and allow you to write type-safe code. A typed DataSet is a class created by the wizard on the fly and it becomes part of the project. The advantage of typed DataSets is that they expose functionality specific to the selected tables and can be easily bound to Windows forms. The code that implements a typed DataSet adds methods and properties to a generic DataSet, so all the functionality of the DataSet object is included in the autogenerated class.

Master It Describe the basic components generated by the wizard when you create a typed DataSet with the visual tools of Visual Studio.

Bind Windows forms to typed DataSets. The simplest method of designing a data-bound form is to drop a DataTable, or individual columns, on the form. DataTables are bound to DataGridView controls, which display the entire DataTable. Individual columns are bound to simple controls such as TextBox, CheckBox, and DateTimePicker controls, depending on the column's type. In addition to the data-bound controls, the editor generates a toolbar control with some basic navigational tools and the Add/Delete/Save buttons.

Master It Outline the process of binding DataTables to a DataGridView control.

Chapter 24

Advanced DataSet Operations

As you know very well by now, DataSets are miniature databases that reside in the client computer's memory. They're made up of tables related to one another, they enforce relations, and they're practically as close to a client-side database as you can get without installing an actual database management system at every client. However, you can't execute SQL statements directly against the DataSet's data. You can't update a DataTable by issuing an `UPDATE` statement, and can't add rows with an `INSERT` statement. You know how to iterate through the rows of a DataTable and how to locate its related rows in other tables, and how to select rows from a DataTable with the `Select` method. You can iterate through the rows of a table, locate the ones that meet certain criteria, and process their columns in any way you wish: calculate aggregates on selected columns, update rows in other tables based on the values of certain columns, format text columns, even create new columns and set their values. And, of course, you know how to edit, add and delete rows, and submit the DataSet to the server.

You can also process the data in the DataSet at the client with LINQ. Besides LINQ to SQL, which was discussed briefly in Chapter 17, ''Querying Collections and XML with LINQ,'' there's another component, the LINQ to DataSet component, which enables you to query the DataSet's data. In this chapter, I'll focus on more-traditional querying techniques.

In this chapter, you'll learn how to do the following:

- Use SQL to query DataSets
- Add calculated columns to DataTables
- Compute aggregates over sets of rows

Working with SQL Expressions

Let's consider a DataSet that contains the Orders and Order Details tables of the Northwind database. In Chapter 22, ''Programming with ADO.NET,'' you saw how to populate a DataSet with data from one or more database tables, how to iterate through a table's rows, and how to traverse the related rows. In this chapter, you'll look at some more-advanced features, such as adding custom columns to the DataSet's DataTables and filtering a DataTable's rows with SQL-like expressions.

You can add a new column to the Order Details DataTable of the DataSet (and not the actual table in the database) and use this column to store each line's total. You can calculate the line's total by multiplying the quantity by the price and then subtracting the discount. Similarly, you can add a column to the Orders DataTable and store the order's total there. To calculate an order's total, you can sum the line totals in the Order Details DataTable over all the rows that belong to the specific order. Writing VB code to accomplish these tasks is almost trivial; the simplest approach is to write a loop that iterates through a DataTable's rows and calculates the aggregate. This approach, however, is neither the most elegant nor the most efficient. A better approach is to use SQL expressions to define calculated columns, as well as use SQL expressions to select rows.

Do we really need all this functionality at the client? We can certainly write code to retrieve the rows needed at any time from the database. A connected application should request data from the database as needed. If you're writing a disconnected or eventually connected application, you should be able to load a DataSet with the data you're interested in and process it at the client. Even connected applications may benefit from the richness of the DataSet.

Selecting Rows

The `Select` method of the DataTable object allows you to select the rows that meet certain criteria. The criteria are expressed with statements similar to the `WHERE` clause of a `SELECT` statement. There are several overloaded versions of the `Select` method. The simplest form of the method accepts no arguments at all and returns all the rows in the DataTable to which it's applied. Another form accepts a string argument, which is a filter expression equivalent to the `WHERE` clause of the `SELECT` statement, and returns the rows that match the specified criteria. Another form of the `Select` method accepts the filter expression and a second string argument that determines the order in which the rows will be returned. The last overloaded form of the method accepts the same two arguments and a third one that determines the state of the rows you want to retrieve:

```
DataTable.Select(filter, ordering, DataRowState)
```

The `Select` method returns an array of DataRow objects, which are not linked to the original rows. In other words, if you edit one of the rows in the array returned by the `Select` method, the matching row in the DataTable will not be modified. Even if the DataSet is strongly typed, the `Select` method returns an array of generic DataRow objects, not an array of the same type as the rows of the DataTable to which the Select method was applied. The following statements retrieve the ''expensive'' products from the Products DataTable:

```
Dim expensive() As DataRow
expensive = DS.Products.Select("UnitPrice > 100")
```

After the execution of the `Select` method, the *`expensive`* array will hold the rows whose *`UnitPrice`* field is more than $100. You can also combine multiple filter expressions with the usual Boolean operators. To retrieve the expensive product with a stock over three units, use the following filter expression:

```
"UnitPrice > 100 AND UnitsInStock > 3"
```

Assuming that the Orders DataTable holds the rows of the Orders table, you can retrieve the headers of the orders placed in the first three months of 1998 via the following statements:

```
Dim Q1_1998() As DataRow
Q1_1998 = DS.Orders.Select( _
  "OrderDate >= '1/1/1998' AND OrderDate <= '3/31/1998'")
```

If you want the rows sorted by shipment date, add an argument indicating the name of the column on which the columns will be sorted and the ASC or DESC keyword:

```
DS.Orders.Select( _
  "OrderDate >= '1/1/1998' AND OrderDate <= '3/31/1998'", _
  "DateShipped DESC")
```

You can select the rows that have been added to the DataSet by specifying the third argument to the Select method and two empty strings in place of the first two arguments:

```
Dim newRows() As DataRow
newRows = DS.Orders.Select("","", DataRowState.Added)
```

Simple Calculated Columns

Sometimes you'll need to update a column's value based on the values of other columns. For example, you may wish to maintain a column in the Orders DataTable that has the order's total. For this column to be meaningful, its value should be updated every time a related row in the Order Details DataTable is modified. In the actual database, you'd do this with a trigger. However, we want to avoid adding too many triggers to our tables because they slow all data-access operations, not to mention that you'll be duplicating information. If a trigger fails to execute, the line total may not agree with the actual fields of the same row. One of the most basic rules in database design is to never duplicate information in the database. Because duplication of information sometimes helps programmers in retrieving totals from the database quickly, many developers chose to break this rule and maintain totals, even though the totals can be calculated from row data.

To maintain totals in a DataSet, you can add calculated columns to its DataTables. You specify a formula for the calculated column, and every time one of the columns involved in the formula changes, the calculated column's value changes accordingly. Note that calculated columns may involve columns in the same table, or aggregates that involve columns in related tables. As you will see, calculated columns can simplify many complex reporting applications.

To add a new DataColumn object to a DataTable, use the Add method of the table's Columns collection. The Columns property of the DataTable object is a collection, and you can create new columns to the table by adding the DataColumn object to this collection. One of the overloaded forms of this method allows you to specify not only the column's name and data type, but also its contents. The following statement adds the *LineTotal* column to the Order Details table and sets its value to the specified expression:

```
DS.Tables["Order Details"].Columns.Add( _
     "LineTotal", System.Type.GetType("System.Decimal"), _
     "(UnitPrice * Quantity) * (1 - Discount)")
```

The last argument of the Add method is an expression, which calculates the detail line's extended price. This expression isn't calculated when the DataSet is loaded; it's calculated on-the-fly, every time the column is requested. You can change the values of the columns involved in the expression at will, and every time you request the value of the *LineTotal* column, you'll get back the correct value. You should notice that order and invoice data isn't changed after it's been recorded. In other words, you shouldn't edit the contents of an order, or an invoice, after the receipt has been printed. Because order data is static, you can include the line total in your query with a statement like the following:

```
SELECT  [Order Details].*, _
    UnitPrice * Quantity * (1 - Discount) AS LineTotal
FROM    [Order Details]
```

The *LineTotal* column won't change after it's been read, even if you change the row's quantity, price, or discount. If the application is going to use the DataSet for browsing only, you can populate it with the preceding SQL statement. But what if you want to run some ''what if'' scenarios? For example, you may allow users to edit the discount for selected customers and see how it affects profit. In this case, reading the line's total from the database isn't going to do you any good, unless you want to update the *LineTotal* column from within your code. If you add the same column as a calculated column to the Order Details table, you can freely edit discounts and the *LineTotal* column will be always up-to-date.

The *LineTotal* calculated column is as simple as it can get because it involves the values of a few other columns in the same row. Another trivial example is the concatenation of two or more columns, as in the following example:

```
DS.Employees.Columns.Add("EmployeeName", _
   System.Type.GetType("System.String"), _
   "LastName + ' ' + FirstName")
```

You can also use functions in the calculated column expressions. The following calculated column has the value Out of Stock if the *AvailableQuantity* column is 0 or negative, and the value Immediate Availability if the same column is positive:

```
Ds.Tables["Stock"].Columns.Add("Availability", _
     System.Type.GetType("System.String"), _
     "IIF(AvailableQuantity > 0, " & _
     "'Immediate Availability', 'Out of Stock')
```

You can use any of the functions of T-SQL in your calculated column (string functions, date and time functions, math functions, and so on). You can also combine multiple conditions with the `AND`, `OR`, and `NOT` operators. Finally, you can use the `IN` and `LIKE` operators for advanced comparisons.

By the way, the calculated column we just added to our DataSet doesn't violate any database design rules, because the extra column lives in the DataSet and is used primarily for browsing purposes. We haven't touched the database.

Calculated Columns with Aggregates

In addition to the simple calculated columns you explored in the preceding section, you can create calculated columns based on the values of multiple related rows. An aggregate column based on the values of related columns in other tables uses a slightly different syntax. To use aggregates, you must learn two new functions: `Child()` and `Parent()`. They both accept a relation name as an argument and retrieve the child rows or the parent row, respectively, of the current row, according to the specified relation.

Let's consider again a DataSet with the Orders and Order Details tables. The function Child (Orders_Order_Details)returns the rows of the Order Details table that belong to the current row of the Orders table: the child rows of the current order under the Orders_Order_Details relation. Likewise, the function Parent(Orders_Order_Details) returns the row of the Orders table, to which the current Order Details row belongs. The Child() function can be applied to the rows of the parent table in a relation, and the Parent() function can be applied to the rows of the child table. If the table to which either function applies has a single relation to another table, you can omit the relation's name. If the DataSet contains only the Orders and Order Details tables, you can use the functions Child and Parent to refer the current row's child and parent row(s) without specifying the name of the relation.

To add the Items calculated column to the Orders table and store the total number of items in the order to this column, use the following statement:

```
DS.Orders.Columns.Add("Items", _
      System.Type.GetType("System.Int32", _
      SUM(child.Quantity))
```

or the following equivalent statement:

```
DS.Orders.Columns.Add("Items", _
      System.Type.GetType("System.Int32", _
      SUM(child(Orders_Order_Details).Quantity))
```

The last statement is longer but easier to read, and the same code will work even after you add new relations to the Orders table. The Items calculated column is based on an aggregate over selected rows in another table and it can simplify reporting applications that need to display totals along with each order. You can use a similar but slightly more complicated expression to calculate the total of each order:

```
DS.Orders.Columns.Add("OrderTotal", _
      System.Type.GetType("System.Decimal"), _
      SUM(child(Orders_Order_Details).UnitPrice * _
        child(Orders_Order_Details).Quantity * _
        (1 - child(Orders_Order_Details).Discount)")
```

What if we wanted to include the freight cost in the order's total? A calculated column can either be a simple one or contain aggregates over related rows, but not both. If you need a column with the grand total of the order, you must add yet another calculated column to the Orders table and set it to the sum of the OrderTotal and Freight columns.

Computing Expressions

In addition to calculated columns, you can use the Compute method of the DataTable object to perform calculations that involve the current DataTable's rows, their child rows, and their parent rows. The following statement returns an integer value, which is the number of orders placed by a specific customer:

```
DS.Orders.Compute("COUNT(OrderID)", "CustomerID = 'ALFKI'")
```

Actually, the `Compute` method returns an object, which you must cast to the desired data type. To store the value returned by the preceding statement to an integer variable, use the following expression:

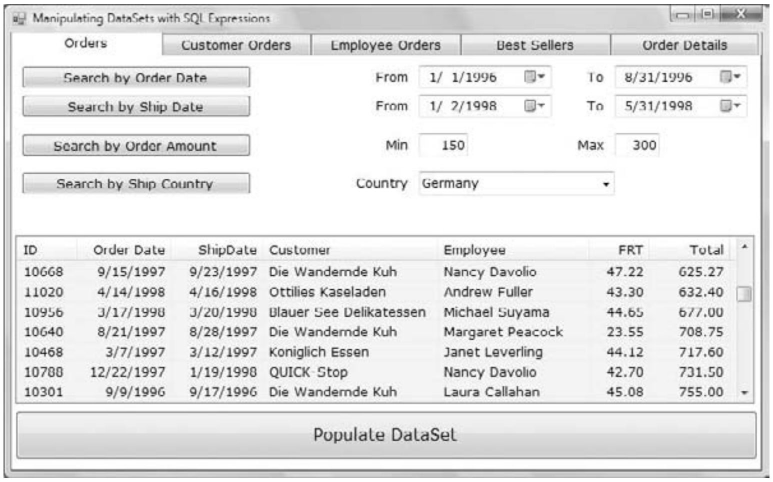
```
Dim count As Integer
count = Convert.ToInt32(DS.Orders. _
  Compute("COUNT(OrderID)", "CustomerID = 'ALFKI'"))
```

The `Compute` method returns a value; it doesn't save the value to another column. Moreover, the `Compute` method allows you to limit the rows that participate in the aggregate with an expression identical to the `WHERE` clause of a `SELECT` statement.

Consider again a DataSet with the Orders and Order Details DataTables. If you need the total of all orders for a specific customer, you must call the `Calculate` method, passing a restriction as an argument, as shown in the preceding statement. If the DataSet contained the Customers table as well, you could add a calculated column to the Customers table and store there each customer's total for all orders. This value should be identical to the one returned by the `Compute` method.

VB 2008 at Work: The SQL Expressions Project

In this sample application, we'll put together all the information presented so far in this chapter to build a functional application for exploring the Northwind Corporation's sales. The application's main form consists of a TabControl with five tab pages, which are shown in Figures 24.1 through 24.5. To use the application, you must click the Populate DataSet button at the bottom of the form.

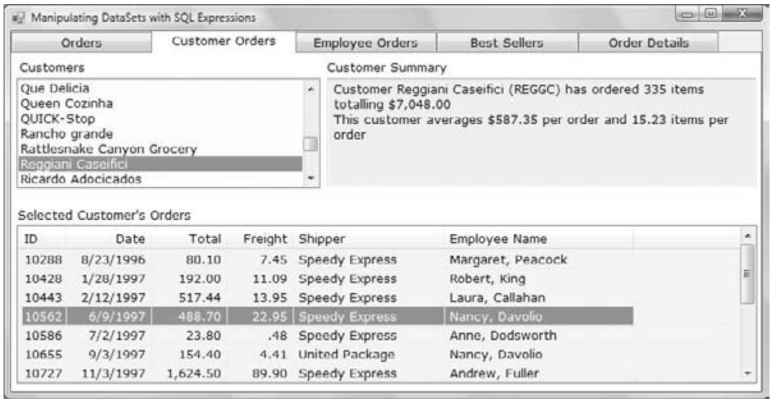


From 1/ 1/1996 To 8/31/1996
From 1/ 2/1998 To 5/31/1998
Min 150 Max 300
Country Germany

ID	Order Date	ShipDate	Customer	Employee	FRT	Total
10668	9/15/1997	9/23/1997	Die Wandernde Kuh	Nancy Davolio	47.22	625.27
11020	4/14/1998	4/16/1998	Ottilies Kaseladen	Andrew Fuller	43.30	632.40
10956	3/17/1998	3/20/1998	Blauer See Delikatessen	Michael Suyama	44.65	677.00
10640	8/21/1997	8/28/1997	Die Wandernde Kuh	Margaret Peacock	23.55	708.75
10468	3/7/1997	3/12/1997	Koniglich Essen	Janet Leverling	44.12	717.60
10788	12/22/1997	1/19/1998	QUICK-Stop	Nancy Davolio	42.70	731.50
10301	9/9/1996	9/17/1996	Die Wandernde Kuh	Laura Callahan	45.08	755.00

FIGURE 24.1
On the Orders tab of the SQL Expressions application, you can select orders based on various user-supplier criteria

On the Orders tab, you can select orders based on various criteria. Note that you can't combine the criteria, but it wouldn't be too much work to take into consideration multiple criteria and combine them with the `AND` operator. On the Customer Orders tab, you can view the customers and select one to see that customer's orders, as well as a summary of the selected customer's orders. The Employee Orders tab is almost identical; it displays orders according to employees. On the Best Sellers tab, you can click the Top Products and Customers button to view the best-selling products and the top customers. If you click a customer name, you'll be switched to the Customer Orders tab, where the same customer will be selected on the list and the customer's orders will appear in the ListView control at the bottom of the form. On the Order Details tab, you see the selected order's details. No matter where you select an order by double-clicking it, its details will appear on this tab.



Customers

Que Delicia
Queen Cozinha
QUICK-Stop
Rancho grande
Rattlesnake Canyon Grocery
Reggiani Caseifici
Ricardo Adocicados

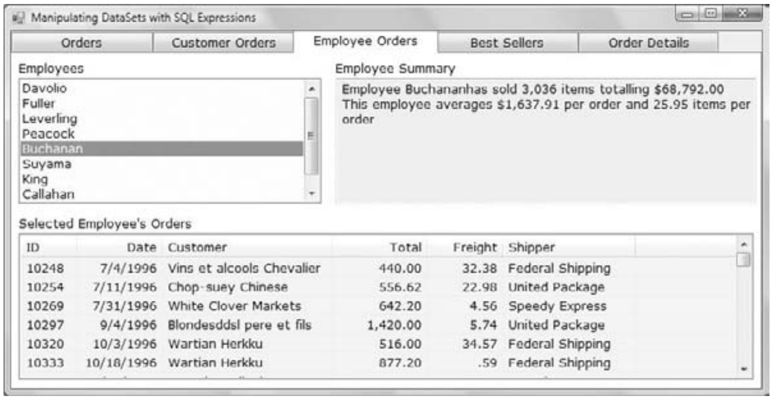
Customer Summary

Customer Reggiani Caseifici (REGGC) has ordered 335 items totalling $7,048.00
This customer averages $587.35 per order and 15.23 items per order

Selected Customer's Orders

ID	Date	Total	Freight	Shipper	Employee Name
10288	8/23/1996	80.10	7.45	Speedy Express	Margaret, Peacock
10428	1/28/1997	192.00	11.09	Speedy Express	Robert, King
10443	2/12/1997	517.44	13.95	Speedy Express	Laura, Callahan
10562	6/9/1997	488.70	22.95	Speedy Express	Nancy, Davolio
10586	7/2/1997	23.80	.48	Speedy Express	Anne, Dodsworth
10655	9/3/1997	154.40	4.41	United Package	Nancy, Davolio
10727	11/3/1997	1,624.50	89.90	Speedy Express	Andrew, Fuller

FIGURE 24.2 On the Customer Orders tab, you can review customers and their orders.



Employees

Davolio
Fuller
Leverling
Peacock
Buchanan
Suyama
King
Callahan

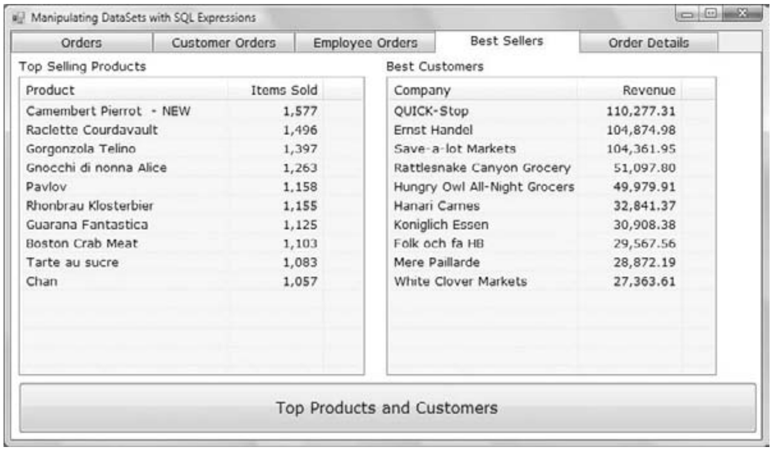
Employee Summary

Employee Buchananhas sold 3,036 items totalling $68,792.00
This employee averages $1,637.91 per order and 25.95 items per order

Selected Employee's Orders

ID	Date	Customer	Total	Freight	Shipper
10248	7/4/1996	Vins et alcools Chevalier	440.00	32.38	Federal Shipping
10254	7/11/1996	Chop-suey Chinese	556.62	22.98	United Package
10269	7/31/1996	White Clover Markets	642.20	4.56	Speedy Express
10297	9/4/1996	Blondesddsl pere et fils	1,420.00	5.74	United Package
10320	10/3/1996	Wartian Herkku	516.00	34.57	Federal Shipping
10333	10/18/1996	Wartian Herkku	877.20	.59	Federal Shipping

FIGURE 24.3 On the Employee Orders tab, you can review employees and their orders.


Manipulating DataSets with SQL Expressions
Orders
Customer Orders
Employee Orders
Best Sellers
Order Details
Top Products and Customers

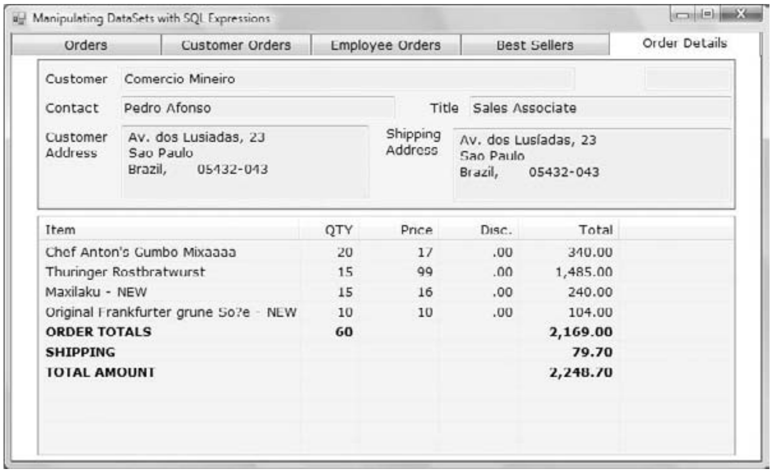
Top Selling Products

Product	Items Sold
Camembert Pierrot - NEW	1,577
Raclette Courdavault	1,496
Gorgonzola Telino	1,397
Gnocchi di nonna Alice	1,263
Pavlov	1,158
Rhonbrau Klosterbier	1,155
Guarana Fantastica	1,125
Boston Crab Meat	1,103
Tarte au sucre	1,083
Chan	1,057

Best Customers

Company	Revenue
QUICK-Stop	110,277.31
Ernst Handel	104,874.98
Save-a-lot Markets	104,361.95
Rattlesnake Canyon Grocery	51,097.80
Hungry Owl All-Night Grocers	49,979.91
Hanari Carnes	32,841.37
Koniglich Essen	30,908.38
Folk och fa HB	29,567.56
Mere Paillarde	28,872.19
White Clover Markets	27,363.61

FIGURE 24.4 On the Best Sellers tab, you can view the best-selling products and the customers with the largest sales figures.



Manipulating DataSets with SQL Expressions

Orders | Customer Orders | Employee Orders | Best Sellers | Order Details

Customer: Comercio Mineiro

Contact: Pedro Afonso Title: Sales Associate

Customer Address: Av. dos Lusiadas, 23
Sao Paulo
Brazil, 05432-043

Shipping Address: Av. dos Lusíadas, 23
Sao Paulo
Brazil, 05432-043

Item	QTY	Price	Disc.	Total
Chef Anton's Gumbo Mixaaaa	20	17	.00	340.00
Thuringer Rostbratwurst	15	99	.00	1,485.00
Maxilaku - NEW	15	16	.00	240.00
Original Frankfurter grune So?e - NEW	10	10	.00	104.00
ORDER TOTALS	**60**			**2,169.00**
SHIPPING				**79.70**
TOTAL AMOUNT				**2,248.70**

FIGURE 24.5
On the Order Details tab, you can view an order's details.

The SQL Expressions project demonstrates how to write an application for navigating through related tables in all possible ways (at least, in all meaningful ways). This application is unique in that it populates a DataSet at the client and uses the data for its calculations. After the data is downloaded to the client, not a single trip to the server will be required, and as you can guess, the code uses calculated columns and the `Select` and `Compute` methods of the DataTable class.

First, you must create a new DataSet, the NorthwindDataSet, with the following tables: Orders, Order Details, Customers, Employees, and Products. The relations between any two of these tables will be picked up by the DataSet Designer and added to the DataSet. I was interested in the sales of the Northwind Corporation, but I've included the Employees, Customers, and Products tables to display actual names (product, customer, and employee names) instead of meaningless IDs. When the form is loaded, the code adds a bunch of calculated columns to the DataSet's tables, as shown here:

[Order Details].Subtotal A simple calculated column for storing each detail line's total:

```
If NorthwindDataSet1.Order_Details.Columns _
        ("DetailSubtotal") Is Nothing Then
     NorthwindDataSet1.Order_Details.Columns.Add _
        ("DetailSubtotal",
        System.Type.GetType("System.Decimal"), _
        "(UnitPrice * Quantity) * (1 - Discount)")
End If
```

Orders.OrderTotal A calculated column with aggregates for storing the order's total:

```
If NorthwindDataSet1.Orders.Columns("OrderTotal") _
        Is Nothing Then
    NorthwindDataSet1.Orders.Columns.Add("OrderTotal", _
      System.Type.GetType("System.Decimal"), "SUM(Child.DetailSubtotal)")
End If
```

Customers.CustomerTotal A calculated column with aggregates for storing the customer's total:

```
If NorthwindDataSet1.Customers.Columns _
      ("CustomerTotal") Is Nothing Then
    NorthwindDataSet1.Customers.Columns.Add _
      ("CustomerTotal", _
       System.Type.GetType("System.Decimal"), _
       "SUM(Child.OrderTotal)")
End If
```

Products.ItemsSold A calculated column with aggregates for storing the number of items of the specific product that were sold:

```
If NorthwindDataSet1.Products.Columns _
        ("ItemsSold") Is Nothing Then
    NorthwindDataSet1.Products.Columns.Add _
        ("ItemsSold", _
        System.Type.GetType("System.Int32"), _
        "SUM(Child(FK_Order_Details_Products).Quantity)")
End If
```

Employees.Employee A simple calculated column for storing the employee name as a single string (instead of two columns):

```
If NorthwindDataSet1.Employees.Columns _
        ("Employee") Is Nothing Then
    NorthwindDataSet1.Employees.Columns.Add _
        ("Employee", _
              System.Type.GetType("System.String"), _
              "LastName + ' ' + FirstName")
End If
```

Notice that the code checks to make sure that the column it's attempting to add to the DataTable doesn't exist already. In addition, the `Quantity` column's type is changed from Integer to Decimal. This conversion is necessary because the `Quantity` column is used in aggregates later in the code, and the result of the aggregate is the same as the column's type. Specifically, we're going to calculate the average quantity per order with the `AVG(Quantity)` function of T-SQL. If the argument of the function is an integer, the result will also be an integer, even though the average is rarely an integer. The following statement changes the type of a DataColumn in the DataSet:

```
NorthwindDataSet1.Order_Details.Columns("Quantity").DataType = _
          System.Type.GetType("System.Decimal")
```

See the section ''Aggregate Functions' Return Values'' a little later in this chapter for more on handling the return values of aggregate T-SQL values.

Then the code clears the DataSet and loads its tables with the following statements:

```
NorthwindDataSet1.Clear()
Me.Order_DetailsTableAdapter1.Fill(NorthwindDataSet1.Order_Details)
Me.OrdersTableAdapter1.Fill(NorthwindDataSet1.Orders)
Me.ProductsTableAdapter1.Fill(NorthwindDataSet1.Products)
Me.ShippersTableAdapter1.Fill(NorthwindDataSet1.Shippers)
Me.CustomersTableAdapter1.Fill(NorthwindDataSet1.Customers)
Me.EmployeesTableAdapter1.Fill(NorthwindDataSet1.Employees)
```

Now, on the Orders tab, we can look at the statements that perform the search operations. All the buttons on this tab pick up the appropriate values from the controls on the right side of the form and submit them to the `Select` method of the Orders DataTable. The Search By Order Date button executes the `Select` method shown in Listing 24.1 to retrieve the orders that were placed between the two specified dates.

LISTING 24.1: Retrieving Orders by Date

```
Private Sub bttnSearchOrderByDate_Click(...) _
        Handles bttnSearchOrderByDate.Click
   Dim selectedRows() As DataRow
   selectedRows = NorthwindDataSet1.Orders.Select( _
        "OrderDate>='" & _
         dtOrderFrom.Value & "' AND _
         OrderDate <= '" & _
         dtOrderTo.Value & "'", "OrderDate DESC")
   ShowOrders(selectedRows)
   lblOrderSearchCount.Text = "Search returned " & _
         selectedRows.Length.ToString & " rows"
End Sub
```

The first argument to the `Select` method is an SQL expression that selects orders by data, and the second argument determines the order in which the rows of the Orders table will be returned. When the various date values are substituted, the following `Select` method is executed:

```
Select("OrderDate>='1/1/1996' AND _
        OrderDate <= '8/31/1996'", OrderDate DESC")
```

The arguments passed to the `Select` method are identical to the `WHERE` and `ORDER BY` clauses of a `SELECT` statement you'd execute against the database with the Command object. The `Select` method, however, will act on the data in the DataSet at the client; it will not contact the server. If you need live, up-to-the-minute data, you should execute your queries directly against the database. But for many reporting tools, you can move all the relevant information to the client and work with it without additional trips to the server. The sample application shown in this section is a typical example of the type of applications that will benefit the most from working with a disconnected DataSet.

Real World Scenario

Should We Move Large Tables to the Client?

You'll probably wonder how wise it is to load a large chunk of the database to the client. In some cases, it makes sense to move a lot of data to the client. If a manager needs all this information, for example, the data will be moved to the client, either in small pieces or as a whole.

You should impose some limits to the amount of data you move to the client as always, such as limiting the orders by date or the customers by geographical area, but still you'll end up downloading a whole lot of data to the client. Think of these applications as reports: If you need to print a detailed sales report, you'll move all the required information to the client. Besides, this type of application is used by decision makers and is not used on a daily basis either.

As long as you don't move all the customers and products to the client every time the user must select a client or a product to issue an invoice, your applications won't cause network traffic problems. In other situations, you can afford to move large result sets to the client, because the application will not connect to the server again for a long time — not to mention that you can persist the DataSet at the client and reuse the same data in a later session.

An advantage of moving large sets of data to the client is that we free the server by moving a lot of operations that would otherwise be executed by SQL Server to the client. Again, this makes sense only if the user of the application is going to work for a while with the data and won't be populating the client DataSet every few minutes.

You can also persist the DataSet to a disk file and allow managers to examine the sales figures of last month, or last quarter, for as long as they wish on their workstations, even on their laptops. After the DataSet has been saved to a local file, the end user can disconnect from the database and work with the data in a totally disconnected fashion. With a powerful client, this approach is even more efficient than executing queries against the database as needed, especially if the database server is overwhelmed by other tasks that must be performed in a connected fashion.

Selecting and Viewing an Order's Details

Let's continue with the other operations of the application. When you double-click an order on the Orders tab, you'll see its details in a ListView control as usual, in the Order Details tab. The application's code extracts the ID of the selected order and passes it as an argument to the `DisplayOrder()` subroutine, which populates the controls on the Order Details tab.

The code of the `DisplayOrder()` subroutine is a bit lengthy, so I'll explain the basic operations here. You can open the project to see the entire listing of the subroutine. The `DisplayOrder()` subroutine starts by selecting the details of the specified order with the following statements:

```
Dim selectedRows() As DataRow
selectedRows = NorthwindDataSet1.Order_Details.Select( _
       "OrderID=" & OrderID.ToString, "ProductID")
```

Then it populates the customer controls on the tab by calling the `ShowOrderCustomer()` subroutine, passing as an argument the order's ID. The `ShowOrderCustomer()` subroutine retrieves the order's row from the Orders table by calling the `FindByOrderID` method. This method is exposed automatically by the DataSet, because typed DataSets always provide a method to find rows by their ID value. Then it retrieves the customer row from the Customers table by calling the `FindByCustomerID` method of the Customers table. After reading the customer row, it can populate the appropriate controls on the tab.

```
Dim custOrder As NorthwindDataSet.OrdersRow = _
     NorthwindDataSet1.Orders.FindByOrderID(OrderID)
Dim customer As NorthwindDataSet.CustomersRow = _
     NorthwindDataSet1.Customers.FindByCustomerID(custOrder.CustomerID)
```

Back to the `DisplayOrder()` subroutine: After the customer data have been displayed, the `DisplayOrder()` subroutine goes through the *`selectedRows`* array, which contains the order's detail lines: one DataRow object per element. Note that the `Select` statement returns an array of generic DataRow objects, not a specific row object such as `NorthwindDataSet.Order_DetailsRow`. The code in the `DisplayOrders()` subroutine iterates through the entire array, casts each element of the array to the specific type, and displays its fields in the *`lvOrderDetails`* ListView control of the Order Details tab. Listing 24.2 shows only the statements that retrieve the detail fields and create a new ListViewItem, which is then added to the ListView control.

Listing 24.2: Displaying the Items of a Detail Line

```
Dim Product As String = _
     NorthwindDataSet1.Products.FindByProductID(CType(selectedRows(i), _
     NorthwindDataSet.Order_DetailsRow).ProductID).ProductName
Dim Qty As Integer = CType(CType(selectedRows(i), _
     NorthwindDataSet.Order_DetailsRow).Quantity, Integer)
Dim Disc As Decimal = CType(CType(selectedRows(i), _
     NorthwindDataSet.Order_DetailsRow).Discount, Decimal)
Dim Price As Decimal = CType(CType(selectedRows(i), _
     NorthwindDataSet.Order_DetailsRow).UnitPrice, Decimal)
LI.Text = Product
LI.SubItems.Add(Qty.ToString("#,###"))
LI.SubItems.Add(Price.ToString("#,###"))
LI.SubItems.Add(Disc.ToString("#,###.00"))
```

The Order Details table contains product IDs, which are of no interest to the user, so the first statement extracts the *`ProductID`* field of the current detail row and passes it as an argument to the `FindByProductID` method of the Products DataTable. This method returns the matching row of the Products DataTable, and the code uses it to display the product's name. The statements that extract the other detail fields (quantity, price, and discount) are trivial.

The Customer Orders and Employee Orders tabs are similar: They display a list of customers and employees, respectively, and every time the user selects a customer or employee in the appropriate list, the code displays the selected item's orders (the orders placed by a customer or the orders made by an employee, respectively) in a ListView control. At the same time, it displays a summary for the selected entity in a TextBox control, as you can see in the corresponding figures. The most interesting piece of code in these two pages is the statements that calculate the summary data. The statements of Listing 24.3 calculate the average order quantity, the average order amount, and the total order amount for the selected customer.

LISTING 24.3: Calculating Customer Averages

```
avgQuantity = _
 CType(NorthwindDataSet1.Order_Details.Compute("AVG(Quantity)", _
     "Parent(FK_Order_Details_Orders).CustomerID='" & _
     CustomerID & "'"), Decimal)
avgOrder = CType(NorthwindDataSet1.Orders.Compute( _
     "AVG(OrderTotal)", "CustomerID='" & _
     CustomerID & "'"), Decimal)
totalQuantity = CType(NorthwindDataSet1.Order_Details. _
     Compute("SUM(Quantity)", _
     "Parent(FK_Order_Details_Orders).CustomerID='" & _
     CustomerID & "'"), Integer)
Total = CType(NorthwindDataSet1.Orders.Compute( _
     "SUM(OrderTotal)", "CustomerID='" & _
     CustomerID & "'"), Integer)
```

The first statement uses the `Compute` method of the Order_Details DataTable to compute the average of the `Quantity` column over the subset of rows specified by the second argument (the underscore was inserted by the DataSet designer, because the actual table name contains a space). This argument limits the rows of the Order_Details DataTable to the ones that belong to the orders with *`CustomerID`* equal to the ID value of the selected customer. The expression `Parent(FK_Order_Details_Orders)` returns the rows of the Orders table that are related to the rows of the Order_Details DataTable with the `FK_Order_Details_Orders` relation. From these rows, we select the ones with the specified *`CustomerID`* field. The result of the `Compute` method is an object, which is cast to a Decimal value with the `CType()` function.

Without the ability to evaluate SQL expressions over the client's DataSet data, you'd have to iterate through the rows of the Customers table, locate the child Order rows for each customer, and then the Order_Detail rows of each order, calculate the line totals, and add them. This approach takes a lot of complicated code, is not nearly as efficient, and (most important, in my view) is not elegant. In my experience, an elegant approach is usually more efficient, not to mention that it simplifies the application's code overall and is easier to maintain. Of course, if you're not familiar with SQL expressions and the `Compute` method of the DataTable object, you'll find the preceding code hard to understand and even harder to apply to other situations.

If you still consider the preceding statements cryptic, consider the equivalent SQL statement you'd execute in SQL Server's Management Studio (or Query Analyzer, for users of SQL Server 2000) and try to map its clauses to the arguments of the `Compute` method. The following SQL statement returns the average and total order quantities for a specific customer:

```
SELECT AVG(Quantity)
FROM [Order Details] INNER JOIN Orders
ON [Order Details].OrderID = Orders.OrderID
WHERE Orders.CustomerID = 'BLAUS'
```

In effect, the `Parent()` function corresponds to the `INNER JOIN` clause, and the `AVG(Quantity)` function is the `SELECT` statement's selection list. The join operation in the SQL expression is identified by the argument to the `Parent()` function: The `FK_Order_Details_Orders` relation associated the rows of the Order_Details DataTable with the rows of the Orders DataTable, just as the `JOIN` operator does in the SQL statement.

Aggregate Functions' Return Values

Let me digress here for a moment to explain a tricky aspect of the aggregate functions. If you execute the preceding SQL statement, the result that it will return for the specific customer is 26. The SQL Expressions application will report a slightly different (and more likely) value: 26.12.

Please recall that we had to change the data type of the *`Quantity`* column in the Order_Details DataTable from Integer to Decimal to get the correct results. The `AVG()` function returns a value of the same type as the column specified in its argument list. The *`Quantity`* column is an integer value, and so is the average over this column. This is a technicality you can easily forget and, as a result, get the wrong results from your database. Chances are that you will calculate the average quantities for many customers and notice that all results are integers, a highly unlikely situation for averages. At any rate, one could easily think that the query worked fine because it returned a result that's not ''unrealistically'' incorrect. To fix the previous SQL statement, you must cast the column you're averaging to the desired type, as shown here:

```
SELECT AVG(CAST(Quantity AS Numeric(8,2)))
FROM [Order Details] INNER JOIN Orders
ON [Order Details].OrderID = Orders.OrderID
WHERE Orders.CustomerID = 'BLAUS'
```

This SQL statement will return the same result as the SQL Expressions application.

The Top Products And Customers button on the Best Sellers tab retrieves the 10 products with the largest sales and the 10 best customers in terms of revenue they generated for the corporation. The results are displayed in two ListView controls, `lvBestProducts` and `lvBestCustomers`. The code behind this button is quite short and is shown in Listing 24.4.

Listing 24.4: Retrieving the Best-Selling Products and Best Customers

```
Private Sub bttnBestSellers_Click(...) _
            Handles bttnBestSellers.Click
    Dim selectedRows() As DataRow
    selectedRows = _
       NorthwindDataSet1.Customers.Select("", "CustomerTotal DESC")
    Dim i As Integer
    Dim customerItems As Integer
    Dim LI As ListViewItem
    lvBestCustomers.Items.Clear()
    For i = 0 To 9
      LI = New ListViewItem
      Dim custRow As NorthwindDataSet.CustomersRow = _
          CType(selectedRows(i), NorthwindDataSet.CustomersRow)
      LI.Tag = custRow.CustomerID
      LI.Text = NorthwindDataSet1.Customers. _
          FindByCustomerID _
```

```
                (custRow.CustomerID). _
                 CompanyName.ToString
          Dim custTotal As Decimal = Convert.ToDecimal( _
              NorthwindDataSet1.Customers. _
              FindByCustomerID( _
                  custRow.CustomerID). _
                  Item("CustomerTotal"))
          LI.SubItems.Add(custTotal.ToString("#,###.00"))
          customerItems = _
              CType(NorthwindDataSet1. _
              Order_Details.Compute("SUM(Quantity)", _
               "Parent(FK_Order_Details_Orders). _
               CustomerID='" & _
               custRow.CustomerID & "'"), Integer)
          LI.SubItems.Add(customerItems.ToString("#,###"))
          lvBestCustomers.Items.Add(LI)
        Next

        selectedRows = NorthwindDataSet1.Products. _
                 Select("", "ItemsSold DESC")
        lvBestPoducts.Items.Clear()
        For i = 0 To 9
          LI = New ListViewItem
          Dim prodRow As NorthwindDataSet.ProductsRow = _
                  CType(selectedRows(i), NorthwindDataSet.ProductsRow)
          LI.Text = NorthwindDataSet1.Products.FindByProductID _
                  (prodRow.ProductID).ProductName.ToString
          Dim itemTotal As Integer = Convert.ToInt32( _
                  NorthwindDataSet1.Products.FindByProductID( _
                  prodRow.ProductID).Item("ItemsSold"))
          LI.SubItems.Add(itemTotal.ToString("#,###"))
          lvBestPoducts.Items.Add(LI)
        Next
    End Sub
```

The code starts by selecting the rows of the Customers DataTable sorted by the `CustomerTotal` column in descending order. Then it iterates through the top 10 rows, casts each row to the `NorthwindDataSet.CustomersRow` type (so we can work with a strongly typed object), and displays the appropriate fields on the *`lvBestCustomers`* ListView control. Next it populates the *`selectedRows`* array again, this time with the rows of the Products table, sorted by the *`ItemsSold`* column in descending order. The top 10 rows are the best-selling products, and the code displays the appropriate fields in the *`lvBestProducts`* ListView control.

Note how the code uses the `FindByProductID` method to retrieve the matching product row and read its *`ItemsSold`* column. Even though we're working with a typed DataSet, the *`ItemsSold`* column was added to the DataTable control and was not available when the DataSet was created, which explains why you won't see this member when you enter the name of the variable of the ProductRow type followed by a period.

Open the SQL Expressions project, explore its operations, and take a closer look at its code. I've included many comments to explain the application's operation and ease your way through the code. (This application has more comments than a college student's homework!) It's a functional report-style application, and you can write similar applications for your company's data. You'll be really hard-pressed to achieve this level of functionality with a reporting tool, not to mention that you can add printing capabilities to the application by using the techniques discussed in Chapter 20, ''Printing with VB 2008.''

The idea is to duplicate a small section of the database in the client's computer memory by populating a DataSet with the tables we want to work with (or segments of the tables). The sample application uses the same data over and over again, so it makes sense to move all the required data to the client, because this will save our application from making a trip to the server every time the user clicks one of the items on the form (a customer/employee, an order, and so on). Tables have a tendency to grow quickly, so you should always try to limit the number of rows you move to the client, as well as the number of columns you select from each table. For instance, you can work with the orders of all customers from Germany, or the customers that have placed an order in the last two months, and so on.

The Bottom Line

Use SQL to query DataSets. Although DataSets resemble small databases that reside in the client computer's memory, you can't manipulate them with SQL statements. However, it's possible to query their tables by using the `Select` method and SQL-like criteria. The `Select` method filters the rows of the table to which it's applied and returns an array of DataRow objects, which you can use as a data source for data-bound controls.

Master It How would you select the rows of interest from a DataTable?

Add calculated columns to DataTables. Sometimes you'll need to update a column's value based on the values of other columns. For example, you may wish to maintain a column in the Orders DataTable with the order's total. For this column to be meaningful, its value should be updated every time a related row in the Order Details DataTable is modified. In the actual database, you'd do this with a trigger, but we want to avoid adding too many triggers to our tables because they slow all data-access operations, not to mention that you'll be duplicating information. To maintain totals in a DataSet, you can add calculated columns to its DataTables. You specify a formula for the calculated column, and every time one of the columns involved in the formula changes, the calculated column's value changes accordingly.

Master It Add a calculated column to the Customers DataTable that combines the first and last columns.

Compute aggregates over sets of rows. To calculate aggregates over sets of rows in a DataTable, use the `Compute` method, which accepts two arguments: an SQL-like aggregate function and a filtering expression, which is similar to an SQL `WHERE` clause. The aggregate function isn't limited to columns of the table to which it's applied; you can use the `Child()`function to access the current row's child row in a given relation, and the `Parent()` function to access the current row's parent row(s) in a given relation, which is passed to the method as an argument.

Master It Show the statement for adding a new column to the Orders table with each order's total.

Chapter 25

Building Web Applications

Developing web applications in Visual Studio 2008 has many similarities to developing traditional desktop applications. You drag and drop controls onto a form and build your business logic by using your language of choice — in our case, Visual Basic 2008.

However, as you will see, there are also many differences. There are underlying technologies that you, the developer, should have a solid understanding of, additional control sets to work with, and some fundamental differences in the way that the standard controls behave.

In this chapter, you will learn how to do the following:

- Create a basic XHTML/HTML page
- Format a page with CSS
- Set up a master page for your website
- Use some of the ASP.NET intrinsic objects

Developing for the Web

In the early days of web development (not all that long ago!), a developer could earn big money creating what were essentially online brochures by using a basic knowledge of Hypertext Markup Language (HTML) and some simple design skills.

These days we expect a great deal from our websites and web applications. Entertainment sites are now fully equipped to engage the visitor with rich user interfaces incorporating a wide range of visual and aural experiences. Members of the corporate world expect their virtual presence to mirror their full range of business practices.

In addition, web development, although still seen as a specialized area, is now part of the corporate mainstream, and the developer is expected to be well versed across a range of technologies and techniques.

The modern web application combines a wide range of sophisticated technologies grafted onto the HTTP/HTML backbone. Cascading Style Sheets (CSS) are used to control the layout and appearance of a website. Data is managed with the use of Extensible Markup Language (XML) and back-end databases such as SQL Server, while rich user interfaces are developed using XML, JavaScript, and other technologies such as Adobe Flash. AJAX, a clever implementation of existing technologies, combines XML, JavaScript, and asynchronous technologies to enable the developer to create online applications that exhibit traditional desktop behavior. XML web services, multimedia content, RSS feeds, and the use of microformats to assist data aggregators have all become typical features of modern websites.

In addition, the developer can now expect a website to be accessed by more than just a desktop computer. Mobile phones, PDAs, and other small form factor devices are all used to access the web in the 21st century. Websites, to be truly ubiquitous, are increasingly expected to be able to dynamically render their content into an appropriate format.

Visual Studio 2008 provides a range of tools that enable the modern developer to meet the demands of website creation from the comfort of a Visual Basic environment. Database connectivity is simplified from the traditional complexity of hand-coding scripted server-side pages, and sites can be developed that dynamically render to suit a wide range of devices without the need to build multiple versions of an individual site. By compiling much of the code used to drive a site, many of the security issues that plagued scripted sites are avoided.

Typically, a modern website or web application relies on code that is executed both at the client (web browser) and server (web server) ends. In addition, there may be a whole range of other services provided by other servers on the hosting network such as media or databases, and even services sourced from other websites. Visual Studio 2008 provides the tools to tie all this together.

This chapter gives an overview of the core technologies that drive the modern web application and demonstrates the basic tools available to the developer in Visual Studio 2008.

I will begin with some basic concepts. If you are already familiar with HTML, JavaScript, and server technologies, you may wish to skip ahead to material that is new to you, such as the ''Cascading Style Sheets'' section.

Understanding HTML and XHTML

HTML is essentially a language to describe text formatting and enable linking of documents (web pages) delivered over the Web. HTML has grown since its original inception but is still fundamentally limited. The area where it does excel is in its ability to act as a framework in which other technologies such as JavaScript can be embedded.

Extensible HTML (XHTML) is the latest incarnation of HTML. It was developed by the World Wide Web Consortium (W3 C) to bring HTML syntax into line with other markup languages such as XML. Most of the tags from HTML 4 (the most recent update to HTML) remained the same, but much stricter syntax rules apply. The basic changes are as follows:

- XHTML is case sensitive, and all tags are in *lower case*.
- All tags must be closed. You can no longer get away with using multiple `<p>` tags without corresponding closing `</p>` tags. This includes tags such as `<img>` that previously had no corresponding closing tag. Close these tags with a trailing backslash before the final angle bracket. For example:

  ```
  <img src = 'my_picture.jpg' alt = 'picture' \>
  ```

- All tag attributes must be enclosed in quotation marks (either single or double).
- All pages must include an XHTML `!DOCTYPE` definition and an XML version declaration.
- JavaScript must also conform to case syntax — for example, `onmouseover` *not* `onMouseOver`.

The W3 C encourages developers to use XHTML over HTML. However, for practical purposes, web browsers still support HTML, and you can get away with not updating older sites and continuing to work with HTML's lack of strict syntax. See the following sidebar if you wish to upgrade older sites.

Upgrading from HTML to XHTML

Converters exist for porting your HTML sites to XHTML. There are also tools to assist in the process manually. W3 C provides an online validator at `http://validator.w3.org`.

You can use the validator to initially ensure that your pages conform to the HTML 4 specification and that they all contain a `!DOCTYPE` definition such as the following:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/html4/strict.dtd">
```

After your pages are validated for HTML 4, you will need to add the XML declaration to the top of each page:

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

Then convert the `!DOCTYPE` definition to the XHTML version:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

Finally, modify the `<HTML>` tag to read as follows:

```
<HTML xmlns="http://www.w3.org/1999/xhtml">
```

Assuming that you have made all the correct syntax changes, run the validator again and see how your page performs.

Working with HTML

As a Visual Studio programmer, knowledge of HTML and XHTML can prove invaluable when sorting out the inevitable formatting and design issues that arise when developing complex web applications. In addition, understanding the technologies involved can aid in optimizing the interactions between server and client.

To keep things simple, I will dispense with some of the finer points of XHTML and focus mainly on ''straight'' HTML. This section is meant as a basic HTML primer (and is by no means comprehensive), so feel free to skip ahead if you are already familiar with the language.

More information on HTML can be found easily on the Web or in *Mastering Integrated HTML and CSS* by Virginia DeBolt (Sybex, 2007). Useful websites for tutorials include `www.w3schools.com` and `www.htmlcodetutorial.com`.

You can use any standard text editor to author your HTML. Notepad is fine. More-sophisticated tools that indent and highlight code, such as Notepad2 or EditPad Pro, are available on the Web. EditPad Pro is a commercial application available from `www.editpadpro.com`. Notepad2 is freeware and available from `www.flos-freeware.ch/notepad2.html`.

Remember to save your files with an `.html` file extension for them to be recognized as HTML pages.

Page Construction

HTML pages have two main sections nested between the opening and closing `<html>...</html>` tags.

The first section is known as the *head area*, and it is used to contain information not usually displayed on the page. (The main exception to this is the page title.) The head area is created by using the `<head>...</head>` tags and contains meta-information about the page such as the `!DOCTYPE` definition, author details, keywords, and the like. It is also used to hold style sheet information and scripts that may be called later in the page.

The second section of the page is the *body*, and it contains information that is typically displayed on the page in a web browser. The body is declared by using the `<body>...</body>` tags.

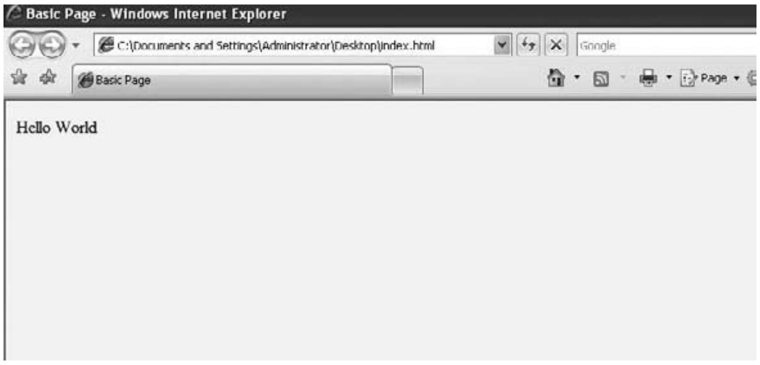
HTML tags are used to describe the formatting or nature of the information contained within the opening and closing tags. Tags may also contain *attributes*, which are used to apply further information to the content between the opening and closing tags. For example, the body tag can use the attribute `bgcolor` to set the background color of the web page. The syntax for setting a page color to blue is `<body bgcolor ="blue">` .

A basic page may appear as shown in Listing 25.1. Some long lines are wrapped here in print, but you can leave them all on one line in your code.

Listing 25.1: A Basic HTML Page

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" _
"http://www.w3.org/TR/html4/strict.dtd">
<html>
   <head>
      <title>Basic Page</title>
      <meta name="description" content="basic page" />
   </head>
   <body bgcolor="cornsilk">
      <p>Hello World</p>
   </body>
</html>
```

Save the page as `index.html`. You must include the `.html` as the file extension. Figure 25.1 illustrates how such a page might appear in a web browser.


Basic Page - Windows Internet Explorer
C:\Documents and Settings\Administrator\Desktop\index.html
Google
Basic Page
Page

Hello World

Figure 25.1
A simple web page running in a web browser window

The `<title>` tag enables the title of the page to appear in the title bar of Internet Explorer. The `<meta>` tag provides a description phrase that can be accessed by search engines. The `bgcolor` attribute of the `<body>` tag sets the background color to blue, and the `<p>` tag is used to denote a simple block of text.

Note that I have used the `!DOCTYPE` definition for HTML 4. Also note that not closing the `<title>` tag correctly can cause the rest of the page to break.

Text Management

There are a range of text management tags, including the previously mentioned `<p>` tag. The principal tags are as follows:

- The heading tags,`<h1>...</h1>`, `<h2>...</h2>` through to `<h6>...</h6>`, are used to control the size of text. `<h1>` is the largest format.
- The font tag, `<font>...</font>`, has a number of attributes including `face` for font type, `color` for text color, and `size` for font size. Font sizes range from 1 to 7, where 7 is the largest. An example of the tag's usage is `<font face ="verdana" color ="red" size ="3">Hello World</font>`.
- The small and big tags — `<small>...</small>`, `<big>...</big>` — can be used to quickly adjust the relative size of text.

Styles can be managed with tags such as the following:

- Bold: `<b>...</b>`
- Underline: `<u>...</u>`
- Italic: `<i>...</i>`
- Strong: `<strong>...</strong>`

Another useful tag when working with text is the line break tag: `<br>`. In HTML, this tag does not require a closing tag.

Spaces between text can be generated and controlled with more precision than simply relying on the client browser to insert your preferred amount of white space by using the following special character: `&nbsp;`.

A number of other special characters exist to accommodate symbols such as quotes, question marks, or copyright symbols. A comprehensive list of HTML tags and their attributes can be found at `www.w3schools.com/tags/default.asp`. You can also refer to the W3C specification for HTML 4.01 at `www.w3.org/TR/html401/`. The XHTML 1.1 specification can be found at `www.w3.org/TR/2007/CR-xhtml-basic-20070713/`.

Horizontal Rules

The `<hr>` tag can be used to draw a line across the page. Its attributes include `align`, `noshade`, `size`, and `width`. Width can be declared as a percentage or as an exact amount in pixels.
In HTML 4, there is no closing tag.

Images

Images can be added to web pages by using the `<img>` tag. This tag is not typically closed under HTML 4. Attributes for this tag include the following:

- The path to the image, `src`, which can be relative or absolute. (Required.)

- A text alternative to the image, `alt`, which is normally recommended for accessibility.
- `align` is used to align an image on a page and to wrap text around the image.
- `border` is used to create a border around the image.
- `width` and `height` are used to help the page load more quickly, and can also be used to scale an image.
- `usemap` is used to create image maps.

A typical use of the `<img>` tag might be the following:

```
<img src='images/myimage.jpg' border='0' width='150' _
height='150' align='left' alt='Test Image'>
```

For use in web pages, images must be in one of the following formats: GIF, PNG, or JPG. You usually use the GIF or PNG formats for drawings or line art, and the JPG format for photographs.

Links

Links can be created on web pages that link to other web pages within the site, other websites, other types of documents, e-mail, or other locations within the host page. Links are created by using the `<a>...</a>` tag.

Typically, the `<a>` tag is used with the `href` attribute to define the destination of the link. For example:

```
<a href='http://www.microsoft.com'>Microsoft</a>
```

The text contained between the tags (*Microsoft*) is what appears as the link on the page. The text can be formatted by using the `<font>` tag inside the `<a>` tags.

Other attributes commonly used include `target` (used inside framesets) and `name` (used for setting up in-page links such as tables of contents).

Embedding Media

Media objects such as Windows Media Player, Apple's QuickTime, and Flash can be embedded in a page by using the `<embed>` tag. At its very simplest, the tag can be made to work by simply specifying the source file for the media and the display size, and then trusting the browser to have the required plug-in and to be able to sort it out. For example:

```
<embed src='multimedia/myvideo.avi' height='200' width='200'></embed>
```

At a more sophisticated level, you can specify a range of options including the type of plug-in, the controls to display, whether it should start automatically, and loop properties.

Comments

To insert comments into your HTML, use `<!-- ... -->`. For example:

```
<!-- This is a comment -->
```

Comments enclosed in this tag are not displayed within the web page.

Scripts

The <script>...</script> tag can be used to insert non-HTML script code such as JavaScript into your pages. Scripts can be written into the header area and called from the body or used directly in the body of the page. Before support for more-recent HTML versions and before JavaScript could be found in virtually every web browser, developers would typically comment out the code by using <!-- ... --> to prevent the code from appearing in the browser page. This is still common practice, although it is no longer usually necessary.

A simple example of the script tag's usage is as follows:

```
<script language='javascript'>
   function mygreatscript(){
      etc etc
   }
</script>
```

Lists

Bulleted and numbered lists can be displayed in a web page by using the list tags <ul>...</ul> for a bulleted list or <ol>...</ol> for a numbered list. Individual items within the list are denoted by using the <li>...</li> tags. An example of creating a bulleted list is as follows:

```
<ul>
   <li>Item 1</li>
   <li>Item 2</li>
</ul>
```

Tables

Tables are used extensively in HTML, not only to display data but also to reassemble sliced images (a technique for minimizing the file size of large images) and to format pages. However, using tables to format pages is no longer recommended by the W3C. For accessibility reasons, technologies such as CSS are the recommended method for creating sophisticated layouts.

Tables are made of rows of cells. When constructing your table, you need to consider the desired width of the table, the number of rows, and the number of columns required. Other factors that you may wish to take into account are the padding between cells, and the padding between the border of a cell and its contents.

Another important consideration is that a badly constructed table is one of the few things that can truly break an HTML page. You must ensure that all your tags are correctly closed. Tables can be nested within one another, but excessive nesting can place undue strain on the rendering engine of the browser and cause unexpected results.

A range of tags are used to create a typical table, and each has its own family of attributes:

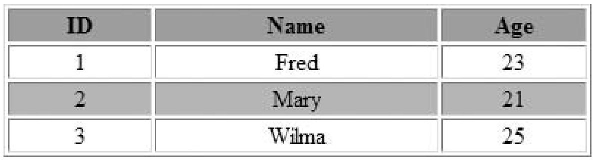
- The <table>...</table> tag acts as the major framework for the table.
- <tr>...</tr> tags are used to create rows.
- Within a given row, <td>...</td> tags are used to create the individual cells. The number of cells defined in the first row sets the number of columns in the table. This is important to consider in subsequent rows if you wish to add or subtract from the number

of cells. The `rowspan` and `colspan` attributes are used to alter the number of columns and rows employed at various points in the table. For example: `colspan = '2'` will force a cell to span over two columns.

- For headings, you can use `<th>...</th>` tags to create cells in the first row. These offer a slightly different format than the `<td>` cells offer.

The following code snippet demonstrates a simple data table of three rows (one header) and three columns. The resulting table is shown in Figure 25.2.

```
<table width='400' border='1'>
   <tr bgcolor='silver'>
      <th width='100'>ID</th>
      <th width='200'>Name</th>
      <th width='100'>Age</th>
   </tr>
   <tr align='center'>
      <td valign='middle'>1</td>
      <td valign='middle'>Fred</td>
      <td valign='middle'>23</td>
   </tr>
   <tr align='center'>
      <td valign='middle' bgcolor='lightblue'>2</td>
      <td valign='middle' bgcolor='lightblue'>Mary</td>
      <td valign='middle' bgcolor='lightblue'>21</td>
   </tr>
   <tr align='center'>
      <td valign='middle'>3</td>
      <td valign='middle'>Wilma</td>
      <td valign='middle'>25</td>
   </tr>
</table>
```



ID	Name	Age
1	Fred	23
2	Mary	21
3	Wilma	25

FIGURE 25.2
A simple data table

The following code snippet illustrates how a table might be used to reconstruct a sliced image. Note the use of the `align` attribute to set horizontal alignment and the `valign` attribute to set vertical alignment:

```
<table width='400' border=0 cellspacing='0' cellpadding='0'>
   <tr>
      <td valign='bottom' align='right'><img src='image1.gif'></td>
      <td valign='bottom' align='left'><img src='image2.gif'></td>
```

```
    </tr>
    <tr>
       <td valign='top' align='right'><img src='image3.gif'></td>
       <td valign='top' align='left'><img src='image4.gif'></td>
    </tr>
</table>
```

Figure 25.3 illustrates how this image will appear in a web page (the left-hand image) and how the table cells come together to reassemble the image (the right-hand image).

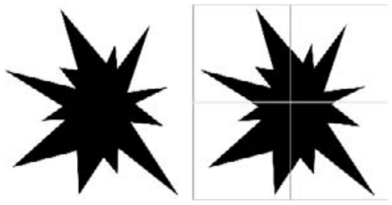


FIGURE 25.3
A sliced image reassembled using an HTML table

Page Formatting

Various methods can be used to format pages in HTML. They all have inherent limitations, but the `<div>` tag offers the most flexibility. The methods are as follows:

Flow format This relies on the browser to format the page according to the order of items in the HTML. Flow format is easy to implement, but of limited usefulness.

A table This is one of the more popular methods, although it is no longer officially recommended by the W3C for accessibility reasons.

Frames These effectively create the different parts of your page in separate HTML documents. You would use a frameset to reassemble them as a single page in the browser.

Inline frames (iFrames) These create a floating frame that can be placed within an HTML page. This is a popular method of displaying content from another website (such as a news feed) within your web page.

***<div>* tags** These can be used to precisely locate content on your page. Combined with CSS, `<div>` tags offer a powerful and flexible method of organizing a page. The output of ASP.NET is largely driven by `<div>` tags for layout purposes.

Later, when you look at Cascading Style Sheets, you will see how `<div>` tags can be used to lay out a page.

Forms and Form Elements

Forms are the traditional method by which users can communicate information back to the web server that is hosting the website being viewed. Information sent back by a form can then be processed in some way at the server, and the outcome can be dynamically incorporated into a new page that the user can view.

For example, a login page would likely use a form to collect the username and password from the user and return this information to the server for authentication before access to the rest of the site is granted. In addition, personal preferences for the site can be applied to the returned pages according to the stored preferences of the registered user.

A form is created by using the `<form>...</form>` tags. The commonly used attributes include the following:

Name Defines a unique name for the form.

Action Specifies the Uniform Resource Locator (URL), or Internet address, of the resource to process the form's response. (Required.)

Method Either `post` or `get` (default). This specifies the HTTP method used to return the form's data. The `get` method sends the data as part of the URL (limited to a maximum of 100 ASCII characters), and `post` sends the form's content in the body of the request.

Within the form, you create your HTML as usual. However, information that you wish to be processed by the form needs to be collected by using *form controls*. The controls that are available include the following:

Buttons Your form must have at least one button for submitting data. Another button that is commonly used clears the form's contents. Syntax for the buttons are as follows:

```
<input type='submit' value='Submit data'>
<input type='reset' value='Reset form'>
```

It is not necessary to include the `value` attribute because this sets the text that will appear in the button, and there are default text values of Submit and Reset.

You can use the following to create other buttons on your forms to run client-side scripts:

```
<input type='button' value='Mybutton' onclick='myscript'>
```

A more flexible control, however, is the `<button>` tag. This can be used anywhere on the HTML page to run scripts and can replace the traditional Submit and Reset buttons in your forms. It offers greater flexibility for formatting its appearance (especially when used with CSS). Its basic syntax is as follows:

```
<button type='button' name='name' onclick='myscript'>Click Here</button>
```

By using an image tag in place of *Click Here*, you can set an image to be the button. Syntax for using the button as a submit button is simply the following:

```
<button type=submit' >Submit</button>
```

Text The Text control enables the user to enter a single line of text. This can be set as a password field to mask the user's entry. The syntax is as follows:

```
<input type='text' name='identity of input data' _
 value='data to be initially displayed in field'>
```

The `name` attribute specifies the identity of the data to be processed at the server end (for example, the username). The `value` attribute displays text that you may wish to appear initially in the field (for example, *Type user name here*). You can also set other attributes such as `size` and `maxlength`. To create a password field, set the `type` attribute to `password`.

TextArea For larger amounts of text, use the `<textarea>` tag. Its syntax is as follows:

```
<textarea name='details' rows='10' cols='40' >Type your details here</textarea>
```

Note that this control requires a closing tag.

Lists To create lists, use the `<select>` tag. Lists can be either single select or multiple select, which is created by using the `multiple` attribute (simply typing ***multiple***). The `size` attribute specifies the number of rows to display. Omitting the `size` attribute renders the control as a drop-down combo box. The contents of the `value` attribute are returned to the server. Individual items are denoted by using the `<option>` tags. By typing ***selected*** within one of the option tags, that item is automatically highlighted in the list. The syntax for the tag is as follows:

```
<select name='items' size='4' multiple>
   <option value='1' selected>Chair</option>
   <option value='2'>Couch</option>
   <option value='3'>Arm Chair</option>
   <option value='4'>Lounge Chair</option>
</select>
```

Check boxes To create a check box, you use a variation on the `<input>` tag and set the `type` attribute to `'checkbox'`. To initially select a check box, type the attribute `checked`. The syntax is `<input type = 'checkbox' name = 'Check1' checked>`.

Radio buttons These are yet another variation on the `<input>` tag. Set the `type` attribute to `'radio'`. If you are using a set of linked radio buttons, type the same `name` attribute for each radio button in the set. The `value` attribute is used to return appropriate data when the radio button is selected. Here is the syntax:

```
<input type='radio' name='radioset' value='1' checked>
<input type='radio' name='radioset' value='2'>
<input type='radio' name='radioset' value='3'>
```

Hidden fields Hidden fields contain information that you may want to make the round-trip to the server, but that you do not want displayed on the client's web page. You can use this field to help maintain *state* (discussed later in this chapter in the ''Maintaining State'' section). Using this field is particularly useful when a client has disabled cookies or when the information is too long or sensitive to incorporate into the URL. For example, you may wish to maintain information gathered in previous forms from the client. ASP.NET uses hidden fields extensively. Here is the syntax:

```
<input type='hidden' name='name of information'
 value='information to be stored'>
```

Cascading Style Sheets (CSS)

Cascading style sheets offer a powerful method of controlling the format and layout of the pages and content of your websites. Styles can be written directly into your HTML pages or created in a separate text document with the `.css` extension. The advantage to the developer of using separate

CSS pages is that the format and layout of an entire site can be controlled from a single page. In large sites, consisting of tens or even hundreds of pages, this can be a huge time-saver as well as introducing a much higher level of consistency and reliability.

In addition, styles are applied sequentially and can override previously set styles. This enables the web developer to create specific styles for specific sections of the site that may modify the global settings. You can create and apply multiple style sheets in this manner and even write individual style settings onto individual pages if necessary. Styles can also be applied directly to individual elements within a page. As long as the desired settings are the last to be applied, the page will appear as required.

Syntax for CSS is quite different from syntax for HTML and is quite strict. You can apply styles directly to HTML tags (for example, you may wish to format the `<h1>` tag with a particular font and color) or set them up in their own classes that can be applied when required. For example, you may wish to create your own `<h8>` class.

An external style sheet is included in an HTML page by using the `<link>` tag, which is typically placed in the head area of the web page. The following example incorporates a style sheet titled `mystylesheet.css` from the `styles` directory into the web page:

```
<link rel='stylesheet' type='text/css' href='styles/mystylesheet.css'>
```

An internal style sheet can be created directly in the HTML page by using the `<style>`... `</style>` tags. Again, this is typically created in the head area of the document.

If you wish to create a style locally to a particular tag, you add the style attributes inside the tag. For example, to extend the style of the `<p>` tag, you would use the following:

```
<p style='font-size:18pt; color:red;'>
```

Formatting Styles with CSS

Listing 25.2 illustrates a sample style sheet. It demonstrates several simple style attributes that can be applied to text. You can use the styles directly in a web page by inserting the listing between `<style>`...`</style>` tags or you can use it externally by saving the listing out to a separate text document with a `.css` file extension. (for example, `mystylesheet.css`). Some long lines are wrapped here in print, but you can leave them all on one line in your code.

Listing 25.2: A Sample Style Sheet

```
h1 {font-weight: bold; font-size: 24pt; color:red; _
 background: silver; text-align: center;}
p {font-family: arial, sans serif; font-size: 120%;}
p.quote {font-face:verdana; font-size: 10pt; font-style: italic;}
a {text-decoration:none; color:blue;}
a:visited {text-decoration:none; color:blue;}
a:hover {text-decoration:none; font-weight: bold; _
 font-size: 120%; color:darkblue;}
a:active {text-decoration:none; color:blue;}
```

If you were to use Listing 25.2 as an external style sheet, you could link it to your web page by inserting `<link rel = 'stylesheet' type = 'text/css' href = 'mystylesheet.css'>`

somewhere in the head area of your web page. This also assumes that the style sheet is sitting in the same directory as your web page.

Points to note in Listing 25.2 include the following:

- ◆ The creation of a separate *quote* class for use with the `<p>` tag. To employ this class in your HTML, simply use `<p class = 'quote'>...</p>`.
- ◆ By setting styles for the various permutations of the `<a>` tag, I have also created a simple rollover effect for use with links. (The order in which these are applied is important for the rollover effect to work.)

Rollovers can be created by using other methods such as JavaScript, but CSS offers a simple way of globally controlling the effect. You can also create quite sophisticated-looking buttons around your links by using the formatting and style properties of CSS.

Page Formatting with CSS

CSS also can be used to define page formatting and layout for an HTML document. CSS is typically used to define and control `<div>` tags for this purpose, although you can also use it to set and control table properties.

Listing 25.3 demonstrates a CSS style sheet used to control basic formatting. Most of the items should be self-explanatory. (I have used named colors for the background colors for purposes of clarity. Usually it is preferable to use the hexadecimal equivalents.)

LISTING 25.3: Style Sheet to Control Page Layout

```
.title{
 height:80px;
 background:lightblue;
 margin:5px 10px 10px 10px;
 text-align: center;
 }

.menu{
 position: absolute;
 top: 110px;
 left: 20px;
 width: 130px;
 background: silver;
 padding: 10px;
 bottom: 20px;
 }

.content{
 background: lightblue;
 padding: 30px;
 position: absolute;
 top: 110px;
 bottom: 20px;
 left: 180px;
 right: 20px
 }
```

I have created three classes — *title*, *menu*, and *content* — to describe the three main areas of my page. The size of the area can be defined as well as its precise location. In the case of the title class, I haven't specified an exact location, and the title area will appear relative to where it is written into the code. Other properties of the areas can also be defined such as padding (distance between the area's border and its internal elements) and background color. We use the `margin` property to set the width of the title area by defining how far it is located from adjacent elements and the page border.

Using the `margin` property in this context can be a little confusing. If four values are listed, they refer to top, right, bottom, and left, respectively. However, listing just one value will apply it to all four borders. Listing two values will apply them to the top/bottom and right/left in combination. If there are three values listed, the missing values are taken from the opposite side. It is sometimes easier to refer specifically to the `margin-right`, `margin-top`, etc. properties.

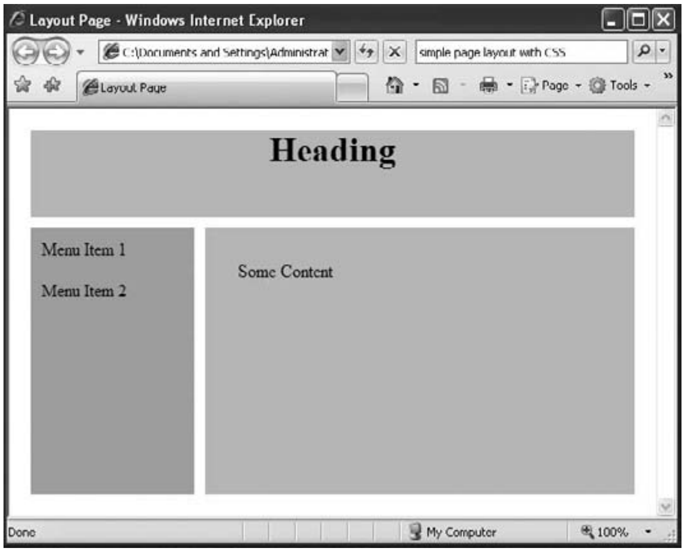
You can either embed Listing 25.3 into an HTML page or access it by using an external style sheet. Listing 25.4 demonstrates the code embedded into an HTML page and utilized to set the layout of the page. Some long lines are wrapped here in print, but you can leave them all on one line in your code.

LISTING 25.4: Using a Style Sheet to Set the Layout of a Web Page

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" _
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title>Layout Page</title>

<style>

.title{
 height:80px;
 background:lightblue;
 margin:5px 10px 10px 10px;
 text-align: center;
 }

.menu{
 position: absolute;
 top: 110px;
 left: 20px;
 width: 130px;
 background: silver;
 padding: 10px;
 bottom: 20px;
 }

.content{
 background: lightblue;
 padding: 30px;
 position: absolute;
 top: 110px;
 bottom: 20px;
 left: 180px;
 right: 20px
 }
</style>
</head>

<body>
   <div class='title'>
      <h1>Heading</h1>
   </div>
   <div class='menu'>
      <p>Menu Item 1</p>
      <p>Menu Item 2</p>
   </div>
   <div class='content'>
      <p>Some Content</p>
   </div>

</body>
</html>
```

Figure 25.4 illustrates how the layout from Listing 25.4 appears in a web page. Carefully look at the code and you will see how the individual layout classes are used inside the `<div>` tags to generate the layout structure of the page.

FIGURE 25.4
Listing 25.4 running as a web page


Layout Page - Windows Internet Explorer
C:\Documents and Settings\Administrat
simple page layout with CSS
Layout Page
Page
Tools
Heading
Menu Item 1
Menu Item 2
Some Content
Done
My Computer
100%

This is a very brief overview of CSS. For more-comprehensive coverage, please refer to *Mastering Integrated HTML and CSS* by Virginia DeBolt. There are also many online tutorials available, such as `www.w3schools.com/css/`.

JavaScript

You can embed JavaScript into your HTML pages to create interactive and dynamic elements at the client end. JavaScript can be used to create named functions inside the script tags that can be called later in the page. You can also use JavaScript attached directly to HTML elements.

Currently, 18 events specified in HTML 4 and XHTML 1 can be used as triggers to run individual scripts. Table 25.1 lists these events.

Table 25.1: Events Available for Use in HTML

Keyboard Events (Not Valid in Base, BDO, BR, Frame, Frameset, Head, HTML, iFrame, Meta, Param, Script, Style, and Title Elements)	
`onkeydown`	When a keyboard key is pressed
`onkeypress`	When a keyboard key is pressed and released
`onkeyup`	When a keyboard key is released
Mouse Events (Not Valid in Base, BDO, BR, Frame, Frameset, Head, HTML, iFrame, Meta, Param, Script, Style, and Title Elements)	
`onclick`	When an object is clicked with the mouse
`ondblclick`	When an object is double-clicked with the mouse
`onmousedown`	When the mouse is clicked on an object
`onmousemove`	When the mouse is moved
`onmouseover`	When the mouse is moved over the object
`onmouseout`	When the mouse is moved away from the object
`onmouseup`	When the mouse button is released
Form Element Events (Valid Only in Forms)	
`onchange`	When the content of the field changes

TABLE 25.1: Events Available for Use in HTML *(CONTINUED)*

`onsubmit`	When the form is submitted by clicking the submit button
`onreset`	When the form is reset by clicking the reset button
`onselect`	When some content of the field is selected
`onblur`	When an object loses focus
`onfocus`	When an object gains focus as the user selects the object
WINDOW EVENTS (VALID ONLY IN BODY AND FRAMESET ELEMENTS)	
`onload`	When the page is loaded
`onunload`	When the page is unloaded

The following code snippet gives an example of using JavaScript to create a rollover effect on a link:

```
<a href="newpage.html" >
<font color='blue' face="verdana" onmouseover="this.style.color = 'lightblue';"
onmouseout="this.style.color = 'blue';" size=1>New Page</font></a>
```

This script sets the link color to blue. Rolling the mouse over the link changes it to a light blue. Moving the mouse off the link resets it to the normal blue.

Listing 25.5 demonstrates how a JavaScript function can be embedded into a web page and then called from a button press. In this example, clicking the Test button will set the background color of the web page to blue. Note that the use of `bgColor` in the JavaScript function is case sensitive. Some long lines are wrapped here in print, but you can leave them all on one line in your code.

LISTING 25.5: Demonstration of a JavaScript Function

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" _
"http://www.w3.org/TR/html4/strict.dtd
<html>
<head>
<title>Javascript example</title>
<script language='javascript'>
   function changecolor(){
      document.bgColor='blue'
   }
</script>
</head>
```

```
<body>
<button type='button' onclick='changecolor()'>Test</button>
</body>
</html>
```

We have only touched on the possibilities of JavaScript in these examples. Please refer to *JavaScript Bible*, Sixth Edition, by Danny Goodman and Michael Morrison (Wiley, 2007) for more-thorough coverage. There are many online tutorials also available. A good starting point is at `www.w3schools.com/js/`.

AJAX

Asynchronous JavaScript and XML (AJAX) enables the web developer to create online applications that behave more like standard desktop apps. The asynchronous nature of the technology enables you to make an HTTP request to the server and continue to process data while waiting for the response. Data transfers are handled by using the XMLHTTPRequest object. This combines with the Document Object Model (DOM), which combines with JavaScript to dynamically update page elements without the need for a browser refresh.

A detailed exploration of AJAX is beyond the scope of this book. For our purposes, it is important to note that AJAX has been incorporated into ASP.NET 3.5 and can be leveraged into your web applications from Visual Studio 2008. We will look at this in more detail in Chapter 26, ''ASP.NET 3.5.''

An online AJAX tutorial is available from `www.w3schools.com/ajax/default.asp`.

Microformats

You can use microformats to list data on your website so that it can be accessed by various data-aggregation tools. *Microformats* are really just specifications for formatting data such as address book details or calendar information. Thousands of microformats exist. The `Microformats.org` website (`http://microformats.org`) is a good starting point for the various specifications.

The *hCard* is an example of how microformats work. The hCard specification is modeled on the *vCard* specification, which is widely used for address books. You can use the hCard creator on the Microformats website to automatically generate the code for use on your own site. You can also hand-roll your own code according to the specification. The advantage of listing your address details in the hCard format is that anyone visiting your site can automatically add your details to their own address book according to the vCard specification.

Server-Side Technologies

So far you have looked mainly at the technologies that drive the client side of a web-based application. This is only half of the equation. The website itself is normally hosted on some form of server platform and managed with a web server package. On a Microsoft platform, the web server is typically Microsoft's *Internet Information Services (IIS)*. Requests from the client are processed by the web server, and appropriate web pages are supplied. The server can also be used to process information supplied by the client as part of the request to provide a more interactive experience.

Although the range of technologies available at the client end is fairly limited, the server-side applications can be written in any language or form supported by the web server. In the case of IIS running an ASP.NET application, the application may be written in any of the .NET-supported languages.

Prior to ASP.NET, developers working in the Microsoft platform created web applications mainly using Active Server Pages (ASP). ASPs are scripted pages that combine a variety of technologies including HTML, JavaScript, server objects, Structured Query Language (SQL), and Visual Basic Script (VBScript). ASPs are created as plain-text files and saved with the `.asp` file extension. ASP is powerful and flexible, but maintaining large sites can be time-consuming, and achieving and maintaining a decent level of security can be problematic.

With the introduction of ASP.NET, Microsoft gave developers a much tidier approach to creating their web applications. You can use Visual Studio to create your applications, and superficially at least, the process of building the application is not terribly different from building standard Windows applications. Much of the plumbing for connecting to back-end databases or sophisticated objects (which in ASP you would have to lovingly hand-craft) is now taken care of. You also have the option to write much of the code into *code-behind*, where it can be safely compiled away from prying eyes and offer the performance enhancements integral to a compiled environment. Code-behind is the familiar coding interface normally used to develop desktop applications. Code written into code-behind is compiled into a library that is kept physically distinct from the scripted pages of the web application. At the scripted end of ASP.NET, files are plain text and saved with the `.aspx` file extension.

However, there are differences between the desktop and web development environments, and to really take advantage of `.NET` and fully optimize your web applications, you need to be aware of the differences and how they can be accommodated and exploited.

My favorite example of the differences between desktop and web development in Visual Studio is in the use of fragment caching. *Fragment caching* can be used to cache portions of an ASPX page that are constantly reused (such as headings). This helps create a performance boost over an equivalent page that is completely regenerated each time it is called.

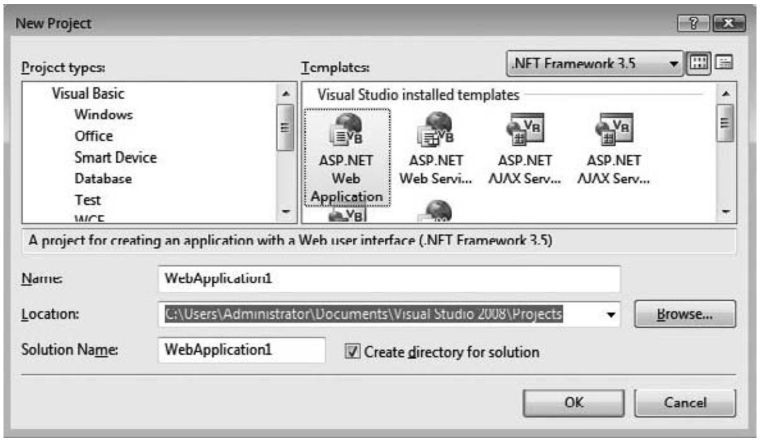
Another area where I have seen developers caught out while making the transition from desktop to web applications is in the use of view state. *View state* is used to maintain information about the current property state of the various controls on a page. It is stored on the client's web page in a hidden field and thus makes the round-trip to the server. Depending on the application, it can get very big very quickly, and even fairly plain pages can suddenly start taking long periods to download to the client if view state is not managed correctly.

For the remainder of this chapter, we will discuss how to begin creating a web application in Visual Studio 2008 and examine the available web form and HTML controls.

Creating a Web Application

Developers have two project models in Visual Studio 2008. You can use either the ASP.NET Web Application from the New Project dialog box or the New Web Site option from the File menu. It is mainly a matter of preference. In the Web Application option, project resources are defined explicitly, and the developer has tighter control over the project. It tends to suit the VB programmer coming into web programming. The New Web Site option is more flexible and tends to suit the web programmer migrating to Visual Studio.

To create a web application, open Visual Studio and select the New Project option from the File menu. From the New Project dialog box, expand the Visual Basic tree and select the Web option. From this screen, choose ASP.NET Web Application, as shown in Figure 25.5.



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

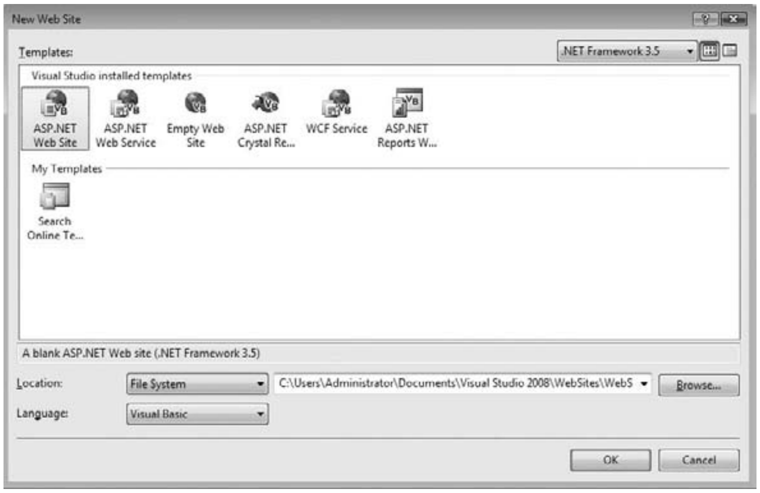
Name: WebApplication1

Location: C:\Users\Administrator\Documents\Visual Studio 2008\Projects

Solution Name: WebApplication1 ☑ Create directory for solution

FIGURE 25.5
Choosing an ASP.NET web application from the New Project dialog box

To create a new website project, open Visual Studio and select New Web Site from the File menu. This will open the New Web Site dialog box. From here, choose ASP.NET Web Site, as shown in Figure 25.6.



A blank ASP.NET Web site (.NET Framework 3.5)

Location: File System C:\Users\Administrator\Documents\Visual Studio 2008\WebSites\WebS

Language: Visual Basic

FIGURE 25.6
Choosing an ASP.NET website from the New Web Site dialog box

After the new project is open, the main difference between the interface for building a web application and that used for building a standard Windows application is that the Designer for web applications has three views: Design, Split, and Source. This enables you to alternate between a graphical view of the page and controls, an ASPX view, and a split view showing both.

The contents of the Toolbox also include HTML controls. You use the Standard controls mainly to create interactive applications, while the HTML controls are essentially client-side controls that mimic many of the traditional HTML elements such as tables and horizontal rules.

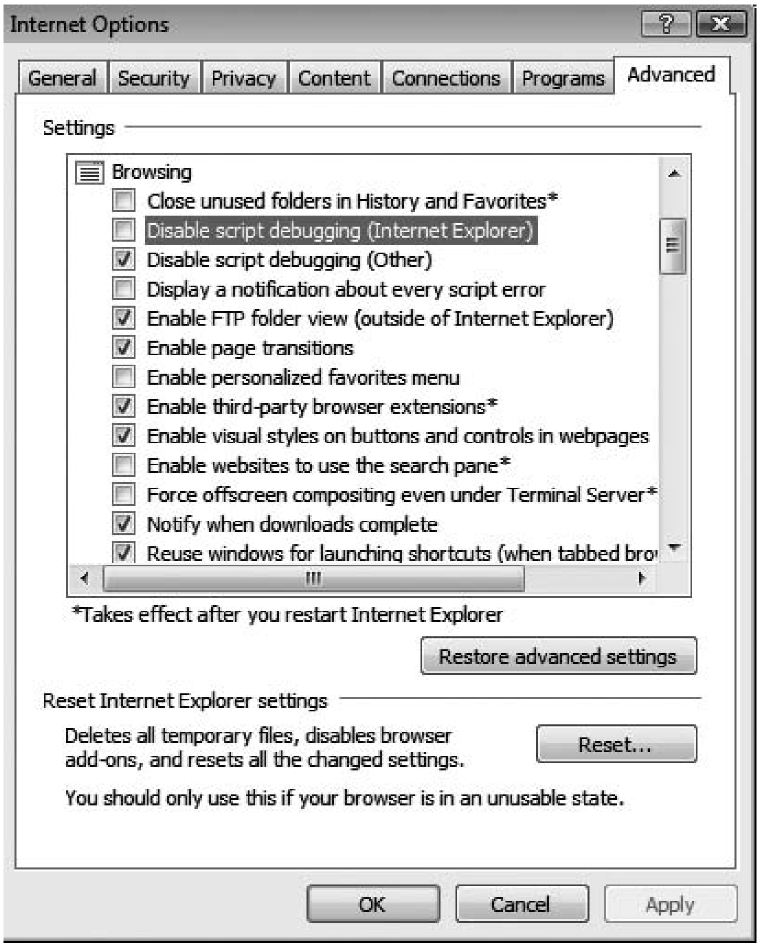
You can drag and drop controls onto the page in Design view and edit properties by using the traditional Properties window, or you can switch to Source or Split view and directly edit the code.

You can actually do most of your coding directly onto the ASPX page in Source view. This includes not only your design elements but also your business logic. However, it makes sense to separate your business logic from your design and use code-behind by hitting F7, by choosing View Code from the Solution Explorer, or by double-clicking the control in question.

When you view your application, it will open in your default browser. You may get a message warning *Debugging Not Enabled* if you have used F5 or the green arrow. You can choose to either run the project without debugging or enable debugging in the `Web.config` file. You can either modify `Web.config` manually or choose to allow Visual Studio to do it for you. However, you will need to remember to disable debugging when you go to deploy your project. To manually modify `Web.config`, double-click the `Web.config` entry in Solution Explorer. `Web.config` should open as a page of code. Under `compilation`, set `debug ="true"` as shown in the following code snippet:

```
<compilation debug="true" strict="false" explicit="true">
```

FIGURE 25.7
Enabling script debugging in Internet Explorer



Browsing
- ☐ Close unused folders in History and Favorites*
- ☐ Disable script debugging (Internet Explorer)
- ☑ Disable script debugging (Other)
- ☐ Display a notification about every script error
- ☑ Enable FTP folder view (outside of Internet Explorer)
- ☑ Enable page transitions
- ☐ Enable personalized favorites menu
- ☑ Enable third-party browser extensions*
- ☑ Enable visual styles on buttons and controls in webpages
- ☐ Enable websites to use the search pane*
- ☐ Force offscreen compositing even under Terminal Server*
- ☑ Notify when downloads complete
- ☑ Reuse windows for launching shortcuts (when tabbed bro

*Takes effect after you restart Internet Explorer

Deletes all temporary files, disables browser add-ons, and resets all the changed settings.

You should only use this if your browser is in an unusable state.

The `Web.config` file is a text file that holds many of the global settings for your website or application. The file is automatically generated when you create a new project and it can be edited manually or through various Visual Studio 2008 wizards. You need to be careful when editing the file because unlike HTML, the XML in the `Web.config` file is case sensitive. Making a mistake in `Web.config` can break your whole application.

You may also need to enable script debugging in Internet Explorer. From the Tools menu, choose Internet Options and click the Advanced tab. Under Browsing, deselect the Disable Script Debugging check box, as shown in Figure 25.7.

Controls

Several sets of controls are available to the developer when creating web applications. These are accessible from the traditional Toolbox and are separated into several categories. These include Standard, Data, Validation, Navigation, Login, WebParts, AJAX Extensions, Reporting, and HTML. Many of these controls exhibit behavior similar to that of their desktop counterparts.

Standard Controls

The Standard controls are also known as *web form controls* and have intrinsic server-side functionality that you can program against. We will explore some of these controls in more detail in Chapter 26.

Table 25.2 contains a list of the Standard controls and a brief description of each.

TABLE 25.2: Standard Controls

STANDARD CONTROL	DESCRIPTION
AdRotator	Randomly inserts content (advertisements) within a specified area according to a weighted index.
BulletedList	Displays a bulleted list.
Button	Displays a command-style button to enact code back on the server.
Calendar	Renders a calendar with calendar-style functionality on the target page.
CheckBox	Displays a single check box.
CheckBoxList	Renders a list with check box functionality against each item.
ContentPlaceHolder	Used in master pages for replaceable content.
DropDownList	Enables creation of a drop-down list of items from which the user can make a selection.
FileUpload	Creates a text box and button combination that can be used to upload a file from the client to the server.
HiddenField	Creates an `<input type = 'hidden'>` element that can be programmed with server code.

TABLE 25.2: Standard Controls *(CONTINUED)*

STANDARD CONTROL	**DESCRIPTION**
HyperLink	Creates links for navigating internally and externally to the site.
Image	Places an image on a page.
ImageButton	Enables a graphic to be specified as a button.
ImageMap	Displays an image with clickable regions.
Label	Standard control for rendering text on a page.
LinkButton	Renders a button as a link. Effectively creates a link that posts back to the server and executes whatever code has been set for it.
ListBox	Displays a list of items that may be selected individually or in multiples by the user.
Literal	Similar to the Label control in that it is used to render text to a web page, but does so without adding any additional HTML tags.
Localize	Displays text in a specific area of the page — similar to the Literal control.
MultiView	Contains View controls and allows you to programmatically display different content.
Panel	Container control that can be used to set global properties (style, etc.) for a group of controls.
PlaceHolder	Is used as a container by controls that are added at runtime and that may vary in number.
RadioButton	Displays a single radio button control.
RadioButtonList	Renders a list with radio button functionality against each item.
Substitution	Contains updateable cache content.
Table	Enables the establishment of dynamically rendered tables at runtime. Should not be used for page layout — use the HTML version of the control.
TextBox	Provides a data-entry field on a web page. Can be set as a password box with the contents obscured.
View	A panel to display a MultiView control.
Wizard	Creates a multipane control for creating wizards.
XML	Can be used to write an XML document into a web page.

Data Controls

Table 25.3 lists the controls available for data access, display, and manipulation in ASP.NET.

TABLE 25.3: Data Controls

DATA CONTROL	DESCRIPTION
DataList	Control for displaying and interacting with data as a list.
DataPager	Provides paging functionality for controls such as ListView.
DetailsView	Renders a single record as a table and allows the user to page through multiple records. Used for master-details forms. Provides ability to create, delete, and modify records.
FormView	Similar to DetailsView without predefined layout.
GridView	Displays data as a table.
ListView	Displays data as a list and supports create, delete, and update functionality.
Repeater	For creating customized lists out of any data available to a page. List format is specified by the developer.
AccessDataSource	For connecting to an Access database.
LinqDataSource	For connecting to a Linq data source.
ObjectDataSource	For connecting to a business object as a data source.
SiteMapDataSource	For use with site navigation. Retrieves navigation information from a site-map provider.
SqlDataSource	For connecting to a SQL database.
XmlDataSource	For connecting to an XML data source.

Validation Controls

Validation controls are used to establish rules for validating data entry in web forms. Table 25.4 lists the Validation controls available.

TABLE 25.4: Validation Controls

VALIDATION CONTROL	DESCRIPTION
CompareValidator	Compares the contents of two fields — for example, when constructing a password-creation confirmation check

TABLE 25.4: Validation Controls *(CONTINUED)*

VALIDATION CONTROL	DESCRIPTION
CustomValidator	Enables customized validation requirements to be set
RangeValidator	Checks that specified content or entries fall within a set range of values
RegularExpressionValidator	Checks that a field entry follows a particular specified template — for example, zip code
RequiredFieldValidator	Checks that a user has made an entry into a specified field
ValidationSummary	Reports validation status of other validation controls being used on the form

Navigation Controls

Three controls exist for assisting in the creation of navigation menus in ASP.NET. Table 25.5 lists the Navigation controls available.

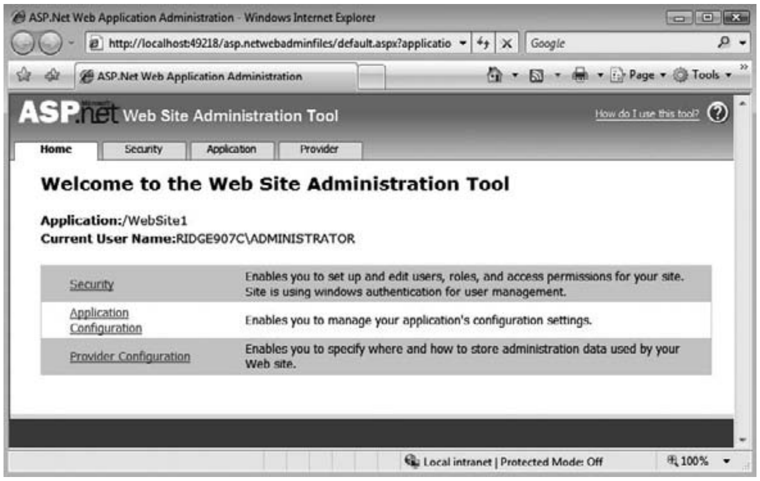
TABLE 25.5: Navigation Controls

NAVIGATION CONTROL	DESCRIPTION
Menu	Creates static and/or dynamic menus
SiteMapPath	Displays the navigation path and obtains information from the site map
TreeView	Displays hierarchical data (such as an index)

Login Controls

ASP.NET includes a *membership system* that can be used to look after authentication, authorization, and member details on your site. It is enabled by default and can be configured by using the Web Site Administration tool. You access this tool by choosing ASP.NET Configuration from the Website menu. (Note that if you are using the Web Application development environment, ASP.NET Configuration is accessed from the Project menu.)

Figure 25.8 illustrates the Web Site Administration tool. Table 25.6 lists the Login controls available.


ASP.Net Web Application Administration - Windows Internet Explorer
http://localhost:49218/asp.netwebadminfiles/default.aspx?applicatio
Google
ASP.Net Web Application Administration
Page
Tools
ASP.net Web Site Administration Tool
How do I use this tool?
Home
Security
Application
Provider
Local intranet | Protected Mode: Off
100%

Welcome to the Web Site Administration Tool

Application:/WebSite1
Current User Name:RIDGE907C\ADMINISTRATOR

Security	Enables you to set up and edit users, roles, and access permissions for your site. Site is using windows authentication for user management.
Application Configuration	Enables you to manage your application's configuration settings.
Provider Configuration	Enables you to specify where and how to store administration data used by your Web site.

FIGURE 25.8
The Web Site Administration tool

TABLE 25.6: Login Controls

LOGIN CONTROL	DESCRIPTION
ChangePassword	Allows users to change their passwords
CreateUserWizard	Displays a wizard for gathering information from a new user
Login	Displays an interface for user authentication
LoginName	Displays user's login name
LoginStatus	Displays logout link for authenticated user and login link for nonauthenticated user
LoginView	Displays different information for anonymous and authenticated users
PasswordRecovery	Recovers passwords based on email details entered when account was created

WebParts Controls

WebParts enable users to personalize their view of your website by modifying the content, appearance, and behavior of the web pages from their browsers. Table 25.7 lists the WebParts controls available.

AJAX Extensions Controls

To fully utilize AJAX in your applications, you will also need to download the ASP.NET AJAX Control Toolkit from the ASP.NET Ajax website at `http://asp.net/ajax/`. Table 25.8 lists the available AJAX Extensions controls that ship in Visual Studio 2008.

TABLE 25.7: WebParts Controls

WEBPARTS CONTROL	DESCRIPTION
AppearanceEditorPart	Enables end user to edit certain appearance properties of an associated WebPart control
BehaviorEditorPart	Enables end user to edit certain behavior properties of an associated WebPart control
WebPartManager	Used once on a page to manage all WebParts controls on that page
CatalogZone	Contains CatalogPart controls — catalog of controls that users can select for use
ConnectionsZone	Contains WebPartConnection controls — two web part controls that are linked
DeclarativeCatalogPart	Used with CatalogZone control to enable you to add a catalog of web parts to your web page
EditorZone	Area in which users can personalize controls
ImportCatalogPart	Imports a description file for a WebPart control — enabling the user to add the web part with predefined settings
LayoutEditorPart	Editor control for users to edit layout properties of a web part
PageCatalogPart	Provides a catalog of all web parts that a user has closed on a web page — enabling the user to add the controls back again
PropertyGridEditorPart	Editor control for user to edit properties of a web part
ProxyWebPartManager	For use in a content page to declare static connections when a WebPart Manager has been used in the associated master page
WebPartZone	Provides overall layout for WebPart controls

TABLE 25.8: AJAX Extensions Controls

AJAX EXTENSIONS CONTROL	DESCRIPTION
ScriptManager	Manages script resources for clients — required for Timer, UpdatePane, and UpdateProgress controls. Use only once on a page.
ScriptManagerProxy	For use in circumstances where a page already has a ScriptManager control.
Timer	Performs postbacks at specified interval.
UpdatePanel	Enables you to asynchronously refresh portions of a page.
UpdateProgress	Provides progress details on partial page updates.

Reporting Controls

Four Reporting controls exist to create data-driven reports in ASP.NET. Table 25.9 lists the Reporting controls available.

TABLE 25.9: Reporting Controls

REPORTING CONTROL	DESCRIPTION
CrystalReportViewer	For hosting a report in a web application
CrystalReportPartsViewer	Displays a Crystal report as a series of linked parts
CrystalReportSource	Data source control for Crystal Reports
MicrosoftReportViewer	Tool for creating and displaying a report

HTML Controls

Table 25.10 lists the HTML controls available. These are not typically exposed to the server for you to program. However, you can convert any HTML control to an HTML server control by adding the attribute `runat ="server"` to the control in ASPX view. This will allow you to manipulate the HTML control's functionality from the server. If you wish to reference the control within your code, you will need to add an `id` attribute as well.

TABLE 25.10: HTML Controls

HTML CONTROL	DEFAULT HTML GENERATED
Div	`<div style="width: 100px; height: 100px"> </div>`
Horizontal Rule	`<hr />`
Image	`<img alt="" src="" />`
Input (Button)	`<input id="Button2" type="button" value="button" />`
Input (Reset)	`<input id="Reset1" type="reset" value="reset" />`
Input (Submit)	`<input id="Submit1" type="submit" value="submit" />`
Input (Text)	`<input id="Text1" type="text" />`
Input (File)	`<input id="File1" type="file" />`
Input (Password)	`<input id="Password1" type="password" />`
Input (Checkbox)	`<input id="Checkbox1" type="checkbox" />`

TABLE 25.10: HTML Controls *(CONTINUED)*

HTML Control	Default HTML Generated
Input (Radio)	`<input id="Radio1" checked="checked" name="R1" type="radio" value="V1" />`
Input (Hidden)	`<input id="Hidden1" type="hidden" />`
Select	`<select id="Select1" name="D1">` `<option></option>` `</select>`
Table	`<table style="width:100%;">` `<tr>` `<td>&nbsp;</td>` `<td>&nbsp;</td>` `<td>&nbsp;</td>` `</tr>` `<tr>` `<td>&nbsp;</td>` `<td>&nbsp;</td>` `<td>&nbsp;</td>` `</tr>` `<tr>` `<td>&nbsp;</td>` `<td>&nbsp;</td>` `<td>&nbsp;</td>` `</tr>` `</table>`
Textarea	`<textarea id="TextArea1" cols="20" name="S1" rows="2"></textarea>`

Maintaining State

An issue for developers when working with web-based applications is that a web server does not intrinsically maintain an ongoing connection with the client, and each request (even by the same client viewing the same website) is treated as an entirely separate request. The business of persisting information about the client and what the client is doing from one request to the next is called *maintaining state*. A set of related requests originating from a client viewing a particular website or using a web application is called the client's *session*.

As a web developer, you need to consider how you will maintain state for your clients and web applications. You need to come up with a way for the server to remember your client and the client session between requests, and for your client to identify itself to the server with each request. The issue is complicated by the fact that there are multiple methods of maintaining state, and each comes with its own set of advantages and disadvantages. At the client end, these methods include the following:

Using cookies *Cookies* are small files deposited in the client browser's cache. Many users turn these off or restrict their usage because of security and privacy concerns.

Using hidden fields in the web page This method is reliable, but you will need to code specifically at the server to read the content. Hidden fields can also end up carrying a lot of data, can pose a security risk because the information is available in clear text in the page's source code, and can get messy if your client uses unexpected navigation techniques such as the browser's back button rather than your built-in site navigation.

Incorporating state information into the URL This method is reliable but restrictive. It also can be a security risk (with data stored in a browser's history, for example) and may cause issues with unexpected navigation techniques.

At the server end, typical methods of maintaining state (after the client has been identified) include the following:

Using session variables This is the simplest method. It uses the Session object. Session variables behave a bit like global variables, and all the usual warnings apply.

Storing information in a database This is a powerful and flexible method, but it adds overhead, particularly for a simple site.

Sending the information back to the client in hidden fields This works well but issues exist, as outlined previously.

For simple sites, ASP.NET takes care of most of these issues for you by using a combination of techniques. You can use session variables to manage small amounts of data between pages, and a database for anything more involved. It is, however, a good idea for you to keep an eye on the ViewState settings of your controls so as to minimize the amount of data making the round-trip from server to client and back again. You can enable/disable ViewState for any individual control by using the `EnableViewState` property in that control's Property window.

To create a session variable, simply type `Session("MyVariableName") ="variable content"`. Insert a relevant name and content. Be careful that you do not reuse a variable name for another purpose because the contents of the original variable will be overwritten.

To access the session variable, you refer to the full `Session("MyVariableName")`.

If you are setting up a site that will employ identification of its users, require some form of authentication, and/or offer customization of settings, it is a good idea to use Microsoft's membership system, which is available through the Web Site Administration tool and the Login controls. Refer to the ''Login Controls'' section earlier in this chapter. You will also see how to use this system in the next chapter, on ASP.NET 3.5.

Master Pages

ASP.NET 2 introduced the idea of *master pages* as a method of maintaining a consistent look and feel for a website or application. This approach has been continued with ASP.NET 3.5.

The idea is to create a page (or a number of pages), known as a master page, from which your web pages derive their common elements. Web pages linked to master pages are known as *content pages*. It is a little like using CSS style sheets to control your web page styles and structure in a scripted setting.

To add a master page to a site, simply choose the Master Page template from the Add New Item option in the Website (for ASP.NET Web Site) or Project (for ASP.NET Web Application) menu. The master page has the file extension `.master`. You can rename the master page appropriately, but do not change the file extension!

In the master page, you can set up standard items that remain consistent across your site such as headers, footers, and navigation bars. You can also place ContentPlaceHolder controls in those areas where you are planning on customizing your content pages. The ContentPlaceHolder controls provide editable locations where you can add additional controls and information. You will need to right-click the master controls and choose the Create Custom Content option. In addition, you can create a style sheet to control the appearance of your master page (and hence its attached content pages).

If you make changes to your master pages, these changes will be reflected in your attached content pages. (You will need to save your changes to the master page before the updates are reflected through the content pages.)

A master page is not automatically added to your pages. You must explicitly attach it. For example, to attach it to a new page, choose Web Form from the Add New Item dialog box and select the Select Master Page check box. Click the Add button, and this will open another dialog from which you can choose the appropriate master page. Click OK and you are ready to go. You can add content into the ContentPlaceHolder controls inherited from the master page.

If you already have your master page open in the IDE, you can simply use the Add Content Page option from the Website menu to directly create a content page attached to the particular master page you are browsing.

Trying to connect an existing page, such as the `default.aspx` page initially created in the application, to a master page can be problematic, so it is often a good idea to delete it. To set a new default page for your website, right-click the desired page in Solution Explorer and choose the Set As Start Page option.

In the next chapter, you will see how a master page is applied.

ASP.NET Objects

Objects are available in ASP.NET that can be used to provide you with information about the state of your application, each user session, HTTP requests, and more. You need to be familiar with some of these because they can be useful in your code. Many of them also expose useful utility methods for managing your web application. For example, you have already seen how you can use the Session object to create a session variable. In this section, you will briefly look at the main objects and some of their methods and properties.

Application Object

The Application object stores information related to the full web application, including variables and objects that exist for the lifetime of the application.

Context Object

The Context object provides access to the entire current context and can be used to share information between pages. This includes the current Request and Response objects.

Request Object

The Request object stores information related to the HTTP request, such as cookies, forms, and server variables. You can use this object to see everything passed back to the server from the client.

The Request object includes the properties shown in Table 25.11. Table 25.12 shows the methods for the Request object.

Table 25.11: Properties for the Request Object

`ApplicationPath`	Indicates the virtual path of the application
`Browser`	Gets or sets information about the client's browser and its capabilities
`ClientCertificate`	Gets the client's security certificate
`Cookies`	Gets the cookies sent by the client
`FilePath`	Gets the virtual path of the request
`Form`	Gets a collection of form variables
`IsAuthenticated`	Indicates whether the request has been authenticated
`IsLocal`	Indicates whether the request originates from a local computer
`Item`	Gets specified object from Cookies, Form QueryString, or ServerVariables
`LogonUserIdentity`	Gets Windows identity for user
`QueryString`	Gets the collection of HTTP query string variables
`ServerVariables`	Gets the collection of web server variables
`URL`	Gets the URL of the request
`UserHostAddress`	Gets the IP address of the client
`UserHostName`	Gets the DNS name of the client
`MapPath`	Maps the virtual path in the requested URL to the physical path on the server
`SaveAs`	Saves the request to disk

Response Object

The Response object contains the content sent to the client from your server. You can use the Response object to send data such as cookies to your client.

The Response object includes the properties shown in Table 25.12 and the methods shown in Table 25.13.

Server Object

The Server object exposes methods that can be used to handle various server tasks. You can use these methods to create objects, to map paths, and to get error conditions.

The properties for the Server object are shown in Table 25.14, and the methods are in Table 25.15.

TABLE 25.12: Properties for the Response Object

`Buffer`	Gets or sets the value as to whether to buffer output
`Cookies`	Returns the response cookies collection
`ContentType`	Gets or sets HTTP MIME type in output
`Expires`	Gets or sets the cache expiration of a page (in minutes)
`IsClientConnected`	Indicates whether a client is still connected

TABLE 25.13: Methods for the Response Object

`AppendCookie`	Adds a cookie to the collection
`AppendHeader`	Adds an HTTP header
`ApplyAppPathModifier`	Adds the session ID to the virtual path if a cookieless session is being used
`Clear`	Clears all content output from the buffer
`End`	Sends all buffered output to the client and stops execution of the page
`Flush`	Sends all buffered output to the client
`Redirect`	Redirects the client to a new URL
`SetCookie`	Updates an existing cookie
`Write`	Writes additional text to the response output

TABLE 25.14: Properties for the Server Object

`MachineName`	Returns the server's name
`ScriptTimeout`	Gets and sets time-out value for requests in seconds

TABLE 25.15: Methods for the Server Object

`Execute`	Commonly used to execute a URL to open another page from within your code
`HTMLDecode`	Decodes a string that has been encoded to remove illegal HTML characters
`HTMLEncode`	Encodes a string to display in a browser
`MapPath`	Gets physical file path of the specified virtual path on the server
`UrlEncode`	Encodes a string for transmission through the URL
`UrlDecode`	Decodes a string encoded for transmission through a URL

Session Object

The Session object stores information related to the user's session, including variables, session ID, and objects. Properties for the Session object are shown in Table 25.16; methods are in Table 25.17.

TABLE 25.16: Properties for the Session Object

`Count`	Returns the number of items in the current session state collection
`Item`	Gets or sets individual session values
`LCID`	Gets or sets the locale identifier
`SessionID`	Gets the identifier for the session
`Timeout`	Gets or sets the time between requests in minutes before the session terminates

TABLE 25.17: Methods for the Session Object

`Abandon`	Terminates the current session
`Add`	Adds a new item to the session state collection
`Clear`	Clears all values from the session state collection
`Remove`	Removes an item from the session state collection
`RemoveAll`	Removes all items from the session state collection

Trace Object

The Trace object can be used to display system and custom diagnostics in the page output. Properties for the Trace object are shown in Table 25.18, and methods are shown in Table 25.19.

TABLE 25.18: Properties for the Trace Object

`IsEnabled`	Gets or sets whether tracing is enabled
`TraceMode`	Gets or sets the order in which trace messages are written to the browser

TABLE 25.19: Methods for the Trace Object

`Write`	Writes trace information to the trace log

Postback

An important aspect of the way that ASP.NET operates is that controls that run on the server are able to post back to the same page. This process is called *postback*. This is different from the old ASP model, in which often there would be two or three pages set up to host the controls, process the code, and provide a response.

Any ASP.NET page that has at least one visible control will include the JavaScript function `_doPostBack`. This function records the control that initiated the postback, plus additional information about the initiating event, and includes it in the data submitted back to the server.

The `Postback` property is a read-only value that is set to *False* when a page is first loaded and is then set to *True* when the page is subsequently submitted and processed. At the server end, you can use the function `Page.IsPostBack()` to determine the state of a page's postback and write code accordingly — this is particularly useful when deriving your page content from a database.

The Bottom Line

Create a basic XHTML/HTML page. Building a basic HTML page is a straightforward process using a simple text editor such as Notepad. Knowledge of XHTML/HTML is still a major asset when developing web applications with Visual Studio 2008.

Master It Develop a web page using HTML that features a heading, some text, an image, and a link to another page. Convert the page to XHTML and verify it by using the W3C verification tool at `http://validator.w3.org`. You might find that you will need to run the validation a couple of times to get everything right. If you attach and use the style sheet in the following Master It challenge, you will find that the validation will be less problematic.

Format a page with CSS. Cascading Style Sheets (CSS) are a powerful tool for controlling the styles and format of a website. You can manually create style sheets by using a text editor. An understanding of their operation and syntax is a useful skill when manipulating CSS in Visual Studio .2008.

Master It Create a CSS style sheet that defines the layout of the web page that you developed in the previous task, including a header section, a left-hand navigation section, and a main content section. Include a rollover for the link and apply formatting to the tags that you have used for your heading and text tags. Attach the style sheet to your web page.

Set up a master page for your website. Using master pages is a reliable method of controlling the overall layout, and look and feel of your websites and applications. Master pages enable you to achieve a high level of consistency in your design and are particularly useful if the site has multiple developers working on it.

Master It Create a website with a master page and attached content page. Use appropriate controls to give the master page a page heading, My Page, which will appear on all attached content pages. Use a combination of Button and Label controls on the content page to create a simple Hello World application.

Use some of the ASP.NET intrinsic objects. ASP.NET objects such as the Response, Request, Session, and Server objects offer a range of important utilities in developing, running, and maintaining your websites and applications. In addition, they also give you access to vital information about the client, the server, and any current sessions at runtime.

Master It Create a simple website with a master page and two attached content pages. Use the `Server.Execute` method attached to a LinkButton control to navigate between the two content pages.

Chapter 26

ASP.NET 3.5

ASP.NET 3.5 is the latest incarnation of Microsoft's principal technology for developing and delivering websites and web-based applications.

As you saw in Chapter 25, ''Building Web Applications,'' building a modern web-based application with ASP.NET involves working with a range of different technologies around the ASP.NET backbone. The modern web developer needs to be conversant with technologies such as Hypertext Markup Language (HTML), Extensible Markup Language (XML), Cascading Style Sheets (CSS), databases, JavaScript and, in our case, Visual Basic 2008.

Visual Studio 2008 brings all these technologies together underneath one umbrella. Visual Studio 2008 provides web developers with seamless access to the various technologies as well as the familiar and supportive environment of the Visual Studio interface.

Developing with ASP.NET offers you the choice of coding within the scripted ASPX environment or working with a language such as Visual Basic 2008 in code-behind. You can also split your code between the two. This offers the advantage of combining the ease and flexibility of the scripted environment with the performance and security of a compiled environment. It also enables you to neatly separate your business logic from your user interface (UI). You have full access to Microsoft's IntelliSense and AutoComplete support in the scripted environment as well as in the more traditional code-behind environment.

In this chapter, we will build a demonstration web application to showcase a number of the technologies, controls, and methodologies available within Visual Studio 2008. In particular, you will see how Visual Studio 2008 greatly simplifies complex tasks such as setting up authentication, managing navigation, and connecting to data sources with a range of graphical tools and wizards.

In this chapter, you will learn how to do the following:

- Create cascading style sheets
- Use web form controls
- Create a web user control

Planning the Demonstration Site

In this chapter, we will build a sample website with various interactive features for the mythical company Wilson's Computer Parts Online. The purpose of the website is to demonstrate the various techniques, controls, and technologies available to you in ASP.NET 3.5. The full source code for this example can be downloaded from this book's website (`www.sybex.com`).

The site will include a main front page (`default.aspx`) with its format and content controlled from a single master page and cascading style sheet. A separate master page and style sheet will be created to define the layout and content of the daughter pages of the site. The structure and function of master pages and their associated content pages were discussed in Chapter 25.

Other techniques that we will showcase with this web application include the following:

- Creating and managing an authentication process using the range of Login controls
- Handling site navigation by creating a site map and using the Menu and SiteMapPath controls
- Creating user-defined controls that can be centrally managed and reused throughout the application
- Creating and connecting to data sources, including an XML page and a database
- Working with the data-bound GridView control to display data from the XML page
- Working with the data-bound DropDownList and ListView controls to create a master/detail form capable of viewing and editing data on the database
- Using the Microsoft Report Viewer to display a report of data on the database

Getting Started

Begin by opening Visual Studio 2008. Choose File ➤ New Web Site to open the New Web Site dialog box. Complete the following steps:

1. From the New Web Site dialog box, choose ASP.NET Web Site.
2. In the Location field, keep the default path but rename the site **WilsonsComputerParts-Online**. Click the OK button.
3. The next step is to delete the `default.aspx` page. You need to do this because you will be using master pages to define the layout and content of all your pages. As described in Chapter 25, it is difficult to attach a master page to an existing page. Right-click `default.aspx` in Solution Explorer and choose Delete from the context menu.
4. Next you need to create the master page that will define the layout and content of the main page for our site. Create a new master page by choosing Add New Item from the Website menu. This opens the Add New Item dialog box. Choose Master Page from the Templates window. You may need to click the path and name of your website in Solution Explorer (at the top of the tree) to get the correct view of the Add New Item dialog box. Keep the default name of `MasterPage.master`, and then click Add.

Working with the Add New Item Dialog Box

Your view of items available in the Add New Item dialog box can vary according to which item you currently have selected in the Solution Explorer tree. In particular, if you have the App_Code or App_Data folder selected, you will see a very limited range of items in the Add New Item dialog box.

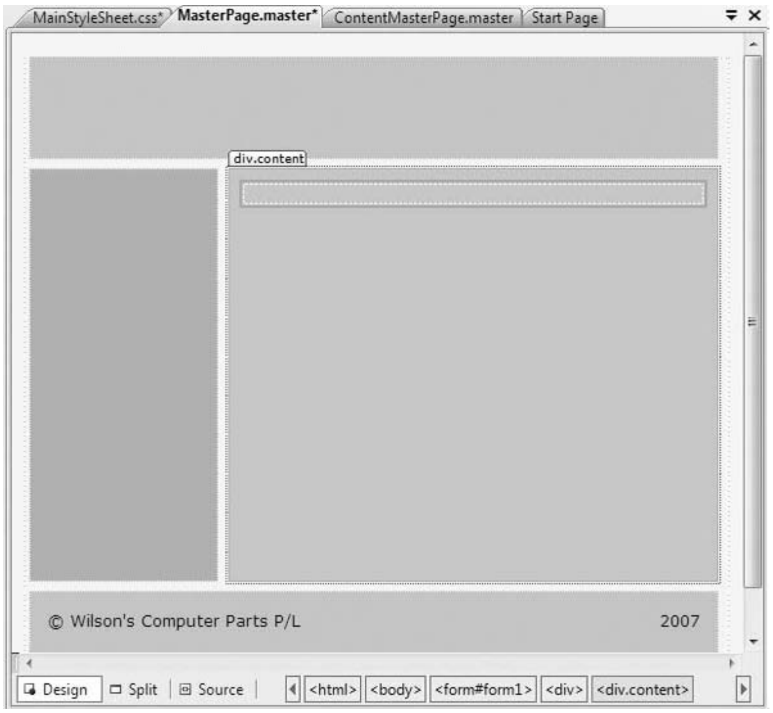
We will leave our master page for the time being to create the style sheet that will determine the layout and style properties of `MasterPage.master`.

Building the Style Sheet for *MasterPage.master*

As you saw in Chapter 25, a cascading style sheet may be used to both define the styles used in a web page and set the layout for the page. In this example, we will use the style sheet primarily to determine the layout for our master page.

MasterPage.master defines the default entry page for our website. We want something that will provide a large heading or title area at the top of the page, a menu or navigation area down the left-hand side of the page, a main content area, and a footer area along the bottom of the page. Figure 26.1 illustrates how this page should appear.

FIGURE 26.1
Page layout for the master page



The various layout areas of MasterPage.master will be defined by using a style sheet. Within the style sheet, we will define the following four classes, one for each layout section:

.title	The main heading area of the page
.menu	The navigation pane on the left side of the page
.content	The main content area of the page
.footer	The footer at the bottom of the page

In CSS, creating footers can be difficult because of the uncertainty of the actual height of the other sections of the page. In particular, this causes issues if ''absolute'' positioning has been used to lay out the page. To overcome this problem, we will use the `float` layout property to control the layout of the menu in combination with the default `static` layout property used for the other sections. Because `static` is the default layout property, it does not have to be specified.

Creating the Style Sheet

To create the style sheet, start by opening the Website menu and choosing Add New Item. In the Add New Item dialog box, create a new style sheet. Name the style sheet **`MainStyleSheet.css`** and click the Add button. This opens `MainStyleSheet.css` as an editable script document.

When creating a style sheet, you have the option of using the Visual Studio 2008 graphical development tools or writing your code directly. Writing code directly tends to be a lot quicker, and you can use the IntelliSense support to help with correct syntax. On the other hand, the graphical tools are great when you know what you want to achieve but are unsure of how to actually do it.

As with most things in Visual Studio 2008, there are various ways to access the graphical style tools — from the Styles menu, the Styles toolbar, or the Styles window tab at the bottom of the Toolbox. Each offers you options to create a style rule and then to build the style. (You may need to click somewhere on your style sheet page to have these options active.)

We will begin by creating the style for the ''body'' element of our style sheet by using the graphical designer tools. The body element should be a default entry in `MainStyleSheet.css`. Follow these steps:

1. Click inside the braces of the body element and then choose Styles ➢ Build Style.
2. In the Modify Style dialog box, select the Background category and type **#FFF8DC** into the background-color area. This sets the background color to Cornsilk. Click the OK button. The following entry should now appear in the code listing for `MainStyleSheet.css`:

   ```
   body
   {
      background-color: #FFF8DC;
   }
   ```

3. Next, you will use the same tools to create the title class. This class will define the heading area of our web page. Click somewhere below the body element on your style sheet and then choose Styles ➢ Add Style Rule. This opens the Add Style Rule dialog box. Click the Class Name radio button and type **title** into the text box. Click the OK button to exit the dialog box.
4. Open the Styles menu and select Build Style to open the Modify Style dialog box. Table 26.1 lists the properties that you need to set for each of the categories. Click the OK button to exit the dialog box and set the styles.

   Alternatively, you can directly type the CSS formatting code. All of the properties in this class should be self-explanatory, with the possible exception of `overflow:hidden`. This style hides any content that may spill outside the title box. The background color of this area has been set to light blue. The completed code should resemble the following snippet (the order of items may vary):

   ```
   .title
   {
      margin: 5px 10px 10px 5px;
      vertical-align: middle;
   ```

```
    text-align: center;
    background-color: #ADD8E6;
    height: 80px;
    overflow: hidden;
}
```

Table 26.1: Property Settings for Each Category for the Title Class

Category	Property	Value
Block	`vertical-align`	middle
	`text-align`	center
Background	`background-color`	#ADD8E6
Box	`margin-top`	5
	`margin-right`	10
	`margin-bottom`	10
	`margin-left`	5
Position	`height`	80px
Layout	`overflow`	hidden

5. Next, you will create the navigation menu sidebar area. Either use the graphical tools as described previously or type the following snippet directly below your completed entry for *title* on `MainStyleSheet.css`. In this case, we are using the `float` property to set the positioning of the menu area. This is to overcome some of the issues described previously for when we create the footer area. The background color of the menu area has been set to silver.

```
.menu
{
    Background-color: #C0C0C0;
    overflow:hidden;
    float: left;
    padding: 10px;
    margin-left:5px;
    margin-right:10px;
    width: 130px;
    height:400px;
}
```

6. The next step is to add the main content area. Again, use the graphical tools or type the following code snippet directly underneath your completed entry for *menu*. The background color for the menu area is light blue.

```
.content
{
   background-color: #ADD8E6;
   overflow:hidden;
   height:400px;
   margin-right: 10px;
   margin-top:10px;
   padding: 10px;
}
```

7. The final step is to create the footer area. Add the following code snippet directly underneath your completed entry for *content*. Setting the `clear` property to *both* for the footer ensures that any other elements are kept clear of either side of the footer. The background color for this area is light blue.

```
.footer
{
   background-color: #ADD8E6;
   clear:both;
   padding: 10px;
   height: 20px;
   margin-top:10px;
   margin-left: 5px;
   margin-right: 10px;
}
```

This completes setting up `MainStyleSheet.css`. Save your work at this stage. The complete code listing for `MainStyleSheet.css` is available from the website for this book.

For more information on working with CSS, the following URL is a good starting point: `www.w3schools.com/css/default.asp`.

Attaching the Style Sheet to the Master Page

The next step is to add a reference to the completed `MainStyleSheet.css` into the `MasterPage.master` that we created earlier. As with most things in Visual Studio, there are several ways of achieving this. By far, the simplest is the drag-and-drop method. Complete the following:

1. From Solution Explorer, double-click the entry for `MasterPage.master` to open the page in Design view. Click the Source tab at the bottom of the page to switch to Source view.
2. Locate the entry for `MainStyleSheet.css`. Select the entry and drag it across to the Source view of `MasterPage.master`. Drop the entry into the <head> area of the markup for `MasterPage.master` — just above the closing </head> tag is suitable. An entry similar to the following snippet should be created.

   ```
   <link href="MainStyleSheet.css" rel="stylesheet" type="text/css" />
   ```

   If you prefer, you can just type this manually into the code. You now need to reference the layout classes on the master page.

3. Switch back to Design view. There is a default Div control already on the form.
4. Drag and drop a Div control from the HTML toolbox into the existing Div control on the form. You do not really need this containing Div control except when you are using the graphical Toolbox controls. Keeping and using the default Div control ensures that controls that you add to the page fall within the page's `<form>` tags. (However, it is a good idea to switch to Source view and check that this has happened — you may still need to adjust the code manually, by shifting the `<div>` tags inside the opening and closing form tags.) If you are working directly within Source view, you can ensure that this happens without keeping the default `<div>` container.
5. In the Properties window for the Div control that you have just added, delete the default entry for the `Style` property and set the `Class` property to **title**.
6. Add three more Div controls directly underneath the first one that you added (and into the default containing Div control). Delete the default entry for the `Style` property for each of these Div controls. Set the `Class` property of the second Div control to **menu**, for the third Div control to **content**, and for the fourth Div control to **footer**.
7. Switch to Source view and delete the `<ContentPlaceHolder>` tags from the Form section of the page. Switch back to Design view and add a ContentPlaceHolder control from the Standard toolbox to the Content area of the master page.
8. Click Save and switch back to Design view. The page layout should now look similar to that shown originally in Figure 26.1.

If you switch back to Source view, the code for the page should appear as in Listing 26.1. There may be some variations depending on how much you have relied on the Visual Designer or worked directly in Source view. Delete any line breaks.

LISTING 26.1: ASPX Code for *MasterPage.master*

```
<%@ Master Language="VB" CodeFile="MasterPage.master.vb" _
Inherits="MasterPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" _
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title>Untitled Page</title>
   <asp:ContentPlaceHolder id="head" runat="server">
   </asp:ContentPlaceHolder>
   <link rel="stylesheet" type="text/css" href="MainStyleSheet.css"/>
</head>
<body>
   <form id="form1" runat="server">
      <div>
         <div class="title">
         </div>
```

```
          <div class="menu">
          </div>
          <div class="content">
             <asp:ContentPlaceHolder id="ContentPlaceHolder1" _
runat="server">
             </asp:ContentPlaceHolder>
          </div>
          <div class="footer">
          </div>
       </div>
    </form>
</body>
</html>
```

Refer back to Chapter 25 for a detailed explanation of the various sections of this markup. Two things to note here are that there are two ContentPlaceHolder controls. The one in the `<head>` area enables you to add metadata to a content page, and the one in the content section enables you to add content to a content page.

This completes the layout for `MainMasterPage.master`. Next we will set up a new master page (and associated style sheet) to manage the daughter pages of the site.

Creating the Content Master Page

The purpose of the content master page (`ContentMasterPage.master`) is to act as a template for the daughter pages of the website. This page will also have its style and layout set by a style sheet. In this example, to keep things simple, we will set up a separate style sheet. However, in a real-world implementation, you might find it more convenient to keep all your styles together in the one sheet.

The `ContentMasterPage.master` file will have a heading (title) area with a horizontal navigation bar (navbar) directly beneath the heading. This will be followed by a large content area (content) filling the full width of the page, and finished with the standard footer (footer) at the bottom of the page. Figure 26.2 illustrates how the completed page layout will appear.

We'll begin by adding a new master page. Complete the following steps:

1. Choose Website ➤ Add New Item.
2. From the Add New Item dialog box, select Master Page and name it **`ContentMasterPage.master`**. Click the Add button to exit.
3. Next you will create the style sheet to set the layout for `ContentMasterPage.master`. Open the Website menu and again choose Add New Item. Select Style Sheet and name it **`ContentStyleSheet.css`**. Click the Add button to exit the dialog box.
4. Click the Save All button on the Standard toolbar to save your work.
5. At this stage, we can attach `ContentStyleSheet.css` to `ContentMasterPage.master` before building the classes in the style sheet. Open `ContentMasterPage.master` in Design mode by double-clicking its entry in Solution Explorer.

6. Switch to Source view and locate the entry for `ContentStyleSheet.css` in the Solution Explorer. Drag the `ContentStyleSheet.css` entry into Source view and drop the entry just above the closing `</head>` tag. This should create an entry in the ASPX code for `ContentMasterPage.master` similar to the following snippet:

```
<link rel="stylesheet" type="text/css" href="ContentStyleSheet.css"/>
```

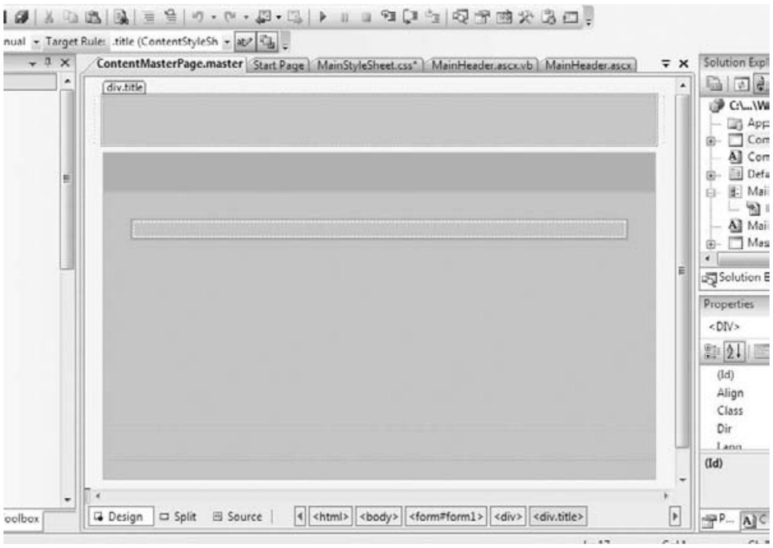


FIGURE 26.2
Page layout for the content master page

Next, we will create the layout classes in `ContentStyleSheet.css`.

Creating *ContentStyleSheet.css*

`ContentStyleSheet.css` sets the layout for `ContentMasterPage.master`. This in turn acts as a template for the daughter pages in the website. Within `ContentStyleSheet.css`, we will define four classes, one for each of the following layout sections:

.title	The main heading area of the page
.navbar	The navigation bar directly below the title area
.content	The main content area of the page
.footer	The footer at the bottom of the page

As with `MainStyleSheet.css`, you can either use the graphical development tools to create the style sheet or simply type the code directly.

The classes for this style sheet are similar to those we created in `MainStyleSheet.css`, with the exception of the navbar area. By having the navbar as a narrow area that runs the full width of the page, we have simplified the layout. The layout areas are stacked vertically down the page, so we can rely on the default static layout property (which does not need to be specified) for all the classes and do not need to use the flow layout property as we did previously. Specifying the left and right margins of each of the layout areas ensures that they extend right across the page.

Begin by opening `MainStyleSheet.css` by double-clicking its entry in Solution Explorer. Complete the following steps:

1. Set the background color of the page by completing the following entry in the default body code skeleton:

    ```
    body
    {
       background-color: #FFF8DC;
    }
    ```

2. Create the title class by adding the following snippet immediately underneath the completed body element:

    ```
    .title
    {
       height:80px;
       background-color: #ADD8E6;
       margin:5px 10px 10px 5px;
       text-align: center;
    }
    ```

3. Create the navbar class by adding the following snippet directly beneath the completed title class:

    ```
    .navbar
    {
       background-color: #C0C0C0;
       overflow:hidden;
       height:10px;
       padding: 10px;
       text-align: left;
       margin-left: 5px;
       margin-right: 10px
    }
    ```

4. Create the content class by adding the following snippet directly beneath the completed navbar class:

    ```
    .content
    {
       background-color: #ADD8E6;
       overflow:hidden;
       padding: 20px;
       height:400px;
       margin-top:10px;
       margin-left: 5px;
       margin-right: 10px
    }
    ```

5. Finally, complete the style sheet by adding the following snippet for the footer class immediately beneath the completed content class:

```
.footer
{
   background-color: #ADD8E6;
   overflow:hidden;
   padding: 10px;
   clear:both;
   height: 20px;
   margin-top: 10px;
   margin-left: 5px;
   margin-right: 10px
}
```

Save your work. The full code listing for `ContentStyleSheet.css` is available from the website for this book.

Next, we will finish setting the layout for `ContentMasterPage.master`.

Completing *ContentMasterPage.master*

Now that we have created the style sheet, we need to apply the layout classes to `ContentMasterPage.master`. Open `ContentMasterPage.master` in Design view by double-clicking its entry in Solution Explorer. Then complete the following steps:

1. From the HTML toolbox, drop a Div control into the default Div control already present in `ContentMasterPage.master`. Again, you can type these directly into Source view and do away with the need for the containing Div tag. However, if you continue to use the Visual Designer, using the default containing Div tag will help ensure that any controls you add will fall within the page's <`form`> tags. (Although it is always a good idea to switch to Source view to check — the Visual Designer does not always play nice.)
2. In the Properties box for the Div control that you have just added, delete the default entry for the `Style` property and set the `Class` property to title.
3. Add three more Div controls from the HTML toolbox so that they fall within the default containing Div control. Delete the default entry for the `Style` property for each Div control. Set the `Class` property of the second Div control to **navbar**, the third Div control to **content**, and the fourth Div control to **footer**.
4. Switch to Source view and delete the <`ContentPlaceHolder`> tags from the Form section of the page. Switch back to Design view and add a ContentPlaceHolder control from the Standard toolbox to the Content area of the master page.
5. Click Save and switch back to Design view. The page layout should now look similar to that shown originally in Figure 26.2.

If you switch to Source view, Listing 26.2 illustrates how the ASPX code should look for the page at this stage. Again, there may be some minor variations depending on how much you have worked directly in Source view and how much you have relied on the tools from the Visual Designers. Some long lines are wrapped here in print, but you can leave them all on one line in your code.

Listing 26.2: ASPX Code for *ContentPageMaster.master*

```
<%@ Master Language="VB" CodeFile="ContentMasterPage.master.vb" _
 Inherits="ContentMasterPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" _
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title>Untitled Page</title>
   <asp:ContentPlaceHolder id="head" runat="server">
   </asp:ContentPlaceHolder>
   <link rel="stylesheet" type="text/css" href="ContentStyleSheet.css"/>
</head>
<body>
   <form id="form1" runat="server">
       <div>
          <div class = "title">
          </div>
          <div class = "navbar">
          </div>
          <div class="content">
             <asp:ContentPlaceHolder ID="ContentPlaceHolder1" _
 runat="server">
           </asp:ContentPlaceHolder>
        </div>
        <div class="footer">
        </div>
      </div>
   </form>
</body>
</html>
```

Look through Listing 26.2. You can use it to repair any inconsistencies that may have appeared in your version of the project at this stage.

Adding Elements to the Main Master Page

Now that we have created our master pages and set the layout properties, we can start to add elements to the pages that will be inherited by any content pages attached to the master pages.

A useful technique is creating custom user controls or web user controls. The advantage of creating a user control is that you can combine a number of controls and appropriate business logic into a single reusable object that you can then apply across as many pages as necessary.

The control can be centrally managed, which is an advantage if you have used the control across a number of master, content, or normal ASPX pages.

For our purposes, we will create two examples to illustrate the technique:

- The first user control is a simple header object that demonstrates the technique for creating and consuming a web user control. It combines a number of controls, HTML elements, and style elements.
- The second user control is a simple footer control that we will use across the two master pages. In this example, there is information (the year) that must be updated annually. This process is simplified by encapsulating the footer into a user control.

We will begin by creating a web user control to act as the website header.

Creating the Web User Control

To create the web user control, do the following:

1. Choose Website ➤ Add New Item, and choose the Web User Control option from the Add New Item dialog box. Name the control **`MainHeader.ascx`** and click the Add button. This opens `MainHeader.ascx` as a blank template in Design view.
2. Add a Table control from the HTML controls to the design surface. Switch to Source view and edit the HTML for the control so that the table has only one row and two columns. The code should appear as follows:

   ```
   <table style="width:100%;>
   <tr>
   <td>&nbsp;</td>
   <td>&nbsp;</td>
   </tr>
   </table>
   ```

   If the style attribute for width hasn't been added, you can add it manually. Note that the `&nbsp;` entries explicitly create blank white space entries in each table cell.
3. Switch back to Design view and set the following attributes for the table by editing the entries in the Properties box:

   - `Border`: 0
   - `CellPadding`: 10
   - `Style: width:`100%`; height`: 80px

   When you add the `Style` attribute for height, you can either type directly into Source view or expand the ellipsis for the Style entry in the Properties tab and use the Modify Style dialog box shown in Figure 26.3.

   You need to be careful when selecting the table to modify its properties because clicking inside an individual cell opens the Properties box for that particular cell. You can explore this by clicking in the right-hand cell of the table and setting the `align` property to **right**.

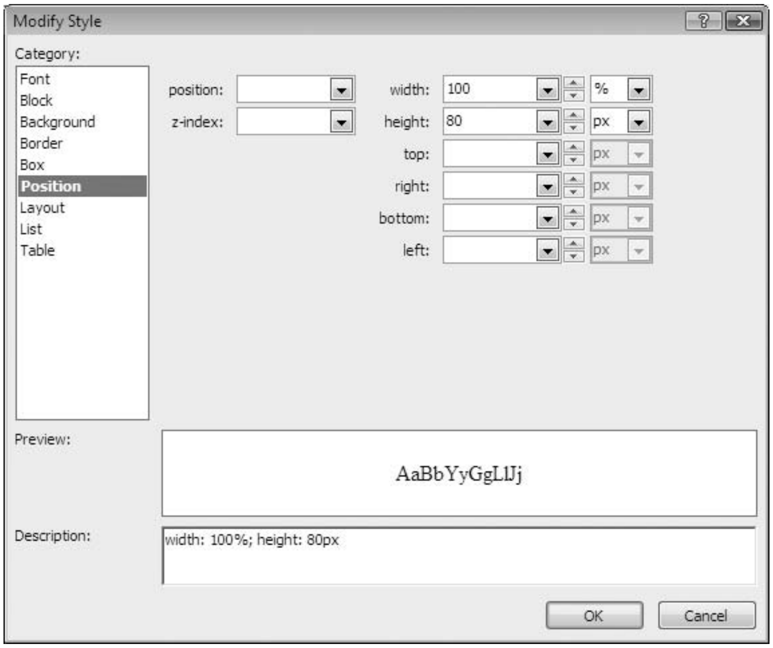


FIGURE 26.3 The Modify Style dialog box

width: 100%; height: 80px

4. Add a Label control to Design view and drag it inside the left-hand cell of the table. Set the following properties for the Label control:
   - `(ID)`: Label1
   - `EnableViewState`: False
   - `Font – Name`: Impact
   - `Font – Size`: XX-Large
   - `ForeColor`: #990000
   - `Text:`Wilson's Computer Parts – Online
5. Add a LinkButton control to the design surface and drag it inside the right-hand cell of the table. Set the following properties for the LinkButton control:
   - `(ID)`: LinkButton1
   - `EnableViewState`: False
   - `Font – Bold`: True
   - `Font – Size`: Medium
   - `ForeColor`: #990000
   - `Text`: >> home

Note that we have disabled the `EnableViewState` property for both of these controls because leaving it enabled would slightly degrade performance for no actual benefit to the application.

6. Double-click the LinkButton control from Design view to open the `LinkButton1_Click` event handler in code-behind (`MainHeader.ascx.vb`). Add the following line to the handler:

```
Response.Redirect("default.aspx")
```

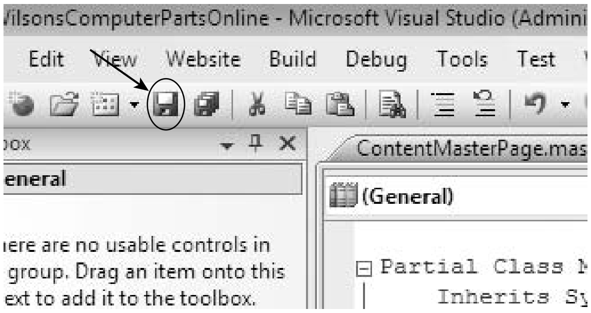
The code for the `Click` handler should now look like this:

```
Protected Sub LinkButton1_Click(ByVal sender As Object, _
 ByVal e As System.EventArgs) Handles LinkButton1.Click
   Response.Redirect("default.aspx")
End Sub
```

The purpose of the LinkButton control in this example is to provide a link back to the main page of the website. In this example, we have used a server-side control employing the Response object to redirect to the main page. We could have also used a simple client-side link or the `Server.transfer` method.

7. Save your work at this stage by using the Save All option from the Standard toolbar, as shown in Figure 26.4.

FIGURE 26.4
Save All toolbar option



ilsonsComputerPartsOnline - Microsoft Visual Studio (Admini
Edit
View
Website
Build
Debug
Tools
Test
ox
eneral
ContentMasterPage.mas
(General)
ere are no usable controls in
group. Drag an item onto this
ext to add it to the toolbox.

```
Partial Class M
    Inherits Sy
```

You have now completed the user control. The full code for the `MainHeader.ascx` control is given in Listing 26.3. (Delete the line breaks.)

LISTING 26.3: The *MainHeader.ascx* Web User Control

```
<%@ Control Language="VB" AutoEventWireup="false" _
 CodeFile="MainHeader.ascx.vb" Inherits="MainHeader" %>
<table style="width:100%;height: 80px;" border="0" cellpadding="10">
   <tr>
      <td>
         <asp:Label ID="Label1" runat="server" Font-Bold="False" _
 Font-Names="Impact" Font-Size="XX-Large" ForeColor="#990000" _
 Text="Wilson's Computer Parts - Online" EnableViewState="False"> _
</asp:Label>
      </td>
      <td align="right">
         <asp:LinkButton ID="LinkButton1" runat="server" _
 EnableViewState="False" ForeColor="#990000"  _
 Font-Underline="False" Font-Bold="True" Font-Size="Medium"> _
 &gt;&gt; home</asp:LinkButton>
      </td>
   </tr>
</table>
```

Adding the Web User Control to Your Page

From the Solution Explorer, double-click the entry for `MasterPage.master` to open the page in Design view. Drag and drop `MainHeader.ascx` from the Solution Explorer into the title area of the `MasterPage.master` layout on the main Design surface. Your `MasterPage.master` should now look like Figure 26.5.

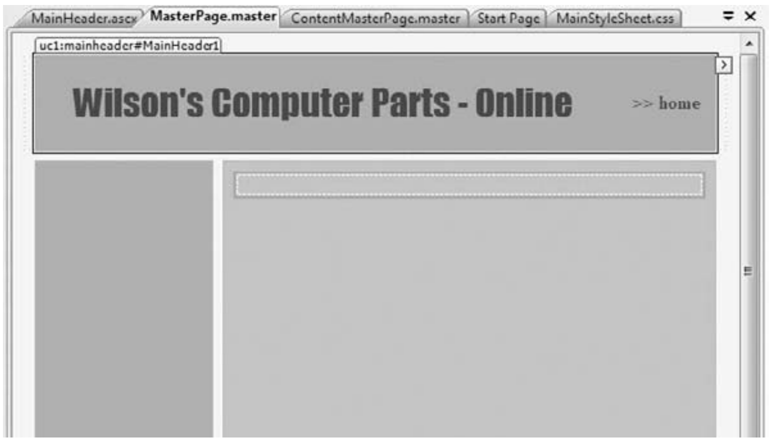


FIGURE 26.5 `MasterPage.master` with the `MainHeader.ascx` Web User control

If you select the instance of the web user control that you have dropped onto your page, you will see that you can set some properties for the control from the Properties box. Use the Properties box to set the `EnableViewState` property to False.

We will use the MainHeader user control to set the heading of the `ContentMasterPage.master` as well. Open `ContentMasterPage.master` in Design view and drag an instance of the `MainHeader.ascx` control from Solution Explorer into the title area of the `ContentMasterPage.master` layout. Set the `EnableViewState` property of the MainHeader web user control to False.

The next step is to create a footer user control for the master pages. The Footer control will be used across all pages and can be created as a web user control.

Creating the *Footer.ascx* Web User Control

The Footer control is a simple example of a reusable control that has updateable content. In this case, if you wish to update the calendar year in the control, it can be managed from a single interface. Complete the following steps:

1. From the Website menu, click Add New Item to open the Add New Item dialog box.
2. Create a new web user control. Name the control **`Footer.ascx`** and click Add. This opens the Design surface for `Footer.ascx`.
3. Add a Table control from the HTML tools to the Design surface. Switch to Source view and edit the table code as follows:

```
<table style="width:100%;" border="0" cellpadding="2">
    <tr>
        <td>
            &nbsp;</td>
        <td align="right">
            &nbsp;</td>
    </tr>
</table>
```

4. Continue to edit the table code by adding `<p>` tags, special characters, and text as shown in the following code snippet:

```
<table style="width:100%;" border="0" cellpadding="2">
    <tr>
        <td>
            <p>&copy; Wilson&#39;s Computer Parts P/L</p></td>
        <td align="right">
            <p>2007</p></td>
    </tr>
</table>
```

5. You can now switch back to Design view and add font and size styles to the `<p>` tags. Click in the left-hand table cell. A small *p* symbol should appear above the table cell on the left side to indicate that the `<p>` tag is being actively edited. In the Properties box, you can select the `Style` property for the `<p>` tag. Click the ellipsis to open the Modify Style dialog box. Under the Font category, set the font-family style to Verdana and the font-size style to Small. Repeat this for the `<p>` tag of the right-hand table cell. The final markup for the Footer control is shown in Listing 26.4. (Delete any line breaks.)

   There are other ways to set these styles. You could have applied a style sheet, used the `<font>` tag directly in Source view, or written the styles directly in Source view.

6. Save all at this stage and switch back to `MasterPage.master` in Design view. Drag an instance of the `Footer.ascx` control from the Solution Explorer into the footer area of the `MasterPage.master` layout. Set its `EnableViewState` property to False.

7. Switch to the `ContentsMasterPage.master` in Design view and drop an instance of `Footer.ascx` into the footer area of this page. Again, set the `EnableViewState` property of the control to False.

LISTING 26.4: Final Markup for the *Footer.ascx* Web User Control

```
<%@ Control Language="VB" AutoEventWireup="false" _
CodeFile="Footer.ascx.vb" Inherits="Footer" %>
<table style="width:100%;" border="0" cellpadding="2">
   <tr>
      <td>
         <p style="font-family: Verdana;font-size: small"> _
&copy; Wilson&#39;s Computer Parts P/L</p></td>
      <td align="right">
         <p style="font-family: Verdana;font-size: small"> _
2007</p></td>
    </tr>
</table>
```

Note the use of the escape character (`&#39;`) instead of the apostrophe in *Wilson's*. We use this character to ensure that the script engine isn't confused between symbols that we wish displayed as text and those used as instructions. A list of common escape codes can be found at `www.petterhesselberg.com/charcodes.html`.

Figure 26.6 illustrates `MasterPage.master` with the `Footer.ascx` control added.

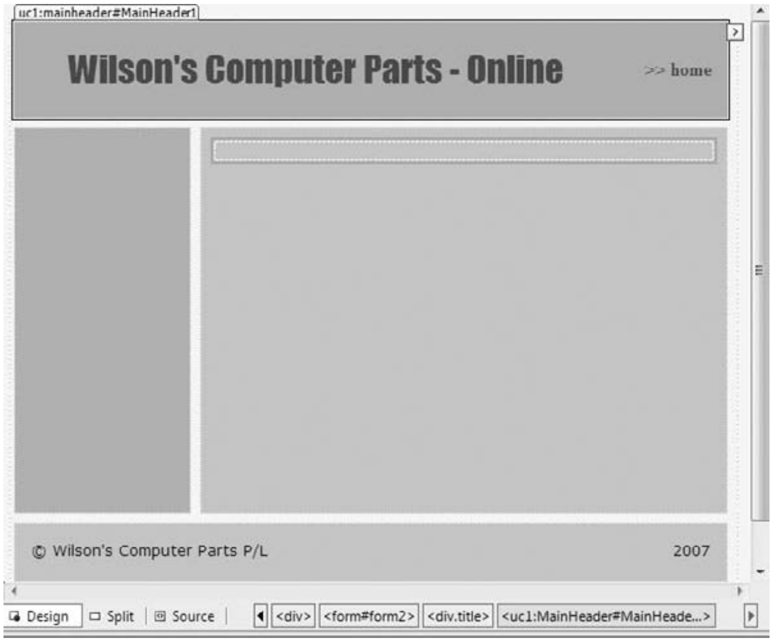


FIGURE 26.6 `MasterPage.master` with the `Footer.ascx` web user control

uc1:mainheader#MainHeader1
Wilson's Computer Parts - Online
>> home
© Wilson's Computer Parts P/L
2007
Design
Split
Source
<div>
<form#form2>
<div.title>
<uc1:MainHeader#MainHeade...>

Building the Site Navigation

Maintaining site navigation can be a major task for the web developer in terms of meeting the need for adding and removing links reliably and seamlessly across what can be hundreds of pages for large sites. Most enterprise-level web-development packages offer this level of functionality. In Visual Studio 2008, site navigation can be managed by using the controls from the Navigation toolbox and the SiteMap template.

In our sample web application, we investigate two methods for setting up site navigation elements that can be dynamically updated from a central source:

- Setting up a navigation menu in the menu area of `MasterPage.master`
- Creating a path of links from the root page to the current page in the navbar area of `ContentMasterPage.master`

We will begin by creating the menu area for `MasterPage.master`. We will use the Navigation tools from the Navigation toolbox. At this stage, we do not have any URLs to add as menu items, so we will simply set up the menu control for later editing.

To use the menu control, you can either statically add items or have the control dynamically populated from a data source such as a SiteMap located in the root of the site. In this example, we will use a SiteMap.

Creating a SiteMap

To create a SiteMap, follow these steps:

1. From the Website menu, choose Add New Item to open the Add New Item dialog box.
2. Click the SiteMap option. Keep the default name of `Web.sitemap` and click the Add button.
3. `Web.sitemap` is an XML document. Edit the first node of the document to read as follows:

```
<siteMapNode  url="Web.sitemap"  title="Sitemap"   description="Sitemap">
```

As you add pages to your website, you can add additional nodes to the SiteMap. Depending on the complexity of your site, the nodes can be added at different levels to aid navigation by providing submenu and rollout navigation menus for your site.

In this example, we have used the SiteMap as the root node because it is our only available page. Later we will modify this node as we add pages to the site and create the true default page.

Configuring the Menu Control for *MasterPage.master*

Now that we have set up a SiteMap, we can apply it to navigation elements in our master pages. We will begin with the menu section of `MasterPage.master`. Complete the following steps:

1. In Solution Explorer, double-click `MasterPage.master` to open the page in Design view.
2. Drop a Menu control from the Navigation toolbox onto the menu part of the `MasterPage.master` layout. It automatically opens a Common Menu Tasks dialog box, where you can statically edit menu items, attach a DataSource linking to the SiteMap, and set styles. To select a style, click the Auto Format option and choose Colorful. You can also switch to Source view and directly edit these styles once applied.
3. Click Choose Data Source and then select New Data Source from the drop-down box. This opens the Data Source Configuration Wizard. Select the SiteMap option, keep the default name of `SiteMapDataSource1`, and click OK.

4. In Design view, you will see a `SiteMapDataSource1` object created under the Menu control. Click `SiteMapDataSource1` and in the Properties window, click the ellipsis in the StartingNodeUrl and select `Web.sitemap`. If you set the `StartingNodeOffset` property to 1, it will keep the Menu control large and manageable if you wish to continue editing it. (Otherwise, the Menu control tends to disappear behind the SiteMapDataSource.) You will need to change this property back to 0 later on.
5. Switch to Source view and edit `Font-Size` in the source of `Menu1` to read **0.9em**. This makes the size of the links a little more agreeable.
6. Finish by saving your work.

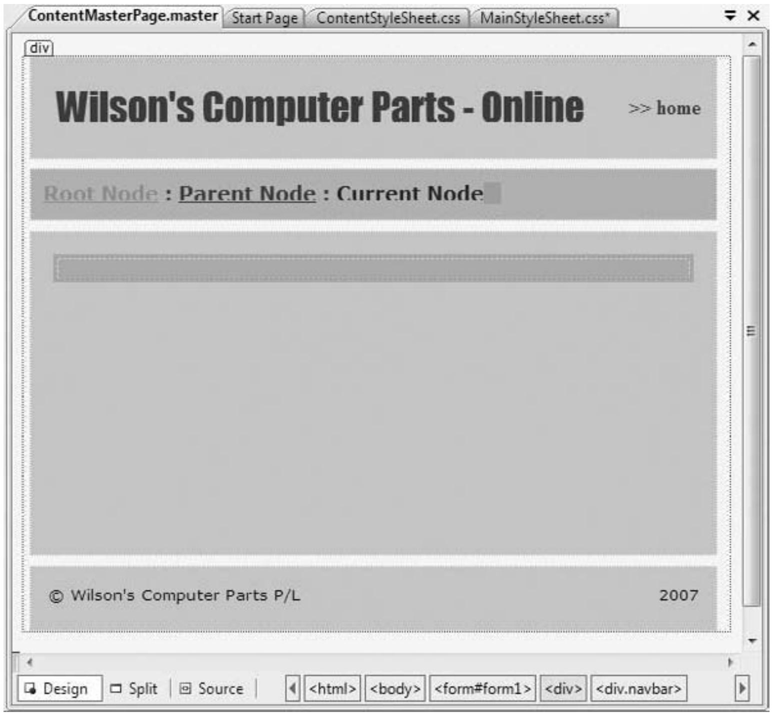
Creating the Navbar in *ContentsMasterPage.master*

In this example, we will use the SiteMapPath control to display the location of the current page as a path of links relative to the root of the site. The SiteMapPath control displays the location of the current page relative to its position in the SiteMap hierarchy. Simply drag and drop a SiteMapPath control from the Navigation toolbox into the Navbar area of the `ContentsMasterPage.master` layout.

A Common Tasks dialog box will open with the control. Select the Auto Format option and choose Colorful. Again, you can switch to Source view and edit the styles for this control.

Switch to Source view and alter the `Font-Size` attribute for the <`SiteMapPath`> tag to read **1.0em**. Figure 26.7 illustrates how this page should now look in Design view.

FIGURE 26.7 `ContentsMasterPage.master` with SiteMapPath control



Adding Authentication

Giving your users the ability to log in to your website has never been easier than with ASP.NET 3.5. A range of controls is available from the Login section of the Toolbox menu that enables you to create a login form, new user wizard, and logged-in user status indicators. You have the option of two types of authentication: Windows authentication for an application designed to run within a local area network (LAN) or forms-based authentication for use with an application run over the Internet.

Using the Login Control

The Login control is a combination control (involving TextBox, Label, and Validation controls) that enables a user to enter a username and password for access to your site.

Double-click the entry for `MasterPage.master` in Solution Explorer to open the page in Design view. From the Login tools in the Toolbox, add an instance of the Login control to the menu area of your master page (underneath the existing Menu control). You may wish to add a Horizontal Rule control from the HTML tools between Menu and Login controls.

A Common Login Tasks dialog box appears next to the Login control. Select the Auto Format option and choose the Colorful style. The page should appear as shown in Figure 26.8.

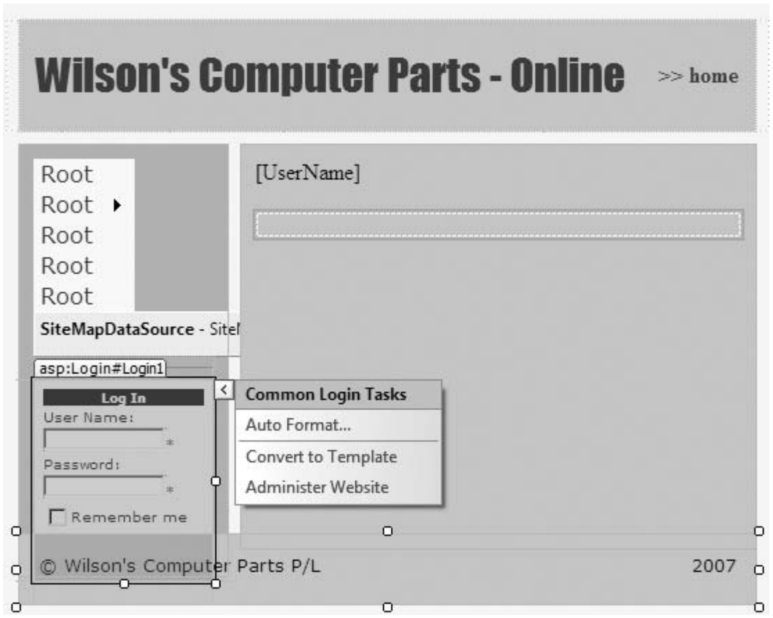


FIGURE 26.8
Adding the Login control

In the Properties box for the Login control, set the `VisibleWhenLoggedIn` property to False.

Establishing Forms-Based Authentication

The next step is to establish the type of authentication your users will employ to access your site. As mentioned earlier, there are two main types:

- Windows authentication
- Forms-based authentication

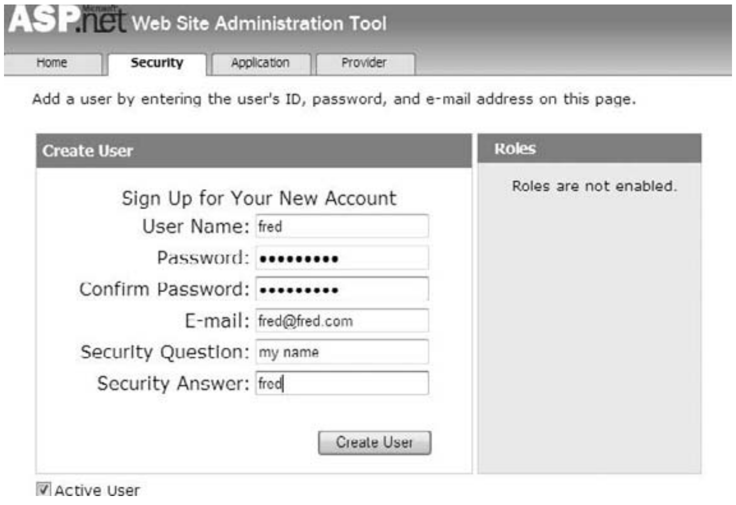
Forms-based authentication enables your users to create new logins, change passwords, and manage their accounts over the Internet. Windows authentication is very secure, but your users need to have been set up with a Windows domain account before they can log in. You can use Windows authentication over the Internet, but it would be for company employees accessing resources rather than for the general public accessing your services.

To set up authentication, select the Administer Website option from the Common Login Tasks dialog box of the Login control. Alternatively, you can open the Website menu and choose ASP.NET Configuration.

This opens the Web Site Administration tool. Select the Security tab to open the section where you can configure your authentication options. Click the Select Authentication Type link and choose the From The Internet option. This sets up your site to use forms-based authentication.

Click Done to return to the main Security screen. From this screen, you can create and manage users, set roles, and create and manage access rules. By default, Visual Studio creates an instance of SQL Server Express to store all relevant site information. If you wish to use a different data store, you can change this from the Provider tab.

To begin configuring the authentication, you need to create a new user. Click the New User link and complete the New User form with appropriate details. Figure 26.9 illustrates the use of this form. Save your user and click back to the main Security page.



Add a user by entering the user's ID, password, and e-mail address on this page.

Create User

Sign Up for Your New Account

User Name:	fred
Password:	●●●●●●●●●
Confirm Password:	●●●●●●●●●
E-mail:	fred@fred.com
Security Question:	my name
Security Answer:	fred

Roles

Roles are not enabled.

☑ Active User

FIGURE 26.9
Using the New User form

Adding an Access Rule

Now that you have a user, you will want to set up an access rule that will limit your site to authenticated visitors. When building your site, you can set up subdirectories within your site containing various pages. You can apply access rules to each of these subdirectories that limit access to either individual users, authenticated users, or particular roles that you may have defined; for example, you may have created a Sales Representative role. You can assign various users to this role and then limit access to a particular directory containing resources relevant to sales representatives to the Sales Representative role. To create an access rule, do the following:

1. Select the Create Access Rules link.
2. Select the root of the WilsonsComputerPartsDirectory. Under Rule Applies To, click the Anonymous Users radio button.
3. Under Permission, click Allow.
4. Click the OK button to return to the main Security screen.

You can check your rule by clicking the Manage Access Rules link. If you maintain this level of security for your site, you will need to create a `login.aspx` page. This page will become the default page for your site and will need to contain all the necessary controls for users to authenticate or, where appropriate, set up new accounts.

For our purposes, we will relax security and keep the site fully open to all users. Click the Manage Access Rules link and delete the access rule that you have created. You will notice that a default rule exists that cannot be edited, which allows all users access to the site. Normally, as you add your own access rules, the additional rules override and limit the default rule.

After you have finished establishing your security settings, close the Administration tool. You may be confronted by a dialog box asking about modifying `Web.config` and whether you wish to reload it. Click the Yes To All option.

Adding a LoginName Control to *MasterPage.master*

The LoginName control displays the login name of the current authenticated user. It is useful for personalizing the pages of your site and also provides a handy security check.

We will add the LoginName control to both of our master pages. Begin by opening `MasterPage.master` in Design view by double-clicking its entry in Server Explorer. Then complete the following steps:

1. From the Login section of the Toolbox, add a LoginName control to the content section of your master page above the `<ContentPlaceHolder>`. You may find it easier to add the control and then switch to Source view to move the control to the appropriate place. This is illustrated in the following code snippet:

   ```
   <div class="content">
   <asp:LoginName ID="LoginName1" runat="server" />
   <asp:ContentPlaceHolder id="ContentPlaceHolder1" runat="server">
      <p>
      </p>
   </asp:ContentPlaceHolder>
   </div>
   ```

2. Add a Label control just before the LoginName control and set the `Text` property to **Welcome**. Keep the `ID` property as (Label1).

3. Double-click somewhere on your page to open up code-behind at the Page_Load code skeleton. We will now add some functionality to determine whether the user is authenticated and to show the Welcome label only if the username is present. Complete the code skeleton for the Page_Load sub as shown in Listing 26.5.

Listing 26.5: *Page_Load* Routine to Hide/Show Welcome Label

```
Protected Sub Page_Load(ByVal sender As Object, ByVal _
 e As System.EventArgs) Handles Me.Load
   If Request.IsAuthenticated = False Then
      Label1.Visible = False
   Else
      Label1.Visible = True
     End If
 End Sub
```

Adding a LoginName Control to *ContentMasterPage.master*

Using the same approach as you used for MasterPage.master, add a LoginName control and a Label control to your ContentMasterPage.master. Set the Label control's text property to Welcome, and use Listing 26.5 to create the show/hide functionality.

Later we will create pages for handling new logins and password change and retrieval.

ContentMasterPage.master is now complete. You can obtain the full ASPX source code for ContentMasterPage.master from the website for this book.

Adding Content Pages

Content pages are the pages that your visitors actually see when they visit your site. They inherit their structure (and much of their content) from their associated master page. We will begin by adding the first content page to the site. This will be the main default page that visitors to the site will initially experience.

There are three main ways to add a content page to your project:

- Right-click MasterPage.master in the Solution Explorer and choose Add Content Page from the context menu. Right-click the newly created page in Solution Explorer and choose the Rename option. Name the page **Default.aspx**. (This should be the default name.)
- With the MasterPage.master open in Design view, choose Website ➢ Add Content Page. If necessary, rename the page as described in the preceding list item.
- Use the Add New Item dialog box from the Website menu. Choose the Web Form option and select the Select Master Page check box. Give the page an appropriate name in the Name text box. Click the Add button to open the Select A Master Page dialog box. Choose the master page and click OK.

The third method described is the preferred method for creating content pages, particularly if you have to rename them. If you have renamed a page by using the context menu in Solution Explorer (as in the first two methods), the new name will need to be set in the Inherits property of the Page directive, and for the Partial class in code-behind. To do this, switch to Source view and set the `Inherits` property in the Page directive to the name of the page. For example, for `Default.aspx` use Default. Then open code-behind for the page (`Default.aspx.vb`) and set the Partial class to the name of that page as well.

After you have created your page in Solution Explorer, right-click `Default.aspx` and choose the Set As Start Page option from the context menu. Select `Default.aspx` in Design view and set the `Title` property for the document in the Properties box to **Wilson's Computer Parts Online - Home**.

Adding an Entry to the SiteMap

We can now update the SiteMap to reflect the new page. Switch to the SiteMap by selecting the Web.sitemap tab and alter the root siteMapNode node to read as follows:

```
<siteMapNode url="Default.aspx" title="Home"  description="Home" />
```

The code listing for the SiteMap should now be as shown in Listing 26.6.

Listing 26.6: *Web.sitemap*

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
   <siteMapNode url="Default.aspx" title="Home"  description="Home">
      <siteMapNode url="" title=""  description=" " />
      <siteMapNode url="" title=""  description="" />
   </siteMapNode>
</siteMap>
```

Updating the SiteMapDataSource Control

To update the control, switch to `MasterPage.master` and select the SiteMapDataSource control that you have used in your page. In the Properties box, set the `StartingNodeURL` property to ~/Default.aspx. You can either type this directly or click the ellipsis and select the file. You will also need to ensure that the `StartingNodeOffset` property has been set to 0.

If you leave the `ShowStartingNode` property at its default value of True, the Menu control will be rendered as a rollout menu from the Home node in the menu column. Set the value to False, and the individual subnodes of the SiteMap will be listed separately in the menu column. In this example, we will use the False setting.

Running the Application

Running the application should produce a web page like the one shown in Figure 26.10. Note that when running the site for the first time by using the start arrow, you will be asked whether you wish to turn on debugging, as shown in Figure 26.11. It's a good idea to turn on debugging while building the application, but you will need to disable it in the `Web.config` file before you distribute the site. This was discussed in Chapter 25.

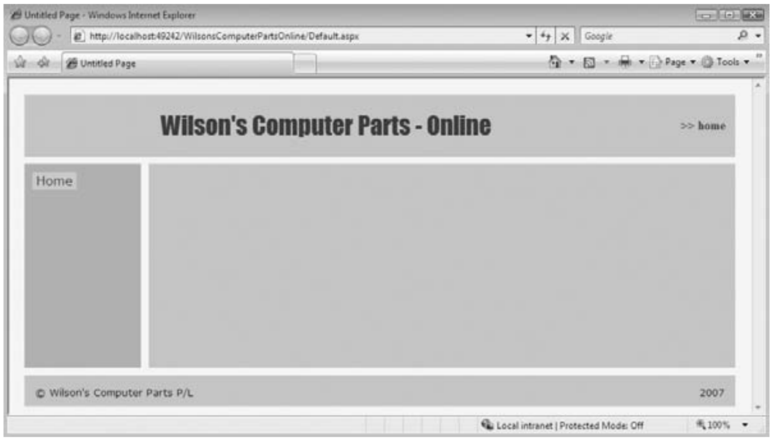

Untitled Page - Windows Internet Explorer
http://localhost:49242/WilsonsComputerPartsOnline/Default.aspx
Wilson's Computer Parts - Online
>> home
Home
© Wilson's Computer Parts P/L
2007
Local intranet | Protected Mode: Off

FIGURE 26.10
The running application

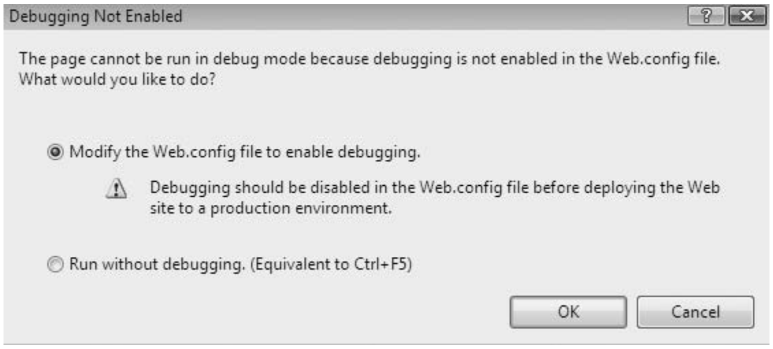


FIGURE 26.11
Turning on debugging

Debugging Not Enabled

The page cannot be run in debug mode because debugging is not enabled in the Web.config file. What would you like to do?

[✓ Modify the Web.config file to enable debugging.]

Debugging should be disabled in the Web.config file before deploying the Web site to a production environment.

Run without debugging. (Equivalent to Ctrl+F5)

OK Cancel

You can now add controls and content to the ContentPlaceHolder that you set up in the content section of the main page layout. Remember to set the ContentPlaceHolder control to Create Custom Content from the Common Content Tasks rollout menu; otherwise, you will not be able to add content to the `ContentPlaceHolder1` control.

Typically, when adding content, you use combinations of server-side controls such as Label, TextBox, and Image. These are particularly useful for data-driven applications, where content is managed from a database. In situations where you do not need to use server-side functionality, it is a good idea to use the controls from the HTML toolbox because they have a smaller footprint and improve application performance.

You can also add your content to the ASPX source code (in Source view) by using HTML tags and writing your HTML directly on the page. You can set styles for your HTML tags in a number of ways, either through your CSS style sheet or applying styles directly to the tags or to the page. You can use the various style-builder tools in Visual Studio to assist in this process.

In addition, in Design view, text can be written directly to the design surface. You can format the text by using the Formatting toolbar.

Adding Further Content Pages

We will now add content pages based on the `ContentsPageMaster.master` template. These will become the daughter pages of the website. Complete the following steps:

1. From the Website menu, choose Add New Item to open the Add New Item dialog box.
2. Select the Web Form template and rename the page **`Password.aspx`**. Select the Select Master Page check box and click the Add button. This opens the Select A Master Page dialog box.
3. Select `ContentMasterPage.master` and click OK. This opens `Password.aspx` in Design view. In the Properties box for `Password.aspx`, set the `Title` property for the document to **Wilsons Computer Parts Online – Password Management**.
4. Repeat these steps to add three more pages and name the pages **`NewUser.aspx`**, **`Parts.aspx`**, and **`Computers.aspx`**.
5. In the Properties box for `NewUser.aspx`, set the `Title` property for the document to **Wilsons Computer Parts Online – New User**; for `Parts.aspx`, set it to **Wilsons Computer Parts Online – Parts**; and for `Computers.aspx`, set it to **Wilsons Computer Parts Online – Computers**.

Adding Items to the SiteMap

We will include the `Parts.aspx` and `Computers.aspx` pages in the SiteMap. Switch to the `Web.sitemap` page and add entries for the Parts Catalog and Computers pages as shown in Listing 26.7 (you can ignore the line breaks). The `NewUser.aspx` and `Password.aspx` pages will be kept out of the SiteMap because we will set up command button–based navigation for them.

LISTING 26.7: Updated SiteMap Code

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
   <siteMapNode url="default.aspx" title="Home"  description="Home">
      <siteMapNode url="parts.aspx" title="Catalogue" _
 description="Catalogue" />
      <siteMapNode url="computers.aspx" title="Computers" _
 description="Computers" />
   </siteMapNode>
</siteMap>
```

At this point, use the Save All command to save your work.

Using Buttons for Navigation

Rather than use SiteMap entries for the NewUser and Password pages, we will add three Button controls under the Login control in `MasterPage.master` and attach custom navigation to the buttons.

From Solution Explorer, switch to `MasterPage.master` in Design view. Complete the following steps:

1. Add three Button controls immediately under the Login control in the menu area of the layout.
2. Stretch the width of each Button control to 130px. You can either use the mouse handles on the control or set the `width` property in the Properties box.
3. Set the `Text` property of `Button1` to **Change Password**, for `Button2` to **Recover Password**, and for `Button3` to **Create New Login**.
4. Double-click `Button1` to enter the handler for the `Click` event in code-behind (`MasterPage.master.vb`). Listing 26.8 gives the completed handlers for each of the three buttons.

In the `Click` events, we use two session variables (`ChangePassword` and `RetrievePassword`) to record which button has been clicked. This is used because the Password Recovery control and Change Password control will be set up on the one page. We will display the active control to the user that is determined by which of the buttons is clicked.

We then use the `Response.Redirect` method to open the Password page. You can also use the `Server.Transfer` method to achieve a similar result. Be aware, however, that `Server.Transfer` does not always play nicely with the SiteMapPath control. If you were to add a page to the SiteMap and use the `Server.Transfer` method to navigate to the page, the path to the page would not be fully articulated in the SiteMapPath control.

LISTING 26.8: *Click* Events for the Three Navigation Buttons

```
Protected Sub Button1_Click(ByVal sender As Object, ByVal e As _
 System.EventArgs) Handles Button1.Click
   Session("ChangePassword") = True
   Session("RetrievePassword") = False
   Response.Redirect("~/Password.aspx")
End Sub

Protected Sub Button2_Click(ByVal sender As Object, ByVal e As _
 System.EventArgs) Handles Button2.Click
   Session("ChangePassword") = False
   Session("RetrievePassword") = True
   Response.Redirect("~/Password.aspx")
End Sub

Protected Sub Button3_Click(ByVal sender As Object, ByVal e As _
 System.EventArgs) Handles Button3.Click
   Response.Redirect("~/NewUser.aspx")
End Sub
```

At this point, `MasterPage.master` is complete. The website for this book contains the full ASPX source code for `MasterPage.master`.

Building the Password Page

The next step is to build the Password page. From Solution Explorer, open `Password.aspx` in Design view. Complete the following steps:

1. Set the ContentPlaceHolder control to Create Custom Content from the Common Content Tasks rollout menu.
2. From the Login controls in the Toolbox, drag and drop a ChangePassword control onto the design surface.
3. From the rollout Common ChangePassword Tasks dialog box, use the Auto Format option to set the Scheme to Colorful. You can use the drop-down Views box to see how each of the different stages of the ChangePassword control will appear. Note that you should leave the Views box set to the first view of the control, because this will be the view that the user will first see.
4. There is a wide range of properties available that you can change to customize this control. In the Properties box for the ChangePassword control, set the following properties:
   - `CancelDestinationPageUrl`: ~/Default.aspx
   - `ContinueDestinationPageUrl`: ~/Default.aspx
   - `CreateUseText`: Create New User
   - `CreateUserUrl`: ~NewUser.aspx
5. The next step is to add a PasswordRecovery control to the page. Drag and drop an instance of this control onto the page from the Login toolbox. Keep the default ID of `PasswordRecovery1`.
6. From the rollout dialog for the PasswordRecovery control, use the Auto Format option to set the Scheme to Colorful.
7. In the Properties box, set the `SuccessPageUrl` property to ~/Default.aspx and the `Visible` property to False.
8. We will now use absolute positioning to place the PasswordRecovery control on top of the ChangePassword control. With `PasswordRecovery1` selected, from the main Visual Studio menu bar at the top of the screen choose Format ➤ Set Position ➤ Absolute. You can now move the `PasswordRecovery1` control to exactly where it is to be rendered on the page. Move the control so that it sits over the top of the ChangePassword control.

Relative and Absolute Positioning

Using the Relative and Absolute options from the Format ➤ Set Position menu enables you to precisely control the positioning of controls on your pages. However, be aware that using these options (particularly the Absolute settings) can produce unexpected formatting results depending on individual browser settings.

The PasswordRecovery control requires some additions to the `Web.config` file as well as a functioning Simple Mail Transfer Protocol (SMTP) service on your web server. Listing 26.9 illustrates the additions that you will need to make to the `Web.config` file. (`Web.config` is accessed from the Solution Explorer.) You will need to modify this listing to meet your local requirements and settings. Add the listing somewhere in the `<configuration>...</configuration>` section, after the `<configSections>...</configSections>` tags.

LISTING 26.9: Additional Code to Add to *Web.config* for the PasswordRecovery Control

```
<system.net>
    <mailSettings>
      <smtp deliveryMethod="Network" from="me@my.web.address.com">
        <network
          host="localhost"
          port="25"
          defaultCredentials="true"
        />
      </smtp>
    </mailSettings>
  </system.net>
```

Figure 26.12 illustrates how the `Password.aspx` page should now appear.

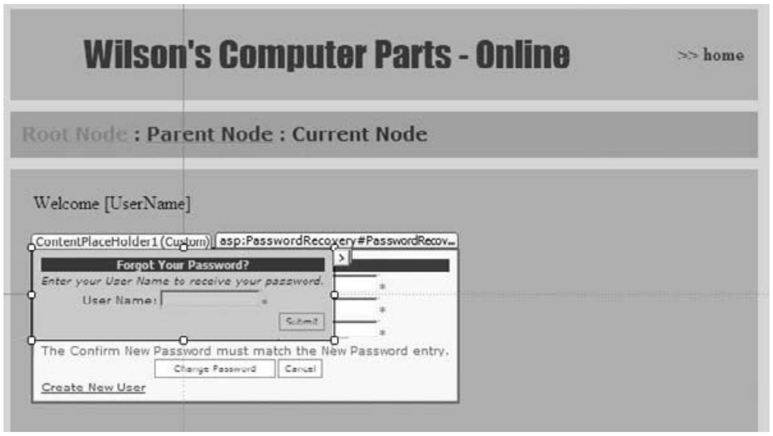

Wilson's Computer Parts - Online
>> home
Root Node : Parent Node : Current Node
Welcome [UserName]
ContentPlaceHolder1 (Custom)
asp:PasswordRecovery#PasswordRecov...
Forgot Your Password?
Enter your User Name to receive your password.
User Name:
Submit
The Confirm New Password must match the New Password entry.
Change Password
Cancel
Create New User

FIGURE 26.12
The completed `Password.aspx` page

WRITING THE CODE-BEHIND FOR *PASSWORD.ASPX*

Double-click the `Password.aspx` page to switch to code-behind (`Password.aspx.vb`) and complete the `Page_Load` sub with the code from Listing 26.10.

LISTING 26.10: Code-Behind for *Password.aspx*

```
Protected Sub Page_Load(ByVal sender As Object, _
ByVal e As System.EventArgs) Handles Me.Load

   If Session("ChangePassword") = True Then
```

```
            ChangePassword1.Visible = True
            PasswordRecovery1.Visible = False

         ElseIf Session("RetrievePassword") = True Then
            ChangePassword1.Visible = False
            PasswordRecovery1.Visible = True

         End If

      End Sub
```

The purpose of Listing 26.10 is to show the appropriate control depending on the state of the `ChangePassword` and `RetrievePassword` session variables.

Save your work and run the application to test the behavior of the Change Password and Recover Password buttons.

You can find the full ASPX source code for the `Password.aspx` page on the website for this book.

Building the *NewUser.aspx* page

From Solution Explorer, switch to `NewUser.aspx` in Design view. Complete the following steps:

1. Set the ContentPlaceHolder control to Create Custom Content from the Common Content Tasks rollout menu.
2. From the Login controls, add a CreateUserWizard control to the page. A rollout Common CreateUserWizard Tasks dialog box opens. Select the Colorful scheme from the Auto Format option.
3. A wide range of properties is available for you to customize the wizard to meet your own needs. In the Properties box for the CreateUserWizard, set the following properties:
   - `CancelDestinationPageUrl`: ~/Default.aspx
   - `ContinueDestinationPageUrl`: ~/Default.aspx
   - `FinishDestinationPageUrl`: ~/Default.aspx

Again, you can check the appearance of each of the steps in this wizard from the Step drop-down box in the Common CreateUserWizard Tasks dialog. Make sure that you leave the control at the first step before deploying your application.

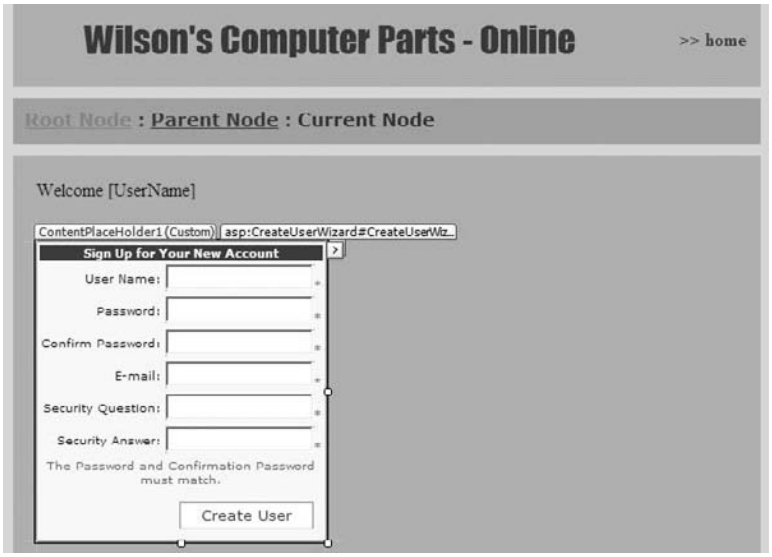
Figure 26.13 illustrates the completed `NewUser.aspx` page.

The full ASPX source code for the `NewUser.aspx` page is available from the Website for this book.

Working with Data

In this section, you will see how to connect to two different data sources (an XML document and a SQL Server database) to create simple data-driven elements in our application. You will explore the GridView control and find out how to create a basic master/detail form by using a combination of the DetailsView and DropDownList controls. You will also see how to create a simple online report by using the MicrosoftReportViewer control.

FIGURE 26.13
The completed `NewUser.aspx` page


Wilson's Computer Parts - Online
>> home
Root Node : Parent Node : Current Node
Welcome [UserName]
ContentPlaceHolder1 (Custom)
asp:CreateUserWizard#CreateUserWiz...
Sign Up for Your New Account
User Name:
Password:
Confirm Password:
E-mail:
Security Question:
Security Answer:
The Password and Confirmation Password must match.
Create User

Visual Studio 2008 ships with the SQL Server Express database package. This is sufficient for creating the SQL Server database that we will be using later in the chapter.

Creating the XML Database

You will begin by creating the XML document that will act as a data source for the GridView control. Complete the following steps:

1. From the Website menu, choose Add New Item to open the Add New Item dialog box. Choose XML File.
2. Name the file **`Parts.xml`** and click the Add button. This will be a simple database of computer parts and prices. Each catalog item has five fields: id, item, type, price, and availability.
3. Add the contents of Listing 26.11 to the XML page. Some long lines are wrapped here in print, but you can leave them all on one line in your code.

LISTING 26.11: *Parts.xml*

```
<?xml version="1.0" encoding="utf-8" ?>
<catalogue>

  <CatalogueItem id="1" item="Random Access Memory" _
 type="256 MB" price="25.00" availability="In Stock" />
  <CatalogueItem id="2" item="Random Access Memory" _
 type="512 MB" price="50.00" availability="In Stock" />
  <CatalogueItem id="3" item="Random Access Memory" _
 type="1 GB" price="80.00" availability="In Stock" />
```

```
  <CatalogueItem id="4" item="Random Access Memory" _
 type="2 GB" price="120.00" availability="Not In Stock" />
  <CatalogueItem id="5" item="Random Access Memory" _
 type="256 MB" price="25.00" availability="In Stock" />
  <CatalogueItem id="6" item="Random Access Memory" _
 type="512 MB" price="50.00" availability="In Stock" />
  <CatalogueItem id="7" item="Random Access Memory" _
 type="1 GB" price="80.00" availability="In Stock" />
  <CatalogueItem id="8" item="Random Access Memory" _
type="2 GB" price="120.00" availability="Not In Stock" />
  <CatalogueItem id="9" item="Hard Drive" _
 type="80 GB" price="100.00" availability="In Stock" />
  <CatalogueItem id="10" item="Hard Drive" _
 type="120 GB" price="120.00" availability="In Stock" />
  <CatalogueItem id="11" item="Hard Drive" _
 type="180 GB" price="150.00" availability="In Stock" />
  <CatalogueItem id="12" item="Hard Drive" _
 type="200 GB" price="180.00" availability="Not In Stock" />
  <CatalogueItem id="13" item="Hard Drive" _
 type="320 GB" price="220.00" availability="In Stock" />
  <CatalogueItem id="14" item="Hard Drive" _
 type="500 GB" price="300.00" availability="In Stock" />
  <CatalogueItem id="15" item="Hard Drive" _
 type="750 GB" price="420.00" availability="In Stock" />
  <CatalogueItem id="16" item="Case" type="Type 1" _
 price="25.00" availability="Not In Stock" />
  <CatalogueItem id="17" item="Case" type="Type 2" _
 price="55.00" availability="In Stock" />
  <CatalogueItem id="18" item="Case" type="Type 3" _
 price="50.00" availability="In Stock" />
  <CatalogueItem id="19" item="Case" type="Type 4" _
 price="80.00" availability="In Stock" />
  <CatalogueItem id="20" item="Case" type="Type 5" _
 price="120.00" availability="Not In Stock" />
  <CatalogueItem id="21" item="Case" type="Type 6" _
 price="25.00" availability="In Stock" />
  <CatalogueItem id="22" item="Case" type="Type 7" _
 price="50.00" availability="In Stock" />
  <CatalogueItem id="23" item="Case" type="Type 8" _
 price="80.00" availability="In Stock" />
  <CatalogueItem id="24" item="Case" type="Type 9" _
 price="120.00" availability="Not In Stock" />

</catalogue>
```

We have included 24 items in this example so that we can demonstrate pagination in the final working example. Feel free to shorten it. Save your work, and from Solution Explorer open the `Parts.aspx` page in Design view.

Working with the GridView Control

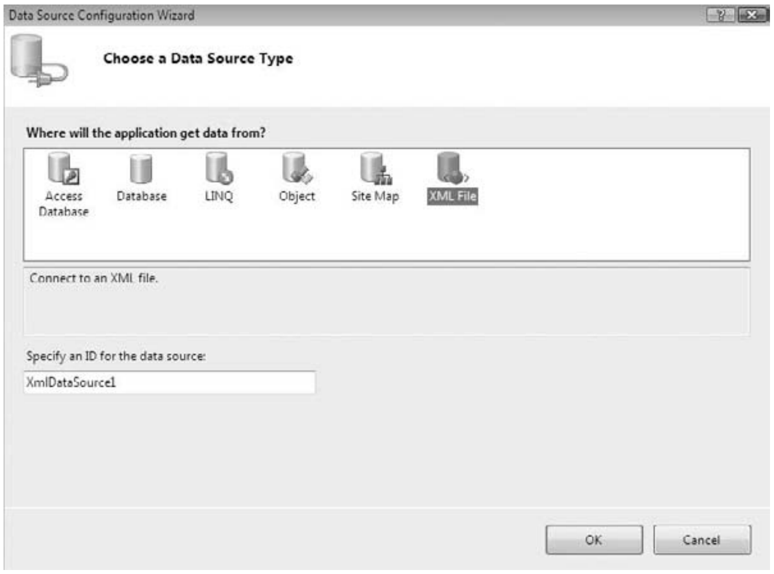
The GridView control enables you to create a tabular view of data. It is the ASPX equivalent of the Windows forms DataGrid control, although the GridView control has less built-in functionality.

You can enable a Select option in the GridView control to allow users to select individual rows. You can then programmatically manipulate the selection in code-behind and perform a range of operations such as returning information to the server, populating a session variable, updating a database, or writing to a file. The GridView control also enables you to automatically generate Update, New, and Delete functionality for an editable database. A wide variety of properties are available to manage the appearance and behavior of the control, including pagination, which allows you to limit the number of rows of data displayed at any one time. In this instance, we will link a GridView control to the `Parts.xml` file.

Begin by setting the ContentPlaceHolder control to Create Custom Content from the Common Content Tasks rollout menu. Then complete the following steps:

1. Drag and drop a GridView control onto ContentPlaceHolder in the main content section of `Parts.aspx` in Design view. This creates an instance of the control with the default name `GridView1`.
2. The Common GridView Tasks rollout dialog box will open. Select the Auto Format option and choose the Mocha scheme. Click OK.
3. Click the Choose Data Source drop-down box and select New Data Source. This opens the Data Source Configuration Wizard, as shown in Figure 26.14.

FIGURE 26.14
Data Source Configuration Wizard


Data Source Configuration Wizard
Choose a Data Source Type
Where will the application get data from?
Access Database
Database
LINQ
Object
Site Map
XML File
OK
Cancel

Connect to an XML file.

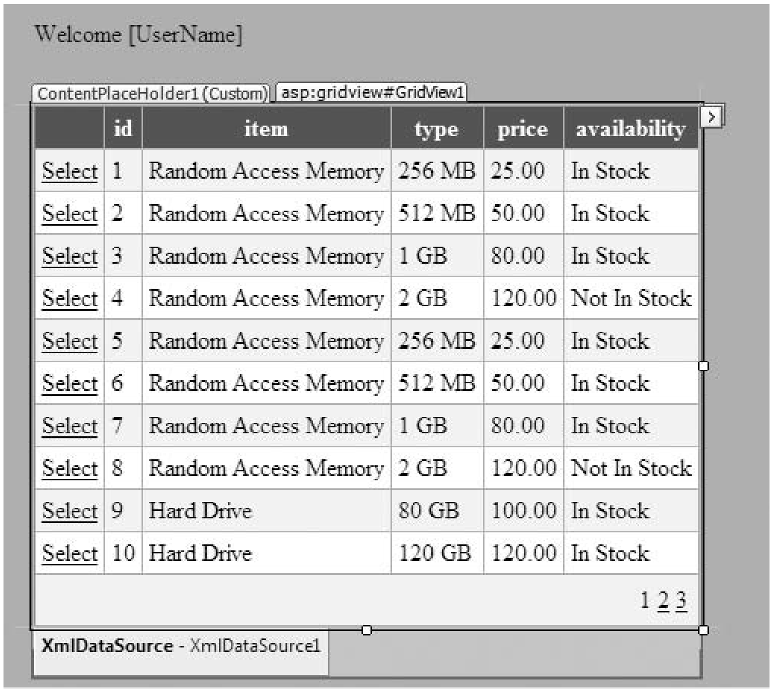
Specify an ID for the data source: XmlDataSource1

4. In the Data Source Configuration Wizard, choose the XML File option; keep the default name of `XMLDataSource1` and click OK.

5. The next screen of the wizard asks you to browse for the data file. Click the Browse button and select `Parts.xml`. In this example, we will not be working with a Transform file or XPath expression, so you can leave these options blank and click OK.
6. You should now be returned to the Common GridView Tasks dialog box, which will be expanded to include two additional check box options: Enable Paging and Enable Selection. Select both check boxes and close the dialog box.

The GridView control should now be populated (in Design view) with some of the data from `Parts.xml`, as shown in Figure 26.15.

FIGURE 26.15
The completed GridView control



Welcome [UserName]

ContentPlaceHolder1 (Custom) asp:gridview#GridView1

	id	item	type	price	availability
Select	1	Random Access Memory	256 MB	25.00	In Stock
Select	2	Random Access Memory	512 MB	50.00	In Stock
Select	3	Random Access Memory	1 GB	80.00	In Stock
Select	4	Random Access Memory	2 GB	120.00	Not In Stock
Select	5	Random Access Memory	256 MB	25.00	In Stock
Select	6	Random Access Memory	512 MB	50.00	In Stock
Select	7	Random Access Memory	1 GB	80.00	In Stock
Select	8	Random Access Memory	2 GB	120.00	Not In Stock
Select	9	Hard Drive	80 GB	100.00	In Stock
Select	10	Hard Drive	120 GB	120.00	In Stock

1 2 3

XmlDataSource - XmlDataSource1

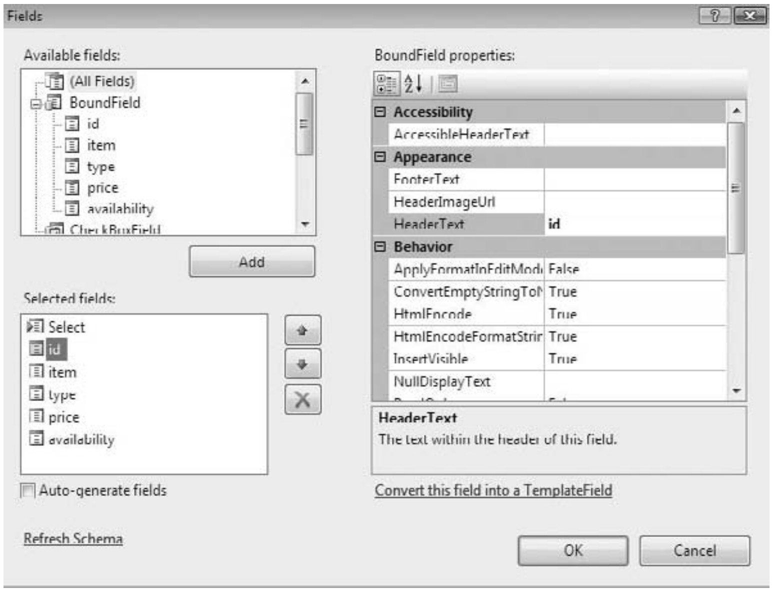
Underneath `GridView1`, you will see that an XMLDataSource object (`XMLDataSource1`) has been created as well. If you need to further edit the data source for the GridView control, you can do it from here, using the rollout dialog box for the XMLDataSource control.

Further Configuration of the GridView Control

After you have the GridView control installed and data bound, you can do a lot to make the control more presentable and usable. As an example, complete the following steps:

1. Select the GridView control and use the resize handles to stretch it out across the screen.
2. Alter the column headings rendered at runtime. Click the small arrow in the top-right corner of the control to open the Common GridView Tasks dialog box, and select the Edit Columns link. This opens the Fields dialog box shown in Figure 26.16.

FIGURE 26.16
The Fields dialog box



Fields

Available fields:
- (All Fields)
- BoundField
  - id
  - item
  - type
  - price
  - availability
- CheckBoxField

Selected fields:
- Select
- id
- item
- type
- price
- availability

☐ Auto-generate fields

BoundField properties:

Accessibility	
AccessibleHeaderText	
Appearance	
FooterText	
HeaderImageUrl	
HeaderText	id
Behavior	
ApplyFormatInEditMode	False
ConvertEmptyStringToNull	True
HtmlEncode	True
HtmlEncodeFormatString	True
InsertVisible	True
NullDisplayText	

HeaderText
The text within the header of this field.

Convert this field into a TemplateField

Refresh Schema

From the Fields dialog box, you can add and remove columns from the GridView control by using the Available Fields box and the Add and Remove (red cross) buttons. You can also change the order of the columns by using the up/down arrow buttons next to the Selected Fields box.

Within the Selected Fields box, you can click any field and edit the properties and appearance of the associated column in the GridView control. In this example, we will set the Header Text property for each of the columns:

- Click the ID field and set the Header Text property to Catalog ID.
- Click the item field and set the Header Text property to Item.
- Click the type field and set the Header Text property to Type.
- Click the price field and set the Header Text property to Price.
- Click the availability field and set Header Text to Availability.

If you click the Select field and examine the properties, you will see the options to turn on Edit, Delete, and Insert buttons. The properties are `ShowEditButton`, `ShowDeleteButton`, and `ShowInsertButton`, respectively. Although not applicable to this example, you can use these buttons to provide additional functionality when you are using an editable database with relevant update, delete, and insert statements. Later, in the section ''Adding the DetailsView Control,'' you will see how this is used.

Working with the Select Button

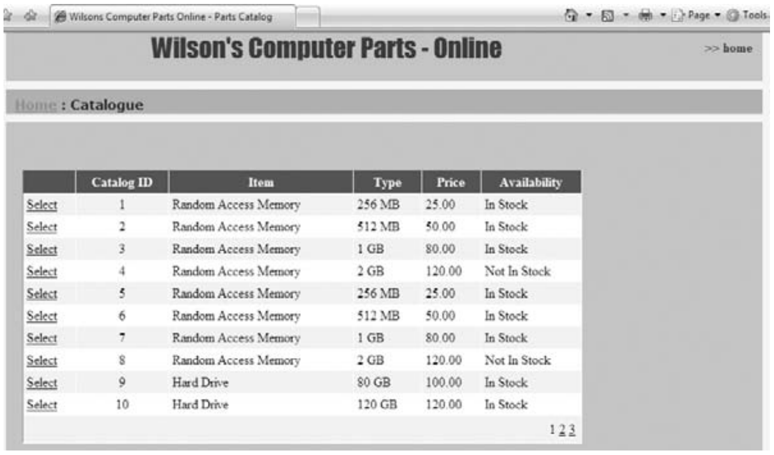
At this stage, we have not given any meaningful employment to the Select button that we have set up in the GridView control. Change to code-behind (`Parts.aspx.vb`) for `Parts.aspx` by double-clicking the page, and select the code skeleton for the `GridView1_SelectedIndexChanged` event. This event fires whenever your user makes a new selection in the GridView control.

You can use `GridView1.SelectedRow` to gather information about your user's selection that you can then use programmatically. For example:

```
ListBox1.Items.Add(GridView1.SelectedRow.Cells(3).Text)
```

This code snippet can be used to populate a ListBox control with the contents of the Type field (column 3) from selections made by a user.

The final step is to run and test `Parts.aspx` and the GridView control. The control should display the first 10 items and paging options to view the remaining items. Figure 26.17 illustrates the running `Parts.aspx` page.



Wilson's Computer Parts - Online

>> home

Home : Catalogue

	Catalog ID	Item	Type	Price	Availability
Select	1	Random Access Memory	256 MB	25.00	In Stock
Select	2	Random Access Memory	512 MB	50.00	In Stock
Select	3	Random Access Memory	1 GB	80.00	In Stock
Select	4	Random Access Memory	2 GB	120.00	Not In Stock
Select	5	Random Access Memory	256 MB	25.00	In Stock
Select	6	Random Access Memory	512 MB	50.00	In Stock
Select	7	Random Access Memory	1 GB	80.00	In Stock
Select	8	Random Access Memory	2 GB	120.00	Not In Stock
Select	9	Hard Drive	80 GB	100.00	In Stock
Select	10	Hard Drive	120 GB	120.00	In Stock

1 2 3

FIGURE 26.17
The running `Parts.aspx` page

You can obtain the full ASPX source code for `Parts.aspx` from the website for this book.

CREATING THE SQL DATABASE

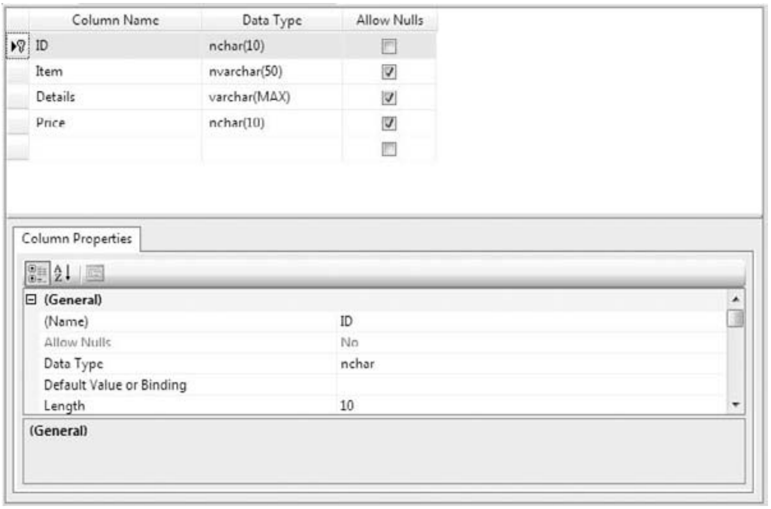
The next step in the project is to build an SQL database that will act as a data source for the `Computers.aspx` page and the Report page that we will build later.

If you have Microsoft's SQL Server installed on your machine, you can use it to create the database; otherwise, the SQL Server Express package that ships with Visual Studio 2008 is quite sufficient. SQL Server Express is a cut-down version of the complete SQL server package.

Complete the following steps:

1. From the Website menu, choose Add New Item to open the Add New Item dialog box. Select the SQL Database option.
2. Name the database **`Computers.mdf`** and click the Add button. You will be presented with a dialog box asking whether you wish to place the database inside the App_Data folder. Click the Yes option. This opens a new entry for `Computers.mdf` in the Data Connections section of the Server Explorer window on the left side of Visual Studio. There should already be an entry for `ASPNETDB.mdf`. In practice, you would most likely have used the existing database and added additional tables.
3. Right-click the Tables entry for `Computers.mdf` and choose the Add New Table option. This opens a table template on the design surface of Visual Studio.

4. Under Column Name, enter **ID**. Keep the default Data Type of nchar10. Right-click the small box to the left of Column Name for the ID field and choose the Set Primary Key option from the context menu.
5. Continue to add three more columns with the following properties:
   - Item: `nvarchar(50)`
   - Details: `varchar(MAX)`
   - Price: `nchar(10)`
6. Click the Save All option. This opens a dialog box asking you to name the table. Name the table **Catalog**. Figure 26.18 illustrates how the database should appear at this stage.



Column Name	Data Type	Allow Nulls
ID	nchar(10)	☐
Item	nvarchar(50)	☑
Details	varchar(MAX)	☑
Price	nchar(10)	☑
		☐

(General)	
(Name)	ID
Allow Nulls	No
Data Type	nchar
Default Value or Binding	
Length	10

FIGURE 26.18 Creating the `Computers.mdf` database

7. In Server Explorer, you can now right-click on the entry for the Catalog table and choose Show Table Data to commence adding some data to the database. You may have to expand `Tables for Computers.mdf` to see the table, and refresh Server Explorer if the entry still doesn't appear.

Table 26.2 illustrates the data to enter into the database.

This is only a simple flat-file database. Visual Studio 2008 allows you to do much more when creating databases. You can set up relationships, write stored procedures, and build complex relational databases.

The next step is to build the `Computers.aspx` page so that we can create database-driven elements for our application.

Building the *Computers.aspx* Page

The purpose of `Computers.aspx` is to display a master/detail form that enables the user to navigate to different database entries from a DropDownList control. The details for the entries are displayed in a DetailsView control, and the user has the option to select and edit the details for individual entries in the database.

TABLE 26.2: Data for *Computers.mdf*

ID	ITEM	DETAIL	PRICE
1	Computer 1	Entry Model	699.00
2	Computer 2	Intermediate Model	899.00
3	Computer 3	Advanced Model	1199.00
4	Computer 4	Business Model	1099.00
5	Computer 5	Gaming Model	1599.00

Adding the DropDownList Control

The DropDownList control will be populated with the list of items from the database. In our master/details scenario, you choose the item for which you want further detail from the DropDownList control.

Begin by opening `Computers.aspx` in Design view from Solution Explorer. To set up the DropDownList control in `Computers.aspx`, complete the following steps:

1. Set the ContentPlaceHolder control in `Computers.aspx` to Create Custom Content from the Common Content Tasks rollout dialog box
2. Begin by placing a DropDownList control from the Standard toolbox onto `ContentPlaceHolder1` of `Computers.aspx` in Design view
3. From the Common DropDownList Tasks rollout dialog box, select Choose Data Source. This opens the Data Source Configuration Wizard
4. From the opening screen of the Data Source Configuration Wizard, choose Select A Data Source and then New Data Source from the drop-down box. This opens the Choose A Data Source Type window .
5. Choose the Database option and keep the default name `SqlDataSource1`. From here you will be taken to the Choose A Connection pane
6. Select the connection string for `Computers.mdf` from the drop-down data connection box. Click the Next button. This takes you to the Configure The Select Statement window. The Catalog table should be identified
7. Check the ID and Item columns
8. Click the Next button to move to the Test Query window. It is a good idea to click the Test Query button at this point to make sure that everything is working properly. Clicking the Test Query button should return the ID and Item values out of the `Computers.mdf` database
9. If everything is okay, click Finish. This returns you to the Choose A Data Source page of the wizard .
10. From the Data Field To Display drop-down list, choose Item. From the Select A Data Field For The Value drop-down list, choose ID. Click OK to exit the wizard

11. Finally, from the Properties box for the DropDownList, set the `AutoPostBack` property to True. This sets the control to automatically post back to the server every time a change of selection is made.

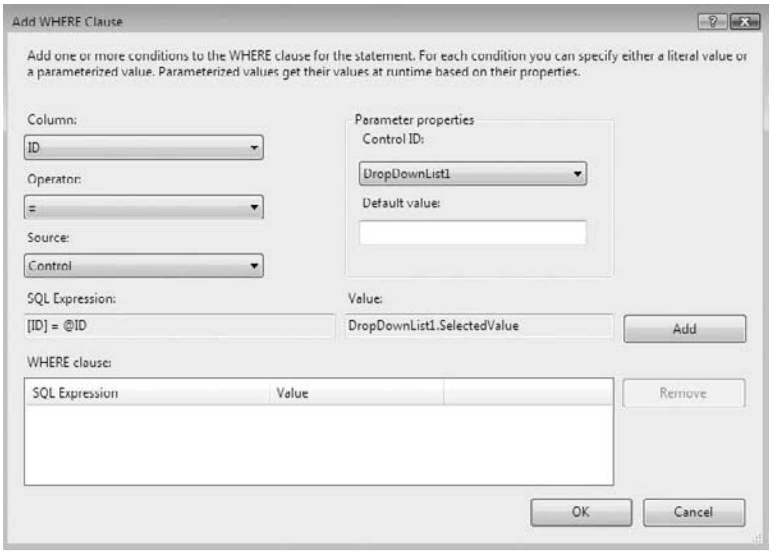
Save your work. You can run the application and test that the DropDownList is populating properly on the page.

Adding the DetailsView Control

The DetailsView control forms the Details part of our master/details construction. It will provide the details of the item selected in the DropDownList and also (in this case) enable the user to edit those details.

With `Computers.aspx` open in Design view, complete the following steps:

1. Drag and drop a DetailsView control from the Data toolbox underneath your DropDown-List in the `Computers.aspx` page.
2. From the rollout Common DetailsView Tasks menu, click Auto Format and choose the Mocha scheme, and click OK.
3. From the Choose Data Source drop-down box in Common DetailsView Tasks, choose New Data Source to open the Data Source Configuration Wizard. Select the Database option and keep the default name of `SqlDataSource2`. Click OK.
4. In the Choose Your Data Connection pane, select the `Computers.mdf` connection from the drop-down box and click the Next button.
5. In the Configure The Select Statement pane, select all the columns of the Catalog table. Click the Advanced button to open the Advanced SQL Generation Options dialog box.
6. Select the Generate Insert, Update, and Delete Statements check box and the Use Optimistic Concurrency option. Click the OK button. This generates the appropriate statements to enable your users to edit the data in the database.
7. Still within the Configure The Select Statement pane of the wizard, click the Where button. This opens the Add Where Clause pane, which enables you to link the content of the DetailsView control to the DropDownList control.
8. From the Column drop-down box, choose ID. Keep the default Operator selection as = and choose Control from the Source drop-down menu. To the right of these menus, under Parameter Properties, set the ControlID to DropDownList1, as shown in Figure 26.19. Click the Add button to finalize the process and then click OK. This returns you to the Configure The Select Statement pane.
9. Click the Next button to open the Test Query window. Click the Test Query button. This opens a dialog box asking for a parameter. Type **1** into the Value field and click OK. If your query has successfully worked, you will see the full entry for Item 1 in the Test Query window.
10. Click the Finish button to return to the design surface.



Add one or more conditions to the WHERE clause for the statement. For each condition you can specify either a literal value or a parameterized value. Parameterized values get their values at runtime based on their properties.

Column: ID

Operator: =

Source: Control

Parameter properties

Control ID: DropDownList1

Default value:

SQL Expression: [ID] = @ID

Value: DropDownList1.SelectedValue

WHERE clause:

SQL Expression	Value

FIGURE 26.19 Setting up the WHERE clause

Further Configuring the DetailsView Control

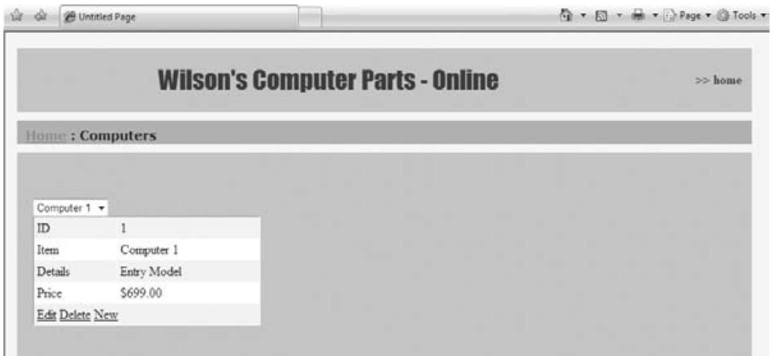
As with the GridView control, the DetailsView control offers a wide range of options to further configure the way it presents data and extend its functionality. In our example, we will perform a few simple further configurations to clean up its appearance and to enable users to edit data displayed in the control. Complete the following:

1. The rollout Common DetailsView Tasks menu for the DetailsView control should now include some additional entries allowing you to enable inserting, editing, and deleting. Select all three check boxes.
2. Click the Edit Fields item to open the Fields dialog box.
3. Click the Price field and scroll the BoundField Properties box down to the `DataFormatString` property. Set the `DataFormatString` property to **${0:c}**. This sets a dollar sign ($) before each entry in this field.
4. Click the OK button to exit the dialog.
5. Close the Common DetailsView Tasks menu and select the resize handles on the Details View control. Stretch the control out to a suitable width on your page.

You will notice that the `SqlDataSource1` and `SqlDataSource2` objects are both located under the DetailsView control. You can use the SqlDataSource controls to edit your data sources if necessary.

Run and test the application. Changing the selection in the DropDownList control should automatically produce a relevant change in the item details displayed in the DetailsView control. In addition, you should be able to use the Edit, Delete, and New links to update the information in your database.

Figure 26.20 demonstrates the running `Computers.aspx` page.



ID	1
Item	Computer 1
Details	Entry Model
Price	$699.00
Edit Delete New	

FIGURE 26.20
The running `Computers.aspx` page

You can obtain the full ASPX source code for `Computers.aspx` from the website for this book.

Building the *Report.aspx* page

Visual Studio 2008 provides two tools for generating online data reports: Crystal Reports and the Microsoft Report Viewer. In this example, you will see how to use the MicrosoftReportViewer control to produce an online report for the data in the `Computers.mdf` database.

Adding the MicrosoftReportViewer Control

The first step is to create a content page linked to `ContentMasterPage.master` that will host the report. Complete the following steps:

1. Begin by choosing Add New Item from the Website menu to open the Add New Item dialog box.
2. Choose the Web Form option and select the Select Master Page check box. Name the page **`Report.aspx`** in the Name text box. Click the Add button. This should open the Select A Master Page dialog box. Choose the master page and click OK.
3. `Report.aspx` should now be open in Design view. Expand the Common Content Tasks rollout menu for the `ContentPlaceHolder1` control and select Create Custom Content.
4. In the Properties box for `Report.aspx`, set the `Title` property of the page to **Wilson's Online Computer Parts – Report**.
5. From the Reporting toolbox, drop an instance of the MicrosoftReportViewer control (`MicrosoftReportViewer1`) into `ContentPlaceHolder1`. This opens the Common Report-Viewer Tasks rollout menu.

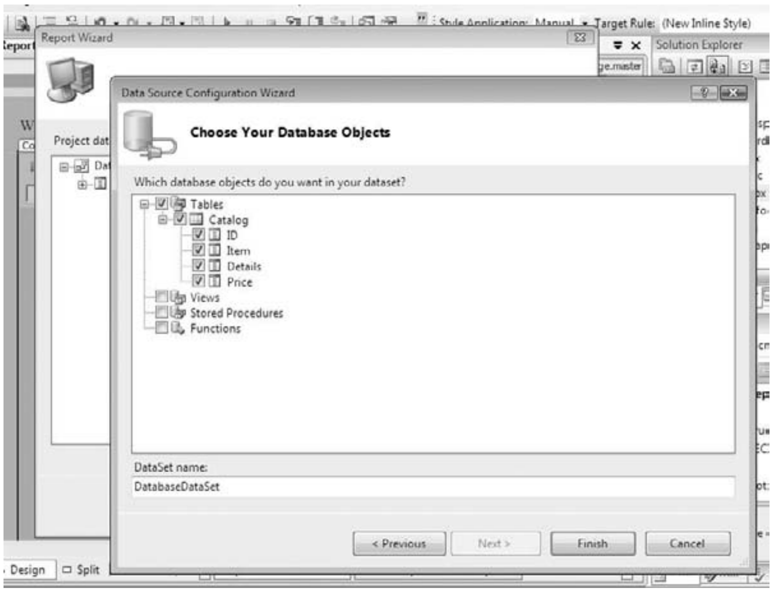
Creating the Report

The next stage is to create the report that will be linked to the MicrosoftReportViewer control. Complete the following steps:

1. From the Common ReportViewer Tasks menu, click the Design A New Report link to open the Report Wizard.

2. In the Report Wizard, click Next to open the Select The Data Source window. Click the Add Data Source button. This opens the Data Source Configuration Wizard.
3. Choose the connection to the `Computers.mdf` database. This may be listed in the drop-down menu as ConnectionString(Web.config), depending on how you named your original connection string. Click the Next button.
4. This opens the Choose Your Database Objects window. It may take a few second to populate, but after the items are listed, open Tables and choose Catalog, as shown in Figure 26.21.

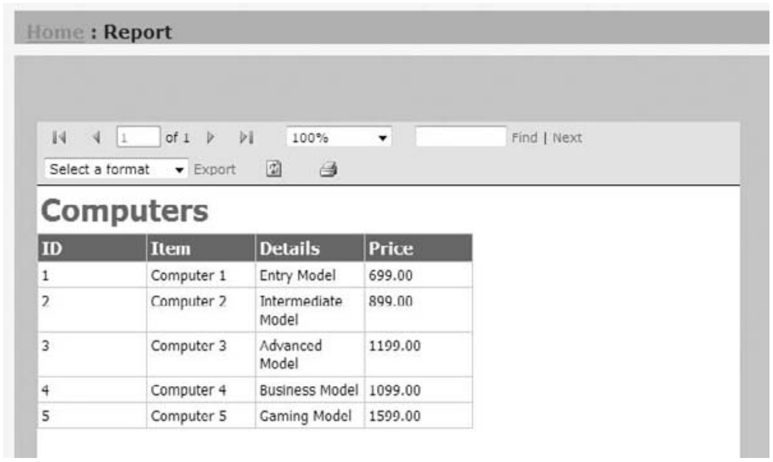
FIGURE 26.21
Using the Choose Your Database Objects window



5. Click the Finish button to return to the Data Source window. The window should now be populated with a DatabaseDataSet object. Open the object, select Catalog, and click the Next button.
6. The Report Wizard will now allow you to start designing your report. In the Select Report Type window, choose the Tabular option and click Next.
7. In the Design The Table window, move the four available fields into the Details window. Click Next.
8. In the Choose The Table Layout window, select Stepped and click Next. In Choose The Table Style, select Slate and click Next.
9. In the Completing The Report Wizard window, set the report name to **Computer Report** and click the Finish button.
10. This opens a new page: `Computer Report.rdlc[Design]`. You may need to select `Computer Report.rdlc` in the Choose Report option in the Common Report Viewer Tasks menu that will be open on this page. You can edit this control directly. For example, select the heading text *report1* and change it to **Computers**.

11. Click back to `Report.aspx` and resize the MicrosoftReportViewer control to stretch it across the page.
12. You will also need to add an entry to the `Report.aspx` page into the SiteMap. Switch to the `Web.sitemap` page and add the following entry: `<siteMapNode url=''report.aspx'' title=''Report'' description=''Report'' />`.
13. Run the application. Figure 26.22 illustrates how the running report should appear.

FIGURE 26.22
`Report.aspx` at runtime



Computers

ID	Item	Details	Price
1	Computer 1	Entry Model	699.00
2	Computer 2	Intermediate Model	899.00
3	Computer 3	Advanced Model	1199.00
4	Computer 4	Business Model	1099.00
5	Computer 5	Gaming Model	1599.00

You can obtain the full ASPX source code for `Report.aspx` from the website for this book.

The Bottom Line

Create cascading style sheets. Even though you may have created a main content area for your page, you will still need to lay out the various text items and images that compose the area. You can employ CSS to create content containers that can be used for layout and styles within the main style sheet areas defined for a page.

Master It Develop a text container using CSS that will occupy the full width of the containing area and expand vertically to accommodate the content. Include style attributes for the `<p>` tag that set the font to Verdana, the text color to dark blue, and the font size to 8 point. Set vertical alignment to top, and horizontal alignment to left.

Use web form controls. Images are an important part of any web page. The Image control from the Standard toolbox gives you a control that combines the ability to display an image with server-side functionality.

Master It Place an Image control onto an ASPX page and use it to display an image.

Create a web user control. Web user controls enable you to create reusable combinations of controls and functionality.

Master It Create a web user control with two TextBox controls and a Label control that calculates a percentage based on amounts entered into the TextBox controls.

Chapter 27

ASP.NET Web Services

An ASP.NET web service is a program capable of communicating across a network such as the Internet by using a combination of the open standard SOAP and XML technologies. (Note that SOAP was previously known as the Simple Object Access Protocol.)

Web services are ideal for creating data-, content-, or processing-related services that can be made available to associated or third-party applications and clients across distributed networks such as the Internet.

In this chapter, you will see how to create a simple ASP.NET web service and a client application to *consume*, or use, the web service.

In addition, this chapter covers the technologies associated with ASP.NET web services, such as SOAP and the Web Services Description Language (WSDL). The chapter briefly covers Microsoft's latest addition to the web service stable — the Windows Communication Foundation (WCF) — and you will see how to use Asynchronous JavaScript and XML (AJAX) technology to create seamless interactions between web services and their consuming applications.

In this chapter, you'll learn how to do the following:

- ◆ Create a simple ASP.NET web service
- ◆ Consume an ASP.NET web service
- ◆ Work with AJAX technologies

Using ASP.NET Web Services and WCF

Microsoft offers two flavors of web service technology:

- ◆ ASP.NET web services
- ◆ Windows Communication Foundation (WCF)

ASP.NET web services (also known as *XML web services*) have been around through all the incarnations of ASP.NET and offer a simple and effective methodology for making software components and other resources available over the Internet.

WCF is a recent inclusion into the .NET Framework and is built around the web services architecture. WCF enables broader integration and interoperability with all the .NET Framework distributed system technologies, including Microsoft Message Queuing (MSMQ), Common Object Model plus (COM+), ASP.NET web services, and .NET Framework Remoting. WCF also offers improved performance and secure data transmission.

ASP.NET Web Services

ASP.NET web services are software resources and components that are able to expose their functionality and/or deliver data over a network such as the Internet by using a combination of XML and SOAP. You can restrict a component to be available to only certain applications or specific users, or you can make it available to many users. The component can be limited to the local computer or the local network, or made available across the Internet. Services can be delivered free of charge or on a fee-paying basis.

Virtually any program that can be encapsulated as a component can be expressed as an ASP .NET web service. For production purposes, you will need a web server to deliver your ASP.NET web service. However, for development purposes, the built-in ASP.NET Development Server that ships with Visual Studio 2008 is quite sufficient.

The process of using an ASP.NET web service or incorporating a web service into your client is called *consuming* an ASP.NET web service.

The advantages of using ASP.NET web services include the following:

- Data and commands are communicated across the standard Internet port: port 80. This greatly simplifies passage around the Internet and most networks.
- The common standards of XML and SOAP are widely supported.
- Early problems with web services, such as lack of a robust security model, have been resolved.
- Visual Studio provides a simple and straightforward environment in which to create and consume web services.

Further information on ASP.NET web services can be found at `http://msdn2.microsoft.com/en-us/webservices/default.aspx`.

Windows Communication Foundation (WCF)

WCF is built on ASP.NET web services and extends their functionality by integrating with a number of distributed .NET Framework technologies.

WCF offers an integrated approach to situations in which you would previously have employed a range of different distributed .NET Framework technologies. Typically, you use WCF as a unified solution enabling you to avoid having to employ different distributed technologies for each requirement of a distributed application. For example, you may have employed message queuing for use with portable devices that are not permanently connected, ASP.NET web services for communication across the Internet, and .NET Framework Remoting for tightly coupled communication within the local network. Employing multiple technologies in this manner results in a complex and potentially unwieldy solution. WCF offers a method of achieving a simpler unified approach.

However, despite its advantages in simplifying complex situations, WCF remains a complex technology that requires a significant amount of work even to produce a simple Hello World–style application. Because of the complexity of WCF, this chapter focuses primarily on building and working with ASP.NET web services.

Further information on WCF can be found at `http://msdn2.microsoft.com/en-us/netframework/aa663324.aspx` and `www.microsoft.com/net/wcf.aspx`.

Understanding Technologies Associated with Web Services

Several technologies underlie and support ASP.NET web services. They include SOAP, WSDL, SOAP Discovery, and Universal Description, Discovery, and Integration (UDDI).

SOAP

SOAP was originally known as the Simple Object Access Protocol. This was changed by the World Wide Web Consortium (W3C) with version 1.2 of the standard in 2003 because the original acronym was believed to be misleading.

SOAP is a lightweight protocol for exchanging XML messages over Hypertext Transfer Protocol/Secure Hypertext Transfer Protocol (HTTP/HTTPS). It forms the basis of the web services stack, which is the set of protocols used to define, locate, and deliver web services.

SOAP is an open standard, enabling web services to be developed and supported across a range of platforms and environments.

There are other services attached to SOAP, including WSDL and SOAP Discovery.

Although you are no longer required to work directly with SOAP when developing ASP.NET web services in Visual Studio, you will still continue to encounter references to the protocol because it underlies the whole web service creation, delivery, and consumption process.

A SOAP tutorial can be found at `www.w3schools.com/soap/`.

Web Services Description Language (WSDL)

WSDL is the language used to create an XML document that describes a web service. Specifically, the document describes the location of the service and the methods exposed by the service.

You can create and edit WSDL documents directly by using a text editor, but they can usually be generated automatically by Visual Studio when you add either a web reference or service reference to your ASP.NET web service. For further information, see the ''Adding a Web Reference'' section later in this chapter.

A WSDL tutorial can be found at `www.w3schools.com/wsdl/default.asp`.

SOAP Discovery

SOAP Discovery is used to locate the WSDL documents that provide the descriptions for ASP.NET web services. You use SOAP Discovery when you want to make your web service publicly available for consumption by third-party applications. For example, you may be providing a weather service for third-party providers to incorporate into their websites. There are two types of discovery: static discovery and dynamic discovery.

In the case of static discovery, an XML document with the `.DISCO` file extension is used. This file contains information about the location of the WSDL documents.

If you wish to enable dynamic discovery for your website, you add a specific reference into the `Web.config` file. Dynamic discovery enables users to discover all web services and discovery files beneath the requested URL.

Discovery files (and particularly dynamic discovery) can be a security risk on a production server because they potentially allow users to search the entire directory tree. Static discovery files are the safer of the two types because they allow the user to search only those resources that you choose to nominate. In Visual Studio 2008, you can explicitly generate a static discovery file by adding a web reference or a service reference. (See ''Adding a Web Reference'' later in this chapter for more information on creating discovery files and enabling dynamic discovery.)

Universal Description, Discovery, and Integration (UDDI)

UDDI was originally created as part of the web service specification to act as a form of yellow pages for web services. Several major players in developing the web services specification (including Microsoft, IBM, SAP, and OASIS) combined to develop an XML-based registry for businesses to promote themselves and their web services to both the public and the corporate world. In 2006, Microsoft, IBM, and SAP closed their public UDDI nodes. However, you can still create UDDI servers on your local network to provide directory services for web services available within your network. More information on configuring Microsoft server technology for UDDI can be found at `www.microsoft.com/windowsserver2003/technologies/idm/uddi/default.mspx`. More information on UDDI can be found at `http://uddi.xml.org`.

Creating a Simple ASP.NET Web Service

Creating and consuming web services in Visual Studio 2008 is a relatively simple and straightforward process. In this example, you will create a simple Hello World–style ASP.NET web service within a website entitled HelloWebServiceDemo. You will then see how to consume the web service from within the same website.

Opening Visual Studio in Administrator Mode

Visual Studio often requires elevated privileges when creating and accessing applications and resources. If you are logged in as a standard user, you may not have those privileges available.

To increase your privileges, from the Start menu, right-click the Visual Studio 2008 entry. From the context menu, choose Run As Administrator. You may be required to enter credentials.

Setting Up the Web Service

This simple web service will have one service, HelloWorld, with a single method, `Hello`. To set up the web service, complete the following steps:

1. Launch Visual Studio and choose File ➤ New Web Site.
2. From the New Web Site dialog box, choose ASP.NET Web Site. In the location text box, name the website **HelloWebServiceDemo**. Click OK.
3. Choose File ➤ Add New Item and select the Web Service template. In the Name text box, delete the default `WebService.asmx` and rename the web service **`HelloWorld.asmx`**. Click the Add button. This opens the `App_Code/HelloWorld.vb` page, where default code for a Hello World–style web service is already set up.
4. You will make one minor change to the default code. In the `<WebMethod()>` section of the code, change the function name from `HelloWorld()` to **`Hello()`**. This enables you to distinguish between the service name and the method. The code should now read as shown in the following snippet:

```
<WebMethod()> _
   Public Function Hello() As String
     Return "Hello World"
   End Function
```

5. Save your work.

Next, you will run and test the web service.

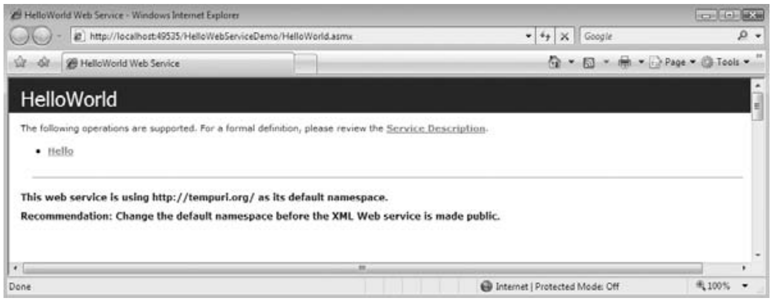
Testing the Web Service

After you have created your web service, it is a good idea to test the service to ensure that it behaves as expected. This presentation of the testing process lacks the polish that you might wish for your web service after it is utilized or consumed by a client application, but it will demonstrate the service's inherent functionality. The product of the test is returned as straight XML. But don't worry — when you finally consume the service, the XML markup will be stripped away from the returned data. Follow these steps:

1. In the Solution Explorer window, right-click `HelloWorld.asmx` and choose the Set As Start Page option.
2. Click the green arrow in the Standard toolbar (or press F5) to start the web service in debugging mode. Click OK in the Debugging Not Enabled dialog box to automatically modify the `Web.config` file to enable debugging.

The ASP.NET web service should now open in your web browser as shown in Figure 27.1.

FIGURE 27.1
HelloWorld web service in Internet Explorer


HelloWorld Web Service - Windows Internet Explorer
http://localhost:49535/HelloWebServiceDemo/HelloWorld.asmx
Google
HelloWorld Web Service
Page
Tools
Done
Internet | Protected Mode: Off
100%

HelloWorld

The following operations are supported. For a formal definition, please review the Service Description.

- Hello

This web service is using http://tempuri.org/ as its default namespace.

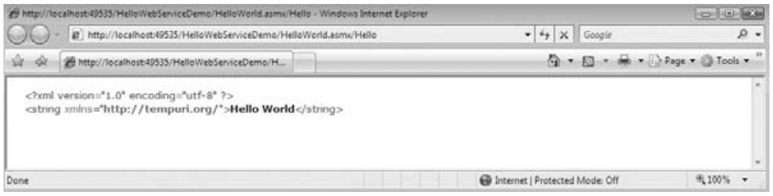
Recommendation: Change the default namespace before the XML Web service is made public.

You can check the service description for HelloWorld by clicking the Service Description link. This opens a WSDL description for the web service.

You will also see a warning about using the default namespace of `http://tempuri.org`. This is the default Microsoft namespace, and you would usually replace it with a reference to a URL that you control before publicly deploying the web service. You will see how to do this later in this chapter, in the ''Building MyWebService'' section.

To call the `Hello` method, click the Hello link. This opens a new page, which displays information concerning the `Hello` method. To run the `Hello` method, click the Invoke button. This opens the full XML page returned by the method, as shown in Figure 27.2.

FIGURE 27.2
Invoking the `Hello` method


http://localhost:49535/HelloWebServiceDemo/HelloWorld.asmx/Hello - Windows Internet Explorer
http://localhost:49535/HelloWebServiceDemo/HelloWorld.asmx/Hello
Google
http://localhost:49535/HelloWebServiceDemo/H...
Page
Tools
Done
Internet | Protected Mode: Off
100%

```
<?xml version="1.0" encoding="utf-8" ?>
<string xmlns="http://tempuri.org/">Hello World</string>
```

Consuming the Web Service

The next step is to consume the HelloWorld web service from within a standard ASPX page. Close any running instances of the HelloWorld web service to stop debugging and return to the HelloWebServiceDemo website in Visual Studio. Complete the following steps:

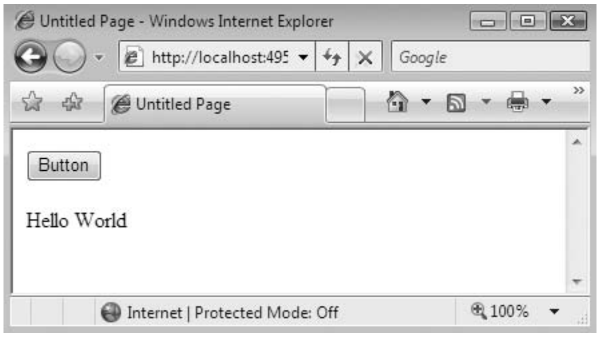
1. In Solution Explorer, double-click `Default.aspx` to open the page in Design view.
2. From the Standard toolbox, drag and drop a Button control into the default Div control on the form.
3. Click the Enter button twice to introduce two line breaks, and add a Label control from the Standard toolbox.
4. In the Properties window for the Label control, delete the default Label text from the `Text` property.
5. Double-click the Button control to open the code skeleton for the `Button1_Click` event in code-behind.
6. Complete the `Button1_Click` event with the following code:

```
Protected Sub Button1_Click(ByVal sender As Object, _
 ByVal e As System.EventArgs) Handles Button1.Click
     Dim myHello As New HelloWorld
     Label1.Text = myHello.Hello()
End Sub
```

   In this example, we declare a local instance of the HelloWorld service and tie the `Text` property of the Label control to the `Hello` method.

7. In Solution Explorer, right-click `Default.aspx` and choose the Set As Start Page option.
8. Run the application. `Default.aspx` should render initially as a page displaying a single button. Clicking the button should display the Hello World text. Figure 27.3 shows the running application.

FIGURE 27.3
The running HelloWeb-ServiceDemo application


Untitled Page - Windows Internet Explorer
http://localhost:495
Google
Untitled Page
Button
Hello World
Internet | Protected Mode: Off
100%

Developing a Stand-Alone Web Service

Web services are designed to run separately from their consuming applications. In this example, you will see how to build a slightly less trivial example of an ASP.NET web service as a stand-alone application. You will then see how to consume the web service from a separate web application.

The example involves building a web service that performs two functions. It returns the current server time and also provides a tool for calculating a percentage. The web service is named MyWebService, and the two methods are named `ServerTime` and `CalculatePercentage`.

Later in this chapter, you will see how to create a simple AJAX implementation that enables the client to automatically and asynchronously update the displayed server time from the web service.

Building MyWebService

You will begin by creating the web service. Unlike the previous example, in which the web service and consuming application were built within the same project, this web service is a stand-alone project. Follow these steps:

1. Open Visual Studio 2008 and choose File ➢ New Web Site. From the New Web Site dialog box, choose ASP.NET Web Service.
2. In the Location text box of the New Web Service dialog box, keep the default path but change the name of the web service to **MyWebService**. Click the OK button to exit the dialog box.
3. The web service should now be opened to the `App_Code/Service.vb` page in the Visual Studio designer. Look through the default code and change the Namespace entry from `http://tempura.org/` to either a URL that you control or, for the purposes of this example, **`http://mywebservice.org`**. This will prevent the warning message about using the default Microsoft namespace from appearing when you run the web service. The line of code should now read as follows:

```
<WebService(Namespace:="http://mywebservice.org/")> _
```

4. Move down to the `<WebMethod()>` section of the code skeleton. Delete the following default `HelloWorld()` public function:

```
Public Function HelloWorld() As String
   Return "Hello World"
End Function
```

5. Add the following code to the `<WebMethod()>` section. This method will return the current server time as the time of day in hours, minutes, and seconds:

```
<WebMethod()> _
Public Function ServerTime() As String
   ServerTime = Left(Now.TimeOfDay().ToString(), 8)
End Function
```

6. Now you'll create the percentage calculator method (`CalculatePercentage`). Underneath the `ServerTime` method, add the following code:

```
<WebMethod()> _
Public Function CalculatePercentage(ByVal myTotal _
 As Integer, ByVal myValue As Integer) As Integer
      CalculatePercentage = CInt(myValue * 100 / myTotal)
End Function
```

This method calculates a percentage based on the `myValue` and `myTotal` parameters. The calculated percentage is returned as an integer.

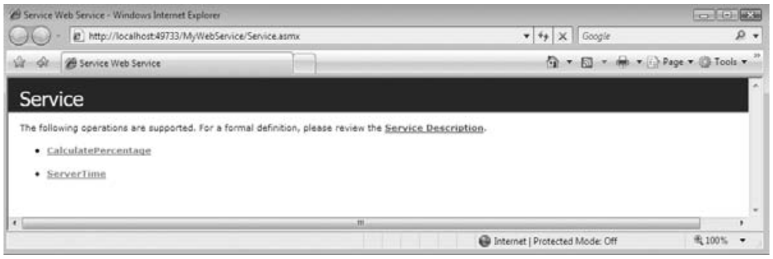
This completes the code for the MyWebService web service. Listing 27.1 gives the full code for the web service as it should appear in `App_Code/Service.vb`.

Listing 27.1: Full Code Listing for MyWebService

```
Imports System.Web
Imports System.Web.Services
Imports System.Web.Services.Protocols

` To allow this Web Service to be called from script, using ASP.NET AJAX _
 uncomment the following line.
` <System.Web.Script.Services.ScriptService()> _
<WebService(Namespace:="http://mywebservice.org/")> _
<WebServiceBinding(ConformsTo:=WsiProfiles.BasicProfile1_1)> _
<Global.Microsoft.VisualBasic.CompilerServices.DesignerGenerated()> _
Public Class Service
    Inherits System.Web.Services.WebService

    <WebMethod()> _
    Public Function ServerTime() As String
        ServerTime = Left(Now.TimeOfDay().ToString(), 8)
    End Function

    <WebMethod()> _
     Public Function CalculatePercentage(ByVal myTotal As Integer, ByVal _
 myValue As Integer) As Integer
        CalculatePercentage = CInt(myValue * 100 / myTotal)
    End Function

End Class
```

To make this the default start page, right-click `Service.asmx` in Solution Explorer and choose Set As Start Page from the context menu.

Test the web service by clicking the green arrow on the Standard toolbar or by pressing F5. The web service should display links to the two web methods, as shown in Figure 27.4. Test the two web methods by clicking the links and then clicking the Invoke button on each of the respective service information pages. `ServerTime` should return the current time of day in 24-hour format (as an XML page). The service information page for `CalculatePercentage` should provide you with input boxes to enter values for `MyValue` and `MyTotal` before you invoke the service. Entering values such as 20 for `MyValue` and 50 for `MyTotal` should return a value of 40 (within an XML page) when you click Invoke.


Service Web Service - Windows Internet Explorer
http://localhost:49733/MyWebService/Service.asmx
Google
Service Web Service
Page
Tools
Internet | Protected Mode: Off
100%

Service

The following operations are supported. For a formal definition, please review the Service Description.

- CalculatePercentage
- ServerTime

FIGURE 27.4
The running MyWebService

Deploying MyWebService

In a production environment, you typically deploy the web service to Microsoft's Internet Information Services (IIS) web server. In order for the web service to run on IIS, you need to have .NET Framework 3.5 registered with IIS. You can also set up appropriate security and access privileges such as Windows authentication, secure socket encryption (HTTPS), and so forth on IIS. For further information on setting up and working with IIS, *Microsoft IIS 7 Implementation and Administration* by John Paul Mueller (Sybex, 2007) gives thorough coverage. You can also refer to Microsoft's community portal for IIS at `www.iis.net`.

Before you can deploy your web service, you must have all the relevant files and directories assembled into a suitable directory without the various development and debugging files. The easiest way to create a folder containing all the production files necessary for a deployment of your web service is to use the Publish Web Site option from the Build menu. This opens the Publish Web Site dialog box, where you can choose to publish to a specific location (including an FTP location) or keep the default location and move the folder later.

USING ASP.NET DEVELOPMENT SERVER

Visual Studio 2008 comes equipped with its own built-in web server for testing web applications: the ASP.NET Development Server. Although the ASP.NET Development Server is ideal for testing web applications as you develop them, it does have its limitations. One of those limitations is that you need to have a separate instance of the web server running for each web application running concurrently on your development machine.

Thus, to consume MyWebService from another application, you need to have opened MyWebService in Visual Studio 2008 and run the application to fire up the ASP.NET Development Server. You can close the web browser running the web service, but you must keep Visual Studio open to MyWebService. To create a separate project to consume MyWebService, you must open another instance of Visual Studio 2008 from the Start menu.

An advantage of using the ASP.NET Development Server is that you do not need to publish or physically deploy your application to a web server before you can test it.

Consuming MyWebService

As discussed in the previous section, unless you are using IIS to test your web applications, you must have MyWebService open in Visual Studio 2008 and have run the web service at least once to open an instance of the ASP.NET Development Server so that you can link to the web service from another application.

Keep this instance of Visual Studio open and use the Start menu to open another instance of Visual Studio 2008. Depending on the account restrictions on your machine, it may be necessary to open the second instance of Visual Studio as Administrator so you can connect to the web service.

To open Visual Studio 2008 in Administrator mode, right-click the Visual Studio entry in the Start menu and choose Run As Administrator from the context menu.

After Visual Studio 2008 opens, complete the following steps:

1. Choose File ➢ New Web Site. In the New Web Site dialog box, choose ASP.NET Web Site. Name the site **MyConsumer** in the Location text box (keeping the rest of the default path). Click the OK button.
2. MyConsumer should open to `Default.aspx` in Design mode. Drag a TextBox from the Standard toolbox into the default Div control on the form. In the Properties window, set the `ID` property for the TextBox to **tbMyValue**.
3. From the Standard toolbox, drop a Label control on the form to the right of tbMyValue. Set the `Text` property of the Label control to **My Value**. Click to the right of the Label and press the Enter key to drop to the next line.
4. Drop a second TextBox control onto the form immediately under tbMyValue. Set the `ID` property for the second TextBox control to **tbMyTotal**.
5. Place a Label control immediately to the right of tbMyTotal. Set the `Text` property of the Label control to **My Total**. Click to the right of the Label control and press the Enter key twice to drop down two lines.
6. From the Standard toolbox, drop a Button control onto the form two lines below tbMy-Total. In the Properties window for the Button, set the `ID` property to **btnCalculate**. Set the `Text` property to **Calculate**. Press the Enter key to drop down one more line.
7. Immediately beneath btnCalculate, place a Label control and set its `ID` property to **lbl-Percentage**. Delete the default contents of the `Text` property for the control.
8. Place another Label control to the right of lblPercentage. Set the `ID` property to **lbl-PercentageLabel**. Set the `Text` property to **= Calculated Percentage** and set the `Visible` property to False. Click to the right of this control and use the Enter key to drop down two more lines.
9. Place a Label control two lines beneath lblPercentage. Set the `ID` property to **lblServerTime** and delete the default `Text` property entry.
10. Place a final Label control to the right of lblServerTime. Set the `Text` property to **= Server Time**.

Figure 27.5 illustrates how the final page should look with all the controls in place.

Adding a Web Reference

We use a web reference created in the Solution Explorer to provide the connection to the web service. For web services within the local domain, you can also use the Service Reference option available in Solution Explorer. Using a service reference is an advantage when you want to fully exploit the AJAX potential in connecting to a web service, because a service reference allows you to call the web service methods by using JavaScript functions from within your consuming application. Using client script in this manner to call web service methods is entirely asynchronous and prevents your page or portions of your page from being locked out from user interaction while waiting for the web service to respond.

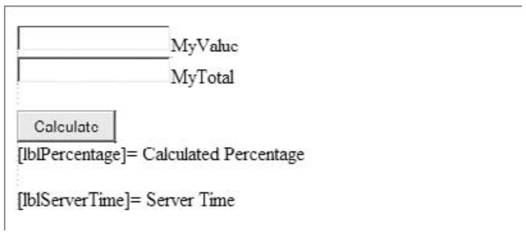

MyValue
MyTotal
Calculate
[lblPercentage]= Calculated Percentage
[lblServerTime]= Server Time

FIGURE 27.5
Layout for `Default.aspx`

For this example, you will add a web reference to the MyWebService web service created in the previous section. Unless you are using IIS to test your web applications, you must have an instance of the ASP.NET Development Server running the MyWebService web service. Refer to the beginning of this section for details.

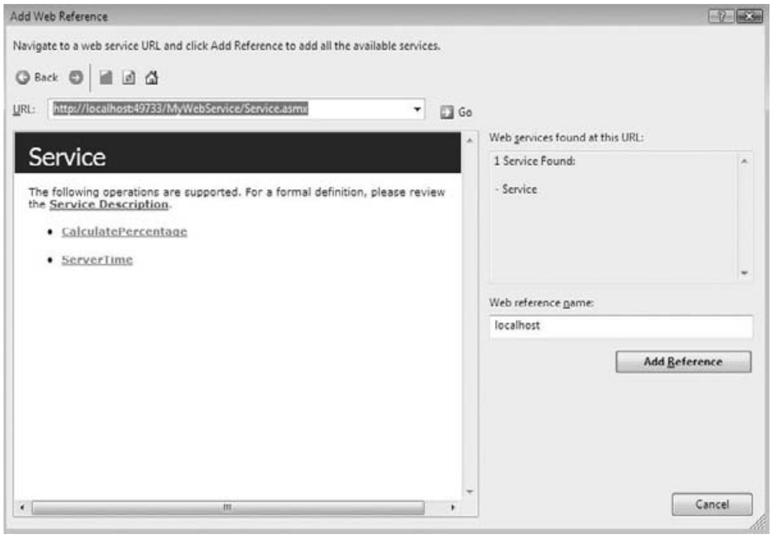
Complete the following steps:

1. In Solution Explorer, right-click the project heading (and path) at the top of the Solution Explorer tree. From the context menu, choose Add Web Reference to open the Add Web Reference dialog box.
2. If you are using IIS and have appropriately placed discovery documents, you could use the Browse To: Web Services On The Local Machine option. You will also see a link to enable you to refer to any UDDI servers set up on your network. However, because we are using the ASP.NET Development Server, you will need to switch to the instance of Visual Studio running MyWebService and open the web service in Internet Explorer. Copy the URL for the MyWebService web service from the address bar of Internet Explorer.
3. Switch back to your MyConsumer build and paste the URL of MyWebService into the URL text box of the Add Web Reference dialog box. The URL should be something like this: `http://localhost:49733/MyWebService/Service.asmx`.
4. Click the Go button. This should now establish a connection to MyWebService, as shown in Figure 27.6. Click the Add Reference button to exit the dialog box. Solution Explorer should now feature an App_WebReferences folder with appropriate entries for MyWebService. Included in these are a discovery (`DISCO`) and a WSDL file that you can copy, edit, and employ in your deployment of MyWebService.

The `DISCO` document created by adding a web reference can be used to enable static discovery of your web service by placing it in a suitable location in your folder hierarchy for the site. If you examine the code in the automatically generated file, you can see how to add and remove XML entries. You can then add a link to the page from some point in your site. If you do not wish to set up static discovery files, you can enable dynamic discovery by editing the `Machine.config` file for your web server. Remember that dynamic discovery potentially allows users to browse your directory tree; unless your web server is suitably protected, dynamic discovery is not recommended for production servers.

For precise details on enabling dynamic discovery, please refer to the relevant Help section of Visual Studio 2008. Type **dynamic discovery** in the Search field.

FIGURE 27.6
The Add Web Reference dialog box


Add Web Reference
Navigate to a web service URL and click Add Reference to add all the available services.
Back
URL: http://localhost:49733/MyWebService/Service.asmx
Go
Service
The following operations are supported. For a formal definition, please review the Service Description.
CalculatePercentage
ServerTime
Web services found at this URL:
1 Service Found:
- Service
Web reference name:
localhost
Add Reference
Cancel

ADDING THE CODE-BEHIND

The next step is to add the code to make the application work. From Design mode, double-click the `btnCalculate` control to enter code-behind and complete the following steps:

1. Begin by declaring a local instance of the web service. At the top of the page, directly under the `Inherits System.Web.UI.Page` entry, add the following:

```
Dim MyWebService As New localhost.Service
```

2. Next is the code to call the `ServerTime` method. In the code skeleton for `Form1_Load`, add the following line of code:

```
lblServerTime.Text = MyWebService.ServerTime
```

3. Next is the code to collect the input values from the user and call the `CalculatePercent-age` method. The code also attaches a percentage sign onto the displayed percentage value and unhides lblPercentageLabel. In the code skeleton for the `btnCalculate Click` event, add the following code snippet:

```
Dim myValue As Integer = CInt(tbMyValue.Text)
Dim myTotal As Integer = CInt(tbMyTotal.Text)
lblPercentageLabel.Visible = True
lblPercentage.Text = MyWebService.CalculatePercentage(myTotal, myValue)_
& "%"
```

The final code should appear as shown in Listing 27.2.

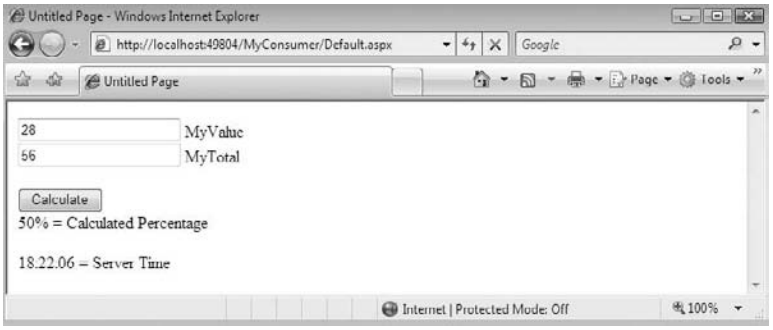
LISTING 27.2: Full Code Listing for *Default.aspx.vb*

```
Partial Class _Default
    Inherits System.Web.UI.Page
    Dim MyWebService As New localhost.Service

    Protected Sub form1_Load(ByVal sender As Object, ByVal e As _
 System.EventArgs) Handles form1.Load
        lblServerTime.Text = MyWebService.ServerTime
    End Sub

    Protected Sub btnCalculate_Click(ByVal sender As Object,_
 ByVal e As System.EventArgs) Handles btnCalculate.Click
        Dim myValue As Integer = CInt(tbMyValue.Text)
        Dim myTotal As Integer = CInt(tbMyTotal.Text)

        lblPercentageLabel.Visible = True
        lblPercentage.Text = MyWebService.CalculatePercentage _
(myTotal, myValue) & "%"
    End Sub
End Class
```

Setting up this part of the project is now complete. In Solution Explorer, right-click `Default.aspx` and choose Set As Start Page. Run the application and test the methods.

Figure 27.7 illustrates the running application after 28 has been entered as My Value and 56 has been entered as My Total.

FIGURE 27.7 The running MyConsumer application


Untitled Page - Windows Internet Explorer
http://localhost:49804/MyConsumer/Default.aspx
Google
Untitled Page
Page
Tools
28
MyValue
56
MyTotal
Calculate
50% = Calculated Percentage
18.22.06 = Server Time
Internet | Protected Mode: Off
100%

Simple AJAX Implementation

ASP.NET 3.0 and 3.5 integrate AJAX to enable developers to easily perform partial updates of web pages accessing ASP.NET web services. These updates not only do not require a full refresh of the page, but also can be performed asynchronously so as not to interfere with other operations on the page.

In this example, we will use a simple combination of the AJAX controls to enable the `ServerTime` method from MyWebService to be continuously updated on a page in MyConsumer. A more sophisticated implementation enables the developer to access the methods in the web service from client script (JavaScript). This latter implementation is fully asynchronous, whereas our example will have some limitations that are explained later in this section.

Open MyWebService in Visual Studio and run the application to open an instance of the ASP.NET Development Server. Next, open a separate instance of Visual Studio 2008 from the Start menu and open the MyConsumer website. Complete the following steps:

1. From the Website menu, click Add New Item. From the Add New Item dialog box, select AJAX Web Form and rename it **`myAjax.aspx`**. Click the Add button.
2. `myAjax.aspx` should now be open in Design mode. You will see that it has a default ScriptManager control on the page. Do not delete this control because it is necessary for the AJAX functionality to work. From the AJAX Extensions toolbox, drop an UpdatePanel control onto your page underneath the ScriptManager control. The UpdatePanel control acts as an area that can be partially refreshed without involving an entire page refresh. Keep the default `ID` property of UpdatePanel1.
3. From the AJAX Extensions toolbox, drop a Timer control into UpdatePanel1. In the Properties box, set the `Interval` property to 1000 (1 second). By placing the Timer control inside the UpdatePanel control, UpdatePanel1 automatically responds to `Tick` events from the Timer. You can also set the UpdatePanel control to respond to events from external controls by using the UpdatePanel's `Triggers` property.
4. From the Standard toolbox, drop a Label control into UpdatePanel1. Set the `ID` property to **lblServerTime** and delete the default entry in the `Text` property.
5. Double-click Timer1 to enter code-behind for the application. This should open `myAjax.aspx.vb`.
6. MyConsumer already has a web reference for MyWebService, so at the top of the page, directly under the `Inherits System.Web.UI.Page` entry, add the following:

   ```
   Dim MyWebService As New localhost.Service
   ```

7. In the code skeleton for the `Timer1_Tick` event, add the following line of code:

   ```
   lblServerTime.Text = MyWebService.ServerTime
   ```

This part of the application is now complete. Listing 27.3 gives the full code listing for `myAjax.aspx.vb`.

LISTING 27.3: Full Code Listing for *myAjax.aspx.vb*

```
Partial Class myAjax
    Inherits System.Web.UI.Page
    Dim MyWebService As New localhost.Service

    Protected Sub Timer1_Tick(ByVal sender As Object, ByVal e As _
 System.EventArgs) Handles Timer1.Tick
```

```
        lblServerTime.Text = MyWebService.ServerTime
    End Sub
End Class
```

Right-click the entry for `myAjax.aspx` in Solution Explorer and choose Set As Start Page from the context menu. Click the green arrow or press F5 to run the application. The running page should display the current server time, which is automatically updated every second.

You can now add further controls and functionality to the page separate from the UpdatePanel control. These controls will not be affected by the partial page refreshes performed by the UpdatePanel control. The main limitation of this approach is that any other control placed inside the UpdatePanel (or any other UpdatePanel control on the page) will be locked out while waiting for MyWebService to complete its business (every second!).

You can test this behavior by adding a second UpdatePanel control to the page and dropping a TextBox control into it. Drop a second TextBox control onto the page, but not inside the UpdatePanel. Run the application and try typing into the TextBoxes. It is difficult to type text into the TextBox inside the UpdatePanel.

If we had used a scripted approach, we could have achieved a fully asynchronous operation. For more information on this topic, refer to ''Using the UpdatePanel Control with a Web Service'' in the Visual Studio 2008 Help documentation.

The Bottom Line

Create a simple ASP.NET web service. Creating ASP.NET web services is straightforward with Visual Studio. ASP.NET web services provide a great method for delivering data and functionality within a distributed environment, including the Internet.

Master It Develop an ASP.NET web service that enables the user to add two numbers.

Consume an ASP.NET web service. Adding a web reference or service reference to a web service is a key element to creating an application that can consume the web service.

Master It Create a new website and add a service reference to a web service on your machine.

Work with AJAX technologies. UpdatePanel controls are used in AJAX implementations to provide partial page refreshes. A control placed within the UpdatePanel will automatically refresh the UpdatePanel with a postback. However, you can also use a control located elsewhere on the page to trigger an UpdatePanel refresh.

Master It Add a Button control to an AJAX web page that is set to trigger an asynchronous update for an UpdatePanel control.

Appendix A

The Bottom Line

Each of the ''Bottom Line'' sections in the chapters suggests exercises to deepen skills and understanding. Sometimes there is only one possible solution, but often you are encouraged to use your skills and creativity to create something that builds on what you know and lets you explore one of many possible solutions.

Chapter 1: Getting Started with Visual Basic 2008

Navigate the integrated development environment of Visual Studio. To simplify the process of application development, Visual Studio provides an environment that's common to all languages, known as an integrated development environment (IDE). The purpose of the IDE is to enable the developer to do as much as possible with visual tools, before writing code. The IDE provides tools for designing, executing, and debugging your applications. It's your second desktop, and you'll be spending most of your productive hours in this environment.

Master It Describe the basic components of the Visual Studio IDE.

Solution The basic components of the Visual Studio IDE are the Form Designer, where you design the form by dropping and arranging controls on it, and the code editor, where you write the code of the application. The controls you can place on your form to design the application's interface are shown in the Toolbox window, and the properties of the selected control are shown in the Properties window.

Understand the basics of a Windows application. A Windows application consists of a *visual interface* and *code*. The visual interface is what users see at runtime: a form with controls with which the user can interact — by entering strings, checking or clearing check boxes, clicking buttons, and so on. The visual interface of the application is designed with visual tools. The visual elements incorporate a lot of functionality, but you need to write some code to react to user actions.

Master It Describe the process of building a simple Windows application.

Solution First you must design the form of the application by dropping controls from the Toolbox window onto the form. Size and align the controls on the form, and then set their properties through the Properties window. The controls include quite a bit of functionality right out of the box. A TextBox control with its `MultiLine` property set to True and its `ScrollBars` property set to Vertical is a complete, self-contained text editor.

After the visual interface has been designed, you can start coding the application. Windows applications follow an event-driven model: We code the events to which we want our application to react. For example, the `Click` events of the various buttons are typical events to which an application reacts. Then, there are events that are usually ignored by developers.

The TextBox control fires some 60 events, but most applications don't react to a single one of them. You select the actions to which you want your application to react and program these events accordingly. When an event is fired, the appropriate event handler is automatically invoked. Event handlers are subroutines that pass two arguments to the application: the `sender` object (which represents the control that fired the event) and the `e` argument (which carries additional information about the event). To program a specific event for a control, double-click the control on the design surface, and the editor will come up with the default event for the control. You can select any other event to program in the Events combo box at the top of the editor's window.

Chapter 2: Variables and Data Types

Declare and use variables. Programs use variables to store information during their execution, and different types of information are stored in variables of different types. Dates, for example, are stored in variables of the Date type, while text is stored in variables of the String type. The various data types expose a lot of functionality that's specific to a data type; the methods provided by each data type are listed in the IntelliSense box.

Master It How would you declare and initialize a few variables?

Solution To declare multiple variables in a single statement, append each variable's name and type to the `Dim` statement:

```
Dim speed As Single, distance As Integer
```

Variables of the same type can be separated with commas, and you need not repeat the type of each variable:

```
Dim First, Last As String, BirthDate As Date
```

To initialize the variables, append the equals sign and the value as shown here:

```
Dim speed As Single = 75.5, distance As Integer = 14902
```

Master It Explain briefly the Explicit, Strict, and Infer options.

Solution These three options determine how Visual Basic handles variable types, and they can be turned on or off. The Explicit option requires that you declare all variables in your code before using them. When this option is off, you can use a variable in your code without declaring it. The compiler will create a new variable of the Object type. The Strict option requires that you declare variables with a specific type. If the Strict option is off, you can declare variables without a type, with a statement like this:

```
Dim var1, var2
```

The last option, Infer, allows you to declare and initialize typed variables without specifying their type. The compiler infers the variable's type from the value assigned to it.

The following declarations will create a String and a Date variable, as long as the Infer option is on. Otherwise, they will create two object variables:

```
Dim D = #3/5/1008#, S = "my name"
```

Use the native data types. The CLR recognized the following data types, which you can use in your code to declare variables: Strings, Numeric types, Date and time types, Boolean data type.

All other variables, or variables that are declared without a type, are Object variables and can store any data type, or any object.

Master It How will the compiler treat the following statement?

```
Dim amount = 32
```

Solution The *amount* variable is not declared with a specific data type. With the default settings, the compiler will create a new object variable and store the value 32 in it. If the Infer option is on, the compiler will create an Integer variable and store the value 32 in it. If you want to be able to store *amount* with a fractional part in this variable, you must assign a floating-point value to the variable (such as 32.00), or append the R type character to the value (32 R).

Create custom data types. Practical applications need to store and manipulate multiple data items, not just integers and strings. To maintain information about people, we need to store each person's name, date of birth, address, and so on. Products have a name, a description, a price, and other related items. To represent such entities in our code, we use structures, which hold many pieces of information about a specific entity together.

Master It Create a structure for storing products and populate it with data.

Solution Structures are declared with the Structure keyword, and their fields with the Dim statement:

```
Structure Product
    Dim ProductCode As String
    Dim ProductName As String
    Dim Price As Decimal
    Dim Stock As Integer
End Structure
```

To represent a specific product, declare a variable of the Product type and set its fields, which are exposed as properties of the variable:

```
Dim P1 As Product
P1.ProductCode = "SR-0010"
P1.ProductName = "NTR TV-42"
P1.Price = 374.99
P1.Stock = 3
```

Use arrays. Arrays are structures for storing sets of data, as opposed to single-valued variables.

Master It How would you declare an array for storing 12 names and another one for storing 100 names and Social Security numbers?

Solution The first array stores a set of single-valued data (names) and it has a single dimension. Because the indexing of the array's elements starts at 0, the last element's index for the first array is 11, and it must be declared as

```
Dim Names(11) As String
```

The second array stores a set of pair values (names and SSNs), and it must be declared as a two-dimensional array:

```
Dim Persons(99,1) As String
```

Chapter 3: Programming Fundamentals

Use Visual Basic's flow-control statements. Visual Basic provides several statements for controlling the sequence in which statements are executed: decision statements, which change the course of execution based on the outcome of a comparison, and loop statements, which repeat a number of statements while a condition is true or false.

Master It Explain briefly the decision statements of Visual Basic.

Solution The basic decision statement in VB is the `If...End If` statement, which executes the statements between the `If` and `End If` keywords if the condition specified in the `If` part is True. A variation of this statement is the `If...Then...Else...End If` statements. If the same expression must be compared to multiple values and the program should execute different statements depending on the outcome of the comparison, use the `Select Case` statement.

Write subroutines and functions. To manage large applications, we break our code into small, manageable units. These units of code are the subroutines and functions. Subroutines perform actions and don't return any values. Functions, on the other hand, perform calculations and return values. Most of the language's built-in functionality is in the form of functions.

Master It How will you create multiple overloaded forms of the same function?

Solution Overloaded functions are variations of the same function with different arguments. All overloaded forms of a function have the same name, and they're prefixed with the `Overloads` keyword. Their lists of arguments, however, are different — either in the number of arguments or in their types.

Pass arguments to subroutines and functions. Procedures and functions communicate with one another via arguments, which are listed in a pair of parentheses following the procedure's name. Each argument has a name and a type. When you call the procedure, you must supply values for each argument and the types of the values should match the types listed in the procedure's definition.

Master It Explain the difference between passing arguments by value and passing arguments by reference.

Solution The first mechanism, which was the default mechanism with earlier versions of the language, passes a reference to the argument. Arguments passed by reference are prefixed by the keyword `ByRef` in the procedure's definition. The procedure has access to the original values of the arguments passed by reference and can modify them.

The second mechanism passes to the procedure a copy of the original value. Arguments passed by value are prefixed with the keyword `ByVal` in the procedure's definition. The procedure may change the values of the arguments passed by value, but the changes won't affect the value of the original variable.

Chapter 4: GUI Design and Event-Driven Programming

Design graphical user interfaces. A Windows application consists of a graphical user interface and code. The interface of the application is designed with visual tools and consists of controls that are common to all Windows applications. You drop controls from the Toolbox window onto the form, size and align the controls on the form, and finally set their properties through the Properties window. The controls include quite a bit of functionality right out of the box, and this functionality is readily available to your application without a single line of code.

Master It Describe the process of aligning controls on a form.

Solution To align controls on a form, you should select them in groups, according to their alignment. Controls can be aligned to the left, right, top, and bottom. After selecting a group of controls with a common alignment, apply the proper alignment with one of the commands of the Format ➢ Align menu. Before aligning multiple controls, you should adjust their spacing. Select the controls you want to space vertically or horizontally and adjust their spacing with one of the commands of the Format ➢ Horizontal Spacing and Format ➢ Vertical Spacing methods. You can also align controls visually, by moving them with the mouse. As you move a control around, a blue snap line appears every time the control is aligned with another one on the form.

Program events. Windows applications follow an event-driven model: We code the events to which we want our application to respond. The `Click` events of the various buttons are typical events to which an application reacts. You select the actions to which you want your application to react and program these events accordingly.

When an event is fired, the appropriate event handler is automatically invoked. Event handlers are subroutines that pass two arguments to the application: the `sender`object (which is an object that represents the control that fired the event) and the `e` argument (which carries additional information about the event).

Master It How will you handle certain keystrokes regardless of the control that receives them?

Solution You can intercept all keystrokes at the form's level by setting the form's `KeyPreview` property to True. Then insert some code in the form's `KeyPress` event handler to examine the keystroke passed to the event handler and process it. To detect the key presses in the `KeyPress` event handler, use an `If` statement like the following:

```
If e.KeyChar = "A" Then
'   process the A key
End If
```

Write robust applications with error handling. Numerous conditions can cause an application to crash, but a professional application should be able to detect abnormal conditions and handle them gracefully. To begin with, *you should always validate your data* before you attempt to use them in your code. A well-known computer term is ''garbage in, garbage out'', which means you shouldn't perform any calculations on invalid data.

Master It How will you execute one or more statements in the context of a structured exception handler?

Solution A structured exception handler has the following syntax:

```
Try
   {statements}
Catch ex As Exception
   {statements to handle exception}
End Try
```

The statements you want to execute must be inserted in the `Try` block of the statement. If executed successfully, program execution continues with the statements following the `End Try` statement. If an error occurs, the `Catch` block is activated, where you can display the appropriate message and take the proper actions. At the very least, you should save the user data and then terminate the application. In many cases, it's even possible to remedy the situation that caused the exception in the first place.

Chapter 5: The Vista Interface

Create a simple WPF application. WPF is a new and powerful technology for creating user interfaces. WPF is one of the core technologies in the .NET Framework 3.5 and is integrated into Windows Vista. WPF is also supported on Windows XP. WPF takes advantage of the graphics engines and display capabilities of the modern computer and is vector based and resolution independent.

Master It Develop a simple ''Hello World'' type of WPF application that displays a Button control and Label control. Clicking the button should set the content property of a Label control to *Hi There!*

Solution Complete the following steps.

1. Open Visual Studio 2008 and choose File ➢ New Project.
2. From the New Project dialog box, select WPF Application and click OK.
3. From the Toolbox, drag a Button control and Label control to `Window1` on the design surface.
4. Double-click the Button control and add the following line of code to the `Button1_Click` event in code-behind:

```
Label1.Content = "Hi There!"
```

Data-bind controls in WPF. The ability to bind controls to a data source is an essential aspect of separating the UI from the business logic in an application.

Master It Data-bind a Label control to one field in a record returned from a database on your computer.

Solution Complete the following steps.

1. Open Visual Studio 2008 and create a new WPF project.
2. Establish a link to an existing database on your system by opening the Server Explorer window (click the appropriate tab at the bottom of the Toolbox area of Visual Studio) and right-clicking Data Connections. Choose Add Connection from the context menu and follow the prompts. Note that you use the Microsoft SQL Server Database File option if you are planning to connect to a database created by SQL Server Express (the default database system that ships with Visual Studio 2008).
3. Open the Data Sources window by clicking the tab at the bottom of the Server Explorer window, and then click the Add New Data Source link. This opens the Data Source Configuration Wizard. Follow the prompts to set up the dataset.
4. Switch to code-behind (`Window1.xaml.vb`) for `Window1.xaml`. Add the following code. Note that the ContactsDataSet, ContactsDataSetTableAdapters, and CustomersTableAdapter names will vary according to the actual names that you have set up for your dataset.

```
Class Window1
    Dim mydataset As New ContactsDataSet
    Dim mydataAdapter As New _
 ContactsDataSetTableAdapters.CustomersTableAdapter
    Private Sub Window1_Loaded(ByVal sender As Object, _
 ByVal e As System.Windows.RoutedEventArgs) Handles Me.Loaded
        mydataAdapter.Fill(mydataset.Tables(0))
        Me.DataContext = mydataset.Tables(0).Rows(0)
    End Sub
End  Class
```

5. Switch back to XAML view for `Window1.xaml` and add the following markup (without line breaks). You may need to change the `FirstName` reference in the binding for `Label1` to whichever database field that you want displayed:

```
<Window x:Class="Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300">
    <Grid>
      <Label Margin="38,129,47,90" Name="Label1" Content="{Binding _
 Path=FirstName}" ></Label>

    </Grid>
</Window>
```

6. Run the application. The contents of your nominated field from the first table indexed in your database (0) should be displayed in the Label control.

Use a data template to control data presentation. WPF enables a very flexible approach to presenting data by using data templates. The developer can create and fully customize data templates for data formatting.

Master It Create a data template to display a Name, Surname, Gender combination in a horizontal row in a ComboBox control. Create a simple array and class of data to feed the application.

Solution Complete the following steps.

1. Add the following code to the XAML source for `Window1.xaml` (delete the line breaks):

```
<Window x:Class="Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="300" Width="300">

   <Grid Name="myGrid">
      <Grid.Resources>
         <DataTemplate x:Key="NameStyle">
            <Grid>
               <Grid.ColumnDefinitions>
                  <ColumnDefinition Width="60" />
                  <ColumnDefinition Width="60" />
                  <ColumnDefinition Width="*" />
               </Grid.ColumnDefinitions>
               <TextBlock Grid.Column="0" _
Text="{Binding Path=FirstName}" />
               <TextBlock Grid.Column="1" _
Text="{Binding Path=Surname}" />
```

```
                <TextBlock Grid.Column="2" _
Text="{Binding Path=Gender}" />
                 </Grid>
           </DataTemplate>
        </Grid.Resources>
        <ComboBox _
           ItemTemplate="{StaticResource NameStyle}" _
ItemsSource="{Binding }" IsSynchronizedWithCurrentItem="true" _
Height="25" VerticalAlignment="Top" Name="ComboBox1"  />
    </Grid>
</Window>
```

2. Switch to code-behind for `Window1.xaml` (`Window1.xaml.vb`) and add the following code:

```
Class Window1
    Private Class myList
        Dim _name As String
        Dim _surname As String
        Dim _gender As String
        Public Sub New(ByVal FirstName As String, ByVal _
 Surname As String, ByVal Gender As String)
            _name = FirstName
            _surname = Surname
            _gender = Gender
        End Sub
        Public ReadOnly Property FirstName() As String
            Get
                Return _name
            End Get
        End Property
        Public ReadOnly Property Surname() As String
            Get
                Return _surname
            End Get
        End Property
        Public ReadOnly Property Gender() As String
            Get
                Return _gender
            End Get
        End Property
    End Class


    Private Sub Window1_Loaded(ByVal sender As Object, _
 ByVal e As System.Windows.RoutedEventArgs) Handles Me.Loaded
```

```
                Dim myArray As New ArrayList
                myArray.Add(New MyList("Fred", "Bloggs", "M"))
                myArray.Add(New MyList("Mary", "Green", "F"))
                myArray.Add(New MyList("Sally", "Smith", "F"))
                myArray.Add(New MyList("John", "Doe", "M"))
                myArray.Add(New MyList("Jemma", "Bloggs", "F"))

                Me.DataContext = myArray

            End Sub
        End  Class
```

3. Run the application. A combo box containing a list of the contacts should be displayed as FirstName, Surname, and Gender.

Chapter 6: Basic Windows Controls

Use the TextBox control as a data-entry and text-editing tool. The TextBox control is the most common element of the Windows interface, short of the Button control, and it's used to display and edit text. You can use a TextBox control to prompt users for a single line of text (such as a product name) or a small document (a product's detailed description).

Master It What are the most important properties of the TextBox control? Which ones would you set in the Properties windows at design-time?

Solution First you must decide whether you want the control to hold a single line of text or multiple text lines, and set the `MultiLine` property accordingly. You must also decide whether the control should wrap words automatically and then set the `WordWrap` and `ScrollBars` properties accordingly. If you want the control to display some text initially, set the control's `Text` property to the desired text. At runtime you can retrieve the text entered by the user in the control, with the same property. Another property, the `Lines` array, allows you to retrieve individual paragraphs of text. Each paragraph can be broken into multiple text lines on the control, but each is stored in a single element of the `Lines` array.

Master It How will you implement a control that suggests lists of words matching the characters entered by the user?

Solution Use the autocomplete properties `AutoCompleteMode`, `AutoCompleteSource`, and `AutoCompleteCustomSource`. The `AutoComplete` property determines whether the control will suggest the possible strings, automatically complete the current word as you type, or do both. The `AutoCompleteSource` property specifies where the strings that will be displayed will come from, and its value is a member of the `AutoCompleteSource` enumeration. If this property is set to `AutoCompleteSoure.CustomSource`, you must set up an `AutoCompleteStringCollection` collection with your custom strings and assign it to the `AutoCompleteCustomCource` property.

Use the ListBox, CheckedListBox, and ComboBox controls to present lists of items. The ListBox control contains a list of items from which the user can select one or more, depending on the setting of the `SelectionMode` property.

Master It How will you locate an item in a ListBox control?

Solution To locate a string in a ListBox control, use the `FindString` and `FindStringExact` methods. The `FindString` method locates a string that partially matches the one you're searching for; `FindStringExact` finds an exact match. Both methods perform case-insensitive searches and return the index of the item they've located in the list.

We usually call the `FindStringExact` method and then examine its return value. If an exact match was found, we select the item with the index returned by the `FindStringExact` method. If an exact match was not found, in which case the method returns −1, we call the `FindString` method to locate the nearest match.

Use the ScrollBar and TrackBar controls to enable users to specify sizes and positions with the mouse. The ScrollBar and TrackBar controls let the user specify a magnitude by scrolling a selector between its minimum and maximum values. The ScrollBar control uses some visual feedback to display the effects of scrolling on another entity, such as the current view in a long document.

Master It Which event of the ScrollBar control would you code to provide visual feedback to the user?

Solution The ScrollBar control fires two events: the `Scroll` event and the `ValueChanged` event. They're very similar, and you can program either event to react to the changes in the ScrollBar control. The advantage of the `Scroll` event is that it reports the action that caused it through the `e.Type` property. You can examine the value of this property in your code and react to actions such as the end of the scroll:

```
Private Sub blueBar_Scroll( _
        ByVal sender As System.Object, _
        ByVal e As System.Windows.Forms.ScrollEventArgs)_
        Handles blueBar.Scroll
 If e.Type = ScrollEventType.EndScroll Then
    ' perform calculations and provide feedback
 End If
End Sub
```

Chapter 7: Working with Forms

Use forms' properties. Forms expose a lot of trivial properties for setting their appearance. In addition, they expose a few properties that simplify the task of designing forms that can be resized at runtime. The `Anchor` property causes a control to be anchored to one or more edges of the form to which it belongs. The `Dock` property allows you to place on the form controls that are docked to one of its edges. To create forms with multiple panes that the user can resize at runtime, use the SplitContainer control. If you just can't fit all the controls in a reasonably sized form, use the `AutoScroll` properties to create a scrollable form.

Master It You've been asked to design a form with three distinct sections. You should also allow users to resize each section. How will you design this form?

Solution The type of form required is easily designed with visual tools and the help of the SplitContainer control. Place a SplitContainer control on the form and set its `Dock` property to Fill. You've just created two vertical panes on the form, and users can change their relative sizes at any time. To create a third pane, place another SplitContainer control on one of the first control's panes and set its `Dock` property to Fill, and its `Orientation` property to Horizontal. At this point, the form is covered by three panes, and users can change each pane's size at the expense of its neighboring panes.

Design applications with multiple forms. Typical applications are made up of multiple forms: the main form and one or more auxiliary forms. To show an auxiliary form from within the main form's code, call the auxiliary form's `Show` method, or the `ShowDialog` method if you want to display the auxiliary form modally (as a dialog box).

Master It How will you set the values of selected controls in a dialog box, display them, and then read the values selected by the user from the dialog box?

Solution Create a Form variable that represents the dialog box and then access any control in the dialog box through its name as usual, prefixed by the form's name:

```
Dim Dlg As AuxForm
Dlg.txtName  =  "name"
```

Then call the form's `ShowDialog` method to display it modally and examine the `DialogResult` property returned by the method. If this value is OK, process the data in the dialog box, or else ignore them:

```
If Dlg.ShowDialog = DialogResult.OK Then
    UserName = Dlg.TxtName
End  If
```

To display an auxiliary form, just call the `Show` method. This method doesn't return a value, and you can read the auxiliary form's contents from within the main form's code at any time. You can also access the controls of the main form from within the auxiliary form's code.

Design dynamic forms. You can create dynamic forms by populating them with controls at runtime through the form's `Controls` collection. First, create instances of the appropriate controls by declaring variables of the corresponding type. Then set the properties of the variable that represents the control. Finally, place the control on the form by adding it to the form's `Controls` collection.

Master It How will you add a TextBox control to your form at runtime and assign a handler to the control's `TextChanged` event?

Solution Create an instance of the TextBox control, set its `Visible` property, and add it to the form's `Controls` collection:

```
Dim TB As New TextBox
TB.Visible = True
' statements to set other properties,
' including the control's location on the form
Me.Controls.Add(TB)
```

Then write a subroutine that will handle the `TextChanged` event. This subroutine, let's call it `TBChanged()`, should have the same signature as the TextBox control's `TextChanged` event. Use the `AddHandler` statement to associate the `TBChanged()` subroutine with the new control's `TextChanged` event:

```
AddHandler TB.TextChanged, _
    New SystemEventHandler(AddressOf TBChanged)
```

Design menus. Both form menus and context menus are implemented through the MenuStrip control. The items that make up the menu are ToolStripMenuItem objects. The ToolStripMenuItem objects give you absolute control over the structure and appearance of the menus of your application.

Master It What are the two basic events fired by the ToolStripMenuItem object?

Solution When the user clicks a menu item, the `DropDownOpened` and `Click` events are fired, in this order. The `DropDownOpened` event gives you a chance to modify the menu that's about to be opened. After the execution of the `DropDownOpened` event handler, the `Click` event takes place to indicate the selection of a menu command. We rarely program the `DropDownOpened` event, but every menu item's `Click` event handler should contain some code to react to the selection of the item.

Chapter 8: More Windows Controls

Use the OpenFileDialog and SaveFileDialog controls to prompt users for filenames. Windows applications use certain controls to prompt users for common information, such as filenames, colors, and fonts. Visual Studio provides a set of controls, which are grouped in the Dialogs section of the Toolbox. All common dialog controls provide a `ShowDialog` method, which displays the corresponding dialog box in a modal way. The `ShowDialog` method returns a value of the `DialogResult` type, which indicates how the dialog box was closed, and you should examine this value before processing the data.

Master It Your application needs to open an existing file. How will you prompt users for the file's name?

Solution First you must drop an instance of the OpenFileDialog control on the form. To limit the files displayed in the Open dialog box, use the `Filter` property to specify the relevant file type(s). To display text files only, set the `Filter` property to `Text files|*.txt`. If you want to display multiple extensions, use a semicolon to separate extensions with the `Filter` property; for example, the string `Images|*.BMP;*.GIF;*.JPG`will cause the control to select all the files of these three types and no others. The first part of the expression (`Images`) is the string that will appear in the drop-down list with the file types. You should also set the `CheckFileExists` property to True to make sure that the file specified on the control exists. Then display the Open dialog box by calling its `ShowDialog` method, as shown here:

```
If FileOpenDialog1.ShowDialog = _
      Windows.Forms.DialogResult.OK
       {process file FileOpenDialog1.FileName}
End If
```

To retrieve the selected file, use the control's `FileName` property, which is a string with the selected file's path.

Master It You're developing an application that encrypts multiple files (or resizes many images) in batch mode. How will you prompt the user for the files to be processed?

Solution There are two techniques to prompt users for multiple filenames. Both techniques, however, are limited in the sense that all files must reside in the same folder. The first technique is to set the `MultiSelect` property of the OpenFileDialog control to True. Users will be able to select multiple files by using the Ctrl and Shift keys. The selected files will be reported to your application through the `FileNames` property of the control, which is an array of strings.

```
OpenFileDialog1.Multiselect = True
OpenFileDialog1.ShowDialog()
Dim filesEnum As IEnumerator
ListBox1.Items.Clear()
filesEnum = _
      OpenFileDialog1.FileNames.GetEnumerator()
While filesEnum.MoveNext
    ' current file's name is filesEnum.Current
End While
```

The other technique is to use the FolderBrowserDialog control, which prompts users to select a folder, not individual files. Upon return, the control's `SelectedPath` property contains the pathname of the folder selected by the user from the dialog box, and you can use this property to process all files of a specific type in the selected folder.

Listing 8.2 earlier in this chapter shows you how to iterate through the files of the selected folder and all its subfolders.

Use the ColorDialog and FontDialog controls to prompt users for colors and typefaces. The Color and Font dialog boxes allow you to prompt users for a color value and a font, respectively. Before showing the corresponding dialog box, set its `Color` or `Font` property according to the current selection, and then call the control's `ShowDialog` method.

Master It How will you display color attributes in the Color dialog box when you open it? How will you display the attributes of the selected text's font in the Font dialog box when you open it?

Solution To prompt users to specify a different color for the text on a TextBox control, execute the following statements:

```
ColorDialog1.Color = TextBox1.ForeColor
If ColorDialog1.ShowDialog = DialogResult.OK Then
    TextBox1.ForeColor = ColorDialog1.Color
End  If
```

To populate the Font dialog box with the font in effect, assign the control's `Font` property to the FontDialog control's `Font` property by using the following statements:

```
FontDialog1.Font = TextBox1.Font
If FontDialog1.ShowDialog = DialogResult.OK Then
    TextBox1.Font = FontDialog1.Font
End  If
```

Use the RichTextBox control as an advanced text editor to present richly formatted text. The RichTextBox control is an enhanced TextBox control that can display multiple fonts and styles, format paragraphs with different styles, and provide a few more advanced text-editing features. Even if you don't need the formatting features of this control, you can use it as an alternative to the TextBox control. At the very least, the RichTextBox control provides more editing features, a more-useful undo function, and more-flexible search features.

Master It You want to display a document with a title in large, bold type, followed by a couple of items in regular style. How will you create a document like the following one on a RichTextBox control?

Document's Title
Item 1
Description for item 1
Item 2
Description for item 2

Solution To append text to a RichTextBox control, use the `AppendText` method. This method accepts a string as an argument and appends it to the control's contents. The text is formatted according to the current selection's font, which you must set accordingly through the `SelectionFont` property. To switch to a different font set the `SelectionFont` again and call the `AppendText` method.

Assuming that the form contains a control named `RichTextBox1`, the following statements will create a document with multiple formats. In this sample I'm using three different typefaces for the document.

```
Dim fntTitle As _
        New Font("Verdana", 12, FontStyle.Bold)
Dim fntItem As _
        New Font("Verdana", 10, FontStyle.Bold)
Dim fntText As _
        New Font("Verdana", 9, FontStyle.Regular)
Editor.SelectionFont = fntTitle
Editor.AppendText("Document's Title" & vbCrLf)
Editor.SelectionFont = fntItem
Editor.SelectionIndent = 20
Editor.AppendText("Item 1" & vbCrLf)
Editor.SelectionFont = fntText
Editor.SelectionIndent = 40
Editor.AppendText( _
              "Description for item 1" & vbCrLf)
Editor.SelectionFont = fntItem
Editor.SelectionIndent = 20
Editor.AppendText("Item 2" & vbCrLf)
Editor.SelectionFont = fntText
Editor.SelectionIndent = 40
Editor.AppendText( _
           "Description for item 2" & vbCrLf)
```

Chapter 9: The TreeView and ListView Controls

Create and present hierarchical lists by using the TreeView control. The TreeView control is used to display a list of hierarchically structured items. Each item in the TreeView control is represented by a TreeNode object. To access the nodes of the TreeView control, use the `TreeView.Nodes` collection. The nodes under a specific node (in other words, the child nodes) form another collection of Node objects, which you can access by using the expression `TreeView.Nodes(i).Nodes`. The basic property of the Node object is the `Text` property, which stores the node's caption. The Node object exposes properties for manipulating its appearance (its foreground/background color, its font, and so on).

Master It How will you set up a TreeView control with a book's contents at design time?

Solution Place an instance of the TreeView control on the form and then locate its `Nodes` property in the Properties Browser. Click the ellipsis button to open the TreeNode Editor dialog box, where you can enter root nodes by clicking the Add Root button, and child nodes under the currently selected node by clicking the Add Child button. The book's chapters should be the control's root nodes, and the sections should be child nodes of those chapter nodes. If you have nested sections, add them as child nodes of the appropriate node. While a node is selected in the left pane of the dialog box, you can specify its appearance in the right pane by setting the font, color, and image-related properties.

Create and present lists of structured items by using the ListView control. The ListView control stores a collection of ListViewItem objects, the `Items` collection, and can display them in several modes, as specified by the `View` property. Each ListViewItem object has a `Text` property and the `SubItems` collection. The subitems are not visible at runtime unless you set the control's `View` property to Details and set up the control's `Columns` collection. There must be a column for each subitem you want to display on the control.

Master It How will you set up a ListView control with three columns to display names, emails, and phone numbers at design time?

Solution Drop an instance of the ListView control on the form and set its `View` property to Details. Then locate the control's `Columns` property in the Properties Browser and add three columns to the collection through the ColumnHeader Collection Editor dialog box. Don't forget to set their headers and their widths for the fields they will display.

To populate the control at design time, locate its `Items` property in the Properties window and click the ellipsis button to open the ListViewItem Collection Editor dialog box (see Figure 9.11). Add a new item by clicking the Add button. When the new item is added to the list in the left pane of the dialog box, set its `Text` property to the desired caption. To add subitems to this item, locate the `SubItems` property in the ListViewItem Collection Editor dialog box and click the ellipsis button next to its value. This will open another dialog box, the ListViewSubItems Collection, where you can add as many subitems under the current item as you wish. You can also set the appearance of the subitems (their font and color) in the same dialog box.

Master It How would you populate the same control with the same data at runtime?

Solution The following code segment adds two items to the *ListView1* control at runtime:

```
Dim LItem As New ListViewItem()
LItem.Text = "Alfred's Futterkiste"
LItem.SubItems.Add("Anders Maria")
LItem.SubItems.Add("030-0074321")
LItem.SubItems.Add("030-0076545")
LItem.ImageIndex = 0
ListView1.Items.Add(LItem)

LItem = New ListViewItem()
LItem.Text = "Around the Horn"
LItem.SubItems.Add("Hardy Thomas")
LItem.SubItems.Add("(171) 555-7788")
LItem.SubItems.Add("(171) 555-6750")
LItem.ImageIndex = 0
ListView1.Items.Add(LItem)
```

Chapter 10: Building Custom Classes

Build your own classes. Classes contain code that executes without interacting with the user. The class's code is made up of three distinct segments: the declaration of the private variables, the property procedures that set or read the values of the private variables, and the methods, which are implemented as Public subroutines or functions. Only the Public entities (properties and methods) are accessible by any code outside the class. Optionally, you can implement events that are fired from within the class's code. Classes are referenced through variables of the appropriate type, and applications call the members of the class through these variables. Every time a method is called, or a property is set or read, the corresponding code in the class is executed.

Master It How do you implement properties and methods in a custom class?

Solution Any variable declared with the `Public` access modifier is automatically a property. As a class developer, however, you should be able to validate any values assigned to your class's properties. To do so, you can implement properties by using a special type of procedure, the Property procedure, which has two distinct segments: a `Set` segment that's invoked when an application attempts to set a property, and a `Get` segment that's invoked when an application attempts to read a property's value. The Property has the following structure:

```
Private m_property As type
Property Property() As type
   Get
     Property = m_property
```

```
    End Get
    Set (ByVal value As type)
    ' your validation code goes here
    ' If validation succeeds, set the local var
      m_property = value
    End Set
End Property
```

The local variable `m_property` must be declared with the same type as the property. The `Get` segment returns the value of the local variable that stores the property's value. The `Set` segment validates the value passed by the calling application and either rejects it or sets the local variable to this value.

Master It How would you use a constructor to allow developers to create an instance of your class and populate it with initial data?

Solution Each class has a constructor, which is called every time a new instance of the class is created with the `New` keyword. The constructor is implemented with the `New()` subroutine. To allow users to set certain properties of the class when they instantiate it, create as many `New()` subroutines as you need. Each version of the `New()` subroutine should accept different arguments. The following sample lets you create objects that represent books, passing the book's ISBN and/or title.

```
Public Sub New(ByVal ISBN As String)
    MyBase.New()
    Me.ISBN = ISBN
End Sub

Public Sub New(ByVal ISBN As String, _
               ByVal Title As String)
    MyBase.New()
    Me.ISBN = ISBN
    Me.Title = Title
End Sub
```

Use custom classes in your projects. To use a custom class in your project, you must add to the project a reference to the class you want to use. If the class belongs to the same project, you don't have to do anything. If the class belongs to another project, you must right-click the project's name in the Solution Explorer and select Add Reference from the shortcut menu. In the Add Reference dialog box that appears, switch to the Browse tab and locate the DLL file with the class's implementation (it will be a DLL file in the project's Bin folder). Select the name of this file and click OK to add the reference and close the dialog box.

Master It How will you call the two constructors of the preceding Master It sections in an application that uses the custom class to represent books?

Solution There are three ways to create new Book objects:

1. Call the parameterless constructor to create books without ISBNs or titles:

```
Dim book As New Book
```

2. Call the `New()` constructor, passing the book's ISBN as a parameter:

```
Dim book As New Book("9780213324543")
```

3. Call the `New()` constructor, passing the book's ISBN and title:

```
Dim book As New Book("9780213324543", _
     "Mastering Visual Studio")
```

Customize the usual operators for your classes. Overloading is a common theme in coding classes (or plain procedures) with Visual Basic. In addition to overloading methods, you can overload operators. In other words, you can define the rules for adding or subtracting two custom objects, if this makes sense for your application.

Master It When should you overload operators in a custom class, and why?

Solution Sometimes it makes sense to apply common operations, such as the addition and subtraction operations, to instances of a custom class. However, the addition operator doesn't work with custom classes. To redefine the addition operator so that it will add two instances of your custom class, you must override the addition operator with an implementation that adds two instances of a custom class. The following is the signature of a function that overloads the addition operator:

```
Public Shared Operator + ( _
            ByVal object1 As customType, _
            ByVal object2 As customType) _
            As customType

    Dim result As New customType
    ' Insert the code to "add" the two
    ' arguments and store the result to
    ' the result variable and return it.
    Return result
End Operator
```

The function that overrides the addition operator accepts two arguments, which are the two values to be added, and returns a value of the same type. The operator is usually overloaded, because you may wish to add an instance of the custom class to one of the built-in data types or objects. In addition to the usual math operators, you should also consider overloading some basic functions that act like operators, especially the `CType()` function.

Chapter 11: Working with Objects

Use inheritance. Inheritance, which is the true power behind OOP, allows you to create new classes that encapsulate the functionality of existing classes without editing their code. To inherit from an existing class, use the `Inherits` statement, which brings the entire class into your class.

Master It Explain the inheritance-related attributes of a class's members.

Solution Any class can be inherited by default. However, you can prevent developers from inheriting your class with the `NonInheritable` keyword, or create an abstract class with the `MustInherit` attribute. Classes marked with this attribute can't be used on their own; they must be inherited by another class. The parent class's members can be optionally overridden if they're marked with the `Overridable` keyword. To prevent derived classes from overriding specific members, use the `NotOverridable` attribute. Finally, methods that override the equivalent methods of the base class must be prefixed with the `Overrides` keyword.

Use polymorphism. Polymorphism is the ability to write members that are common to a number of classes but behave differently, depending on the specific class to which they apply. Polymorphism is a great way of abstracting implementation details and delegating the implementation of methods with very specific functionality to the derived classes.

Master It The parent class Person represents parties, and it exposes the `GetBalance` method, which returns the outstanding balance of a person. The Customer and Supplier derived classes implement the `GetBalance` method differently. How will you use this method to find out the balance of a customer and/or supplier?

Solution If you have Customer or Supplier object, you can call the `GetBalance` method directly. If you have a collection of objects of both types, you must cast them to their parent type and then call the `GetBalance` method.

Chapter 12: Building Custom Windows Controls

Extend the functionality of existing Windows Forms controls with inheritance. The simplest type of control you can build is one that inherits an existing control. The inherited control includes all the functionality of the original control plus some extra functionality that's specific to an application and that you implement with custom code.

Master It Describe the process of designing an inherited custom control.

Solution To enhance an existing Windows Forms control, insert an `Inherits` statement with the name of the control you want to enhance in the project's `Designer.vb` file. The inherited control's interface can't be altered; it's determined by parent control. However, you can implement custom properties and methods, react to events received by the parent control, or raise custom events from within your new control.

The process of implementing custom properties and methods is identical to building custom classes. The control's properties, however, can be prefixed by a number of useful attributes, such as the <`Category`> and <`Description`> attributes, which determine the category of the Properties window where the property will appear, and the control's description that will be shown in the Properties window when the custom property is selected.

Build compound controls that combine multiple existing controls. A compound control provides a visible interface that combines multiple Windows controls. As a result, this type of control doesn't inherit the functionality of any specific control; you must expose its properties by providing your own code. The UserControl object, on which the compound control is based, already exposes a large number of members, including some fairly advanced ones such as the `Anchoring` and `Docking` properties, and the usual mouse and key events.

Master It How will you map certain members of a constituent control to custom members of the compound control?

Solution If the member is a property, you simply return the constituent control's property value in the `Get` section of the Property procedure, and set the constituent control's property to the specified value in the `Set` section of the same procedure. The following Property procedure maps the `WordWrap` property of the `TextBox1` constituent control to the `TextWrap` property of the custom compound control:

```
Public Property TextWrap() As Boolean
    Get
        Return TextBox1.WordWrap
    End Get
    Set(ByVal value As Boolean)
        TextBox1.WordWrap = value
    End Set
End Property
```

If the member is a method, you just call it from within one of the compound control's methods. To map the `ResetText` method of the TextBox constituent control to the `Reset` method of the compound control, add the following method definition:

```
Public Sub Reset()
    TextBox1.ResetText
End Sub
```

Build custom controls from scratch. User-drawn controls are the most flexible custom controls, because you're in charge of the control's functionality and appearance. Of course, you have to implement all the functionality of the control from within your code, so it takes substantial programming effort to create user-drawn custom controls.

Master It Describe the process of developing a user-drawn custom control.

Solution Because you are responsible for updating the control's visible area from within your code, you must provide the code that redraws the control's surface and insert it in the UserControl object's `Paint` event handler. In drawing the control's surface, you must take into consideration the settings of the control's properties.

The `e` argument of the `Paint` event handler exposes the `Graphics` property, which you must use from within your code to draw on the control's surface. You can use any of the drawing methods you'd use to create shapes, gradients, and text on a Form or PictureBox control. Because custom controls aren't redrawn by default when they're resized, you must also insert the following statement in the control's `Load` event handler:

```
Me.SetStyle(ControlStyles.ResizeRedraw, True)
```

If the control's appearance should be different at design time than at runtime, use the `Me.DesignMode` property to distinguish between runtime and design time.

Customize the rendering of items in a ListBox control. To create an owner-drawn list control, you must set the `DrawMode` property to a member of the `DrawMode` enumeration and program two events: `MeasureItem` and `DrawItem`.

Master It Outline the process of creating a ListBox control that wraps the contents of lengthy items.

Solution By default, all items in a ListBox control have the same height, which is the height of a single line of text in the control's font. To display selected items in cells of varying height, do the following:

1. Set the control's `DrawMode` property to *`OwnerDrawnVariable`*
2. In the control's `MeasureItem` event handler, which is invoked every time the control is about to display an item, insert the statements that calculate the desired height of the current item's cell and set the `e.Height` property. You will most likely call the `MeasureString` method of the control's Graphics object to retrieve the height of the item from its text.
3. In the control's `DrawItem` event handler, which displays the current item, insert the statements to print the item in a cell with the dimensions calculated in step 2 via the `DrawString` method. These dimensions are given by the `Bounds` property of the event handler's `e` argument.

Chapter 13: Handling Strings, Characters, and Dates

Use the Char data type to handle characters. The Char data type, which is implemented with the Char class, exposes methods for handling individual characters (`IsLetter`, `IsDigit`, `IsSymbol`, and so on). We use the methods of the Char class to manipulate users' keystrokes as they happen in certain controls (mostly the TextBox control) and to provide immediate feedback.

Master It You want to develop an interface that contains several TextBox controls that accept numeric data. How will you intercept the user's keystrokes and reject any characters that are not numeric?

Solution You must program the control's `KeyPress` event handler, which reports the character that was pressed. The following event handler rejects any non-numeric characters entered in the *`TextBox1`* control:

```
Private Sub TextBox1_KeyPress( _
        ByVal sender As Object, ByVal e As _
        System.Windows.Forms.KeyPressEventArgs) _
        Handles TextBox1.KeyPress
   Dim c As Char
   c = e.KeyChar
   If Not (Char.IsDigit(c) or _
           Char.IsControl(c)) Then
      e.Handled = True
   End If
End Sub
```

Actually, you learned in the previous chapter how to implement custom TextBox controls that inherit existing Windows controls. You can build a custom TextBox control that accepts numeric data along the lines of the design of the FocusedTextBox custom control, discussed in the preceding chapter.

Use the String data type to handle strings. The String data type represents strings and exposes members for manipulating them. Most of the String class's methods are equivalent to the string-manipulation methods of Visual Basic. The members of the String class are shared: they do not modify the string to which they're applied. Instead, they return a new string.

Master It How would you extract the individual words from a large text document?

Solution Start by setting up an array with all possible delimiters. The delimiters array should contain all symbols that separate words, including parentheses, brackets, and so on. Here's the definition of such an array that works even with program listings:

```
Dim delimiters() As Char = _
        {" "c, "."c, ","c, "!"c, ";"c, ":"c, _
         "("c, ")"c, "*"c, """"c, ";"c, "{"c, _
         "}"c, Convert.ToChar(vbTab), _
         Convert.ToChar(vbCr), _
        Convert.ToChar(vbLf)}
```

Notice that `vbTab`, `vbCr`, and `vbLf` constants are strings, and they must be converted implicitly into characters.
Then pass this array as an argument to the `Split` method and retrieve the method's results in an array of strings. These are the words extracted from the text by the `Split` method:

```
Dim words() As String
words = text.Split(delimiters)
Dim word As String
For Each word In words
   If word.Length > 0 Then
   ' process current word
    End If
Next
```

Use the StringBuilder class to manipulate large or dynamic strings. The StringBuilder class is very efficient at manipulating long strings, but it doesn't provide as many methods for handling strings. The StringBuilder class provides a few methods to insert, delete, and replace characters within a string. Unlike the equivalent methods of the String class, these methods act directly on the string stored in the current instance of the StringBuilder class.

Master It Assuming that you have populated a ListView control with thousands of lines of data from a database, how will you implement a function that copies all the data to the Clipboard?

Solution To copy the ListView control's data, you must create a long string that contains tab-delimited strings and then copy it to the Clipboard. Each cell's value must be converted to a string and then appended to a StringBuilder variable. Consecutive rows will be separated by a carriage return/line feed character. Start by declaring a StringBuilder variable:

```
Dim SB As New System.Text.StringBuilder
```

Then write a loop that iterates through the items of the ListView control:

```
Dim LI As ListViewItem
For Each LI In ListView1.Items
    ' append current row's cell values to SB
    SB.Append(vbcrlf)
Next
```

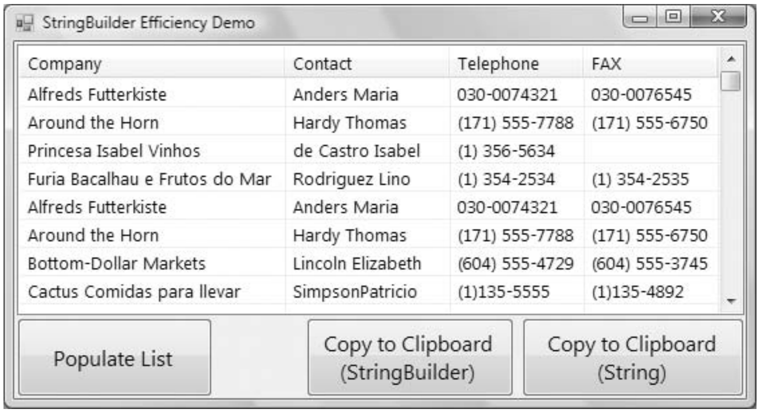
In the loop's body, insert another loop to iterate through the subitems of the `LI` item:

```
Dim LI As ListViewItem
For Each LI In ListView1.Items
    Dim SLI As ListViewItem.ListViewSubItem
    For Each SLI In LI.SubItems
        SB.Append(SLI.Text & vbTab)
    Next
    SB.Remove(SB.Length - 1, 1) ' remove last tab
    SB.Append(vbCrLf)
Next
```

And finally, put the string to the Clipboard by using the following statement:

```
Clipboard.SetText(SB.ToString)
```

One of this chapter's projects is the SBDemo project, which populates a ListView control with data (it's the same few rows repeated over and over). Click the Populate List button several times to create a long list of items and then one of the other two buttons on the form that copy the data to the Clipboard either through a String or through a StringBuilder variable. As you will see, the StringBuilder class runs circles around the String data type when it comes to manipulating dynamic strings.


StringBuilder Efficiency Demo
Populate List
Copy to Clipboard (StringBuilder)
Copy to Clipboard (String)

Company	Contact	Telephone	FAX
Alfreds Futterkiste	Anders Maria	030-0074321	030-0076545
Around the Horn	Hardy Thomas	(171) 555-7788	(171) 555-6750
Princesa Isabel Vinhos	de Castro Isabel	(1) 356-5634	
Furia Bacalhau e Frutos do Mar	Rodriguez Lino	(1) 354-2534	(1) 354-2535
Alfreds Futterkiste	Anders Maria	030-0074321	030-0076545
Around the Horn	Hardy Thomas	(171) 555-7788	(171) 555-6750
Bottom-Dollar Markets	Lincoln Elizabeth	(604) 555-4729	(604) 555-3745
Cactus Comidas para llevar	SimpsonPatricio	(1)135-5555	(1)135-4892

Use the DateTime and TimeSpan classes to handle dates and times. The Date class represents dates and time, and it exposes many useful shared methods (such as the `IsLeap` method, which returns True if the year passed to the method as an argument is leap; the `DaysInMonth` method; and so on). It also exposes many instance methods (such as `AddYears`, `AddDays`, `AddHours`, and so on) for adding time intervals to the current instance of the Date class, as well as many options for formatting date and time values.

The TimeSpan class represents time intervals — from milliseconds to days — with the `FromDays`, `FromHours`, and even `FromMilliseconds` methods. The difference between two date variables is a TimeSpan value, and you can convert this value to various time units by using methods such as `TotalDays`, `TotalHours`, `TotalMilliseconds`, and so on. You can also add a TimeSpan object to a date variable to obtain another date variable.

Master It How will you use the TimeSpan class to accurately time an operation?

Solution To time an operation, you must create a DateTime variable and set it to the current date and time right before the statements you want to execute:

```
Dim T1 As DateTime = Now
```

Right after the statements you want to execute, create a new TimeSpan object that represents the time it took the statements to complete. This duration is the difference between the current time and the time value stored in the variable *`T1`*:

```
Dim duration As New TimeSpan
duration = Now.Subtract(T1)
```

The *`duration`* variable is a time interval, and you can use the methods of the TimeSpan class to express this interval in various units: `duration.MilliSeconds`, `Duration.Seconds`, and so on.

Chapter 14: Storing Data in Collections

Make the most of arrays. The simplest method of storing sets of data is to use arrays. They're very efficient and they provide methods to perform advanced operations such as sorting and searching their elements. Use the `Sort` method of the Array class to sort an array's elements. To search for an element in an array, use the `IndexOf` and `LastIndexOf` methods, or the `BinarySearch` method if the array is sorted. The `BinarySearch` method always returns an element's index, which is a positive value for *exact matches* and a negative value for *near matches*.

Master It Explain how you can search an array and find exact and near matches.

Solution The most efficient method of searching arrays is the `BinarySearch` method, which requires that the array is sorted. The simplest form of the `BinarySearch` method is the following:

```
Dim idx As Integer
idx = System.Array.BinarySearch(arrayName, object)
```

The `BinarySearch` method returns an integer value, which is the index of the object you're searching for in the array. If the *`object`* argument is not found, the method returns a negative value, which is the negative of the index of the next larger item minus one. The following statements return an exact or near match for the word `srchWord` in the `words` array:

```
Dim wordIndex As Integer = _
           Array.BinarySearch(words, srchWord)
If wordIndex >= 0 Then  ' exact match!
    MsgBox("An exact match was found for " & _
           " at index " & wordIndex.ToString)
Else                   ' Near match
    MsgBox("The nearest match is the word " & _
           words(-wordIndex - 1) & _
           " at " & (-wordIndex - 1).ToString)
End If
```

Store data in specialized collections such as ArrayLists and HashTables. In addition to arrays, the Framework provides collections, which are dynamic data structures. The most commonly used collections are the ArrayList and the HashTable. ArrayLists are similar to arrays, but they're dynamic structures. ArrayLists store lists of items, whereas HashTables store key-value pairs and allow you to access their elements via a key. You can add elements by using the `Add` method and remove existing elements by using the `Remove` and `RemoveAt` methods.

HashTables provide the `ContainsKey` and `ContainsValue` methods to find out whether the collection already contains a specific key or value, and the `GetKeys` and `GetValues` methods to retrieve all the keys and values from the collection, respectively.

Master It How will you populate a HashTable with a few pairs of keys/values and then iterate though the collection's items?

Solution To populate the HashTable, call its `Add` method, passing as an argument the item's key and value:

```
Dim HTable As New HashTable
HTable.Add("key1", item1)
HTable.Add("key2", item2)
```

To iterate through the items of a HashTable collection, you must first extract all the keys and then use them to access the collection's elements. The following code segment prints the keys and values in the `HTable` variable:

```
Dim element, key As Object
For Each key In HTable.keys
    element = HTable.Item(key)
    Debug.WriteLine("Item type = " element.GetType.ToString
    Debug.WriteLine("Key= " & Key.ToString)
    Denug.WritrLine("Value= " & element.ToString)
  Next
```

Sort and search collections. Collections provide the `Sort` method for sorting their items and several methods to locate items: `IndexOf`, `LastIndexOf`, and `BinarySearch`. Both sort and search operations are based on comparisons, and the Framework knows how to compare values types only (Integers, Strings, and the other primitive data types). If a collection contains objects, you must provide a custom function that knows how to compare two objects of the same type.

Master It How do you specify a custom comparer function for a collection that contains Rectangle objects?

Solution First you must decide how to compare two Rectangle objects. Let's consider two rectangles equal if their perimeters are equal. To implement a custom comparer, you must write a class that implements the IComparer interface:

```
Class RectangleComparer : Implements IComparer
   Public Function Compare( _
       ByVal o1 As Object, ByVal o2 As Object) _
       As Integer Implements IComparer.Compare
      Dim R1, R2 As Rectangle
      Try
         R1 = CType(o1, Rectangle)
         R2 = CType(o2, Rectangle)
      Catch compareException As system.Exception
         Throw (compareException)
         Exit Function
      End Try
     Dim perim1 As Integer = _
                2 * (R1.Width+R1.Height)
     Dim perim2 As Integer = _
                2 * (R2.Width+R2.Height)
      If perim1 < perim2 Then
         Return -1
      Else
         If perim1 > perim2 Then
            Return 1
         Else
            Return 0
         End If
      End If
   End Function
End Class
```

The following statement sorts the items of the `Rects` ArrayList collection, assuming that it contains only Rectangles:

```
Rects.Sort(New RectangleComparer)
```

To call the `BinarySearch` method for the same ArrayList, use the following statement:

```
Rects.BinarySearch( _
      New Rectangle(0, 0, 33, 33), comparer)
```

Chapter 15: Accessing Folders and Files

Handle Files with the My object. The simplest method of saving data to a file is to call one of the `WriteAllBytes` or `WriteAllText` methods of the My.Computer.FileSystem object. You can also use the IO namespace to set up a Writer object to send data to a file, and a Reader object to read data from the file.

Master It Show the statements that save a TextBox control's contents to a file and the statements that reload the same control from the data file. Use the My.Computer.FileSystem component.

Solution The following statement saves the control's `Text` property to a file whose path is stored in the *`filename`* variable. Prompt users with the Open dialog box control for the path of the file and use it in your code.

```
My.Computer.FileSystem.WriteAllText( _
       fileName, TextBox1.Text, True)
```

To read the data back and place it in the *`TextBox1`* control again, use the following statement:

```
TextBox1.Text = My.Computer.FileSystem.ReadAllText(fileName)
```

Write data to a file with the IO namespace To send data to a file you must set up a FileStream object, which is a channel between the application and the file. To send data to a file, create a StreamWriter or BinaryWriter object on the appropriate FileStream object. Likewise, to read from a file, create a StreamReader or BinaryReader on the appropriate FileStream object. To send data to a file, use the `Write` and `WriteString` methods of the appropriate StreamWriter object. To read data from the file, use the `Read`, `ReadBlock`, `ReadLine`, and `ReadToEnd` methods of the StreamReader object.

Master It Write the contents of a TextBox control to a file using the methods of the IO namespace.

Solution Begin by setting up a FileStream object to connect your application to a data file. Then create a StreamWriter object on top of the FileStream object and use the `Write` method to send data to the file:

```
Dim FS As FileStream
FS = New FileStream(fileName, FileMode.Create)
Dim SW As StreamWriter(FS)
SW.Write(TextBox1.Text)
SW.Close
FS.Close
```

To read the data back and reload the TextBox control, set up an identical FileStream object, then create a StreamReader object on top of it, and finally call the `ReadToEnd` method:

```
Dim FS As New FileStream(fileName, _
        System.IO.FileMode.OpenOrCreate, _
        System.IO.FileAccess.Write)
Dim SR As New StreamReader(FS)
```

```
TextBox1.Text = SR.ReadToEnd()
FS.Close
SR.Close
```

Manipulate folders and files. The IO namespace provides the Directory and File classes, which represent the corresponding entities. Both classes expose a large number of methods for manipulating folders (`CreateDirectory`, `Delete`, `GetFiles`, and so on) and files (`Create`, `Delete`, `Copy`, `OpenRead`, and so on).

Master It How will you retrieve the attributes of a drive, folder, and file using the IO namespace's classes?

Solution Start by creating DriveInfo, DirectoryInfo, and FileInfo files. Specify the path of the corresponding entity in the class's constructor:

```
Dim DrvInfo As New DriveInfo("C:\")
Dim DirInfo As New DirectoryInfo( _
       "C:\Program Files")
Dim FInfo As New FileInfo( _
      "C:\Program Files\My Apps\Readme.txt")
```

Then enter the name of one of these objects, followed by a period, and select the appropriate property from the IntelliSense list.

The available space on drive `C:` is given by the property `DrvInfo.AvailableFreeSpace`, and its type is given by the property `DrvInfo.DriveType.ToString`. The attributes of the folder Program Files are given by the property `DirInfo.Attributes`. The size of the `Readme.txt` file in the same folder is given by the property `FInfo.Length`.

The `DrvInfo`, `DirInfo`, and `FInfo` objects also expose methods for manipulating the corresponding entities. The method `GetDrives` of the DriveInfo class returns the names of all drives on the target computer. To manipulate a folder, use one of the `Delete`, `Create`, `MoveTo`, or other methods of the DirInfo class. Finally, to manipulate a file, use one of the `Delete`, `Encrypt`, `Decrypt`, `Open`, or other methods of the FInfo class.

Monitor changes in the file system and react to them. The FileSystemWatcher is a special component that allows your application to monitor changes in the file system. You can specify the types of changes you want to monitor by using the `NotifyFilter` property, the types of files you want to monitor by using the `Filter` property, and the path you want to monitor by using the `Path` property. The FileSystemWatcher component fires the `Changed`, `Created`, `Deleted`, and `Renamed` events, depending on the type of change(s) you specified. Once activated, the FileSystemWatcher component fires an event every time one of the specified items changes.

Master It Assume that an application running on a remote computer creates a file in the `E:\Downloaded\Orders` folder for each new order. How will you set up a FileSystemWatcher component to monitor this folder and notify your application about the arrival of each new order?

Solution First, drop an instance of the FileSystemWatcher component on your form and insert the following statements in the form's `Load` event handler to set up and activate the FileSystemWatcher component:

```
FileSystemWatcher1.Path = "E:\Downloaded\Orders"
FileSystemWatcher1.IncludeSubdirectories = False
```

```
FileSystemWatcher1.Filter = "*.txt"
FileSystemWatcher1.NotifyFilter = _
            IO.NotifyFilters.CreationTime
FileSystemWatcher1.EnableRaisingEvents = True
```

Then enter some code in the FileSystemWatcher component's `Created` event handler:

```
Private Sub WatcherHandler( _
           ByVal sender As Object, _
           ByVal e As System.IO.FileSystemEventArgs) _
           Handles FileSystemWatcher1.Created, _
    MsgBox("File " & e.FullPath & " arrived!"
End  Sub
```

You can use any of the methods of the My.Computer.FileSystem or the IO namespace to handle the new file, which is given by the property `FullPath` of the handler's `File-SystemEventArgs` argument.

Chapter 16: Serialization and XML

Serialize objects and collections into byte streams. Serialization is the process of converting an object into a stream of bytes. This process (affectionately known as dehydration) generates a stream of bytes or characters, which can be stored or transported. To serialize an object, you can use the BinaryFormatter or SoapFormatter class. You can also use the XmlSerializer class to convert objects into XML documents. All three classes expose a Serialize class that accepts as arguments the object to be serialized and a stream object, and writes the serialized version of the object to the specified stream.

Master It Describe the process of serializing an object with a binary or SOAP formatter.

Solution To serialize an object, you must first create a Stream object that will accept the result of the serialization. This Stream object is usually associated with a file. You also need an instance of the BinaryFormatter or the SoapFormatter class. Both classes expose the `Serialize` method, which accepts as arguments the Stream object you created already and the object to be serialized.

Deserialize streams to reconstruct the original objects. The opposite of serialization is called deserialization. To reconstruct the original object, you use the `Deserialize` method of the same class you used to serialize the object.

Master It Describe the process of serializing an object with a binary or SOAP formatter.

Solution For the reverse process, you create another Stream object that represents the source of the serialized data, create an instance of the appropriate serialization class (depending on the type of serialization), and call the `Deserialize` method. The `Deserial-ize` method accepts a single argument, which is the Stream object from which the method will read the serialized data. The result of the `Deserialize` method is an object, which you must cast to the appropriate type.

Create XML files in your code. XML is a standard for storing data. In addition to the data, an XML document also describes the structure of its contents by using elements and attributes.

Elements represent entities and their properties. Attributes represent the properties of the elements to which they're applied.

Master It How would you create an XML document to describe structured data?

Solution The simplest method to create an XML document is to design a class that represents the entities you want to store in the document. Then populate a few instances of the class and serialize them with the XmlSerializer class. The output of the serialization is a valid XML document, which you can open in Visual Studio. You can also create a schema for this document, so that you can edit it and be sure that it complies with its schema.

Chapter 17: Querying Collections and XML with LINQ

Perform simple LINQ queries. A LINQ query starts with the structure `From` *`variable`* `In` *`collection`*, where *`variable`* is a variable name and *`collection`* is any collection that implements the `IEnumerable` interface (such as an array, a typed collection, or any method that returns a collection of items). The second mandatory part of the query is the `Select` part, which determines the properties of the variable we want in the output. Quite often we select the same variable that we specify in the `From` keyword. In most cases, we apply a filtering expression with the `Where` keyword. Here's a typical LINQ query that selects filenames from a specific folder:

```
Dim files = _
        From file In _
          IO.Directory.GetFiles("C:\Documents") _
          Where file.EndsWith("doc") _
      Select file
```

Master It Write a LINQ query that calculates the sum of the squares of the values in an array.

Solution To calculate a custom aggregate in a LINQ query, you must create a lambda expression that performs the aggregation and pass it as an argument to the `Aggregate` method. The lambda expression accepts two arguments — the running value of the aggregate and the current element — and returns the new aggregate. Such a function would have the following signature and implementation:

```
Function(aggregate, value)
    Return(aggregate + value ^2)
End Function
```

To specify this function as a lambda expression in a LINQ query, call the collection's `Aggregate` method as follows:

```
Dim sumSquares = data.Aggregate( _
    Function(sumSquare As Long, n As Integer) _
           sumSquare + n ^ 2
```

Create and process XML files with LINQ to XML. LINQ to SQL allows you to create XML documents with the XElement and XAttribute classes. You simply create a new XElement object for each element in your document, and a new XAttribute object for each attribute in the current element. Alternatively, you can simply insert XML code in your VB code. To create an XML document dynamically, you can insert embedded expressions that will be evaluated by the compiler and replaced with their results.

Master It How would you create an HTML document with the filenames in a specific folder?

Solution To generate a directory listing, we must first implement the LINQ query that retrieves the desired information. The query selects the files returned by the `GetFiles` method of the IO.Directory class:

```
Dim files = From file In _
                IO.Directory.GetFiles(path) _
                Select New IO.FileInfo(file).Name, _
                New IO.FileInfo(file).Length
```

Now we must embed this query into an XML document by using expression holes. The XML document is actually an HTML page that displays a table with two columns, the file's name and size, as shown next:

```
Dim smallFilesHTML = <html>
       <table><tr>
          <td>FileName</td>
          <td>FileSize</td></tr>
          <%= From file In _
              IO.Directory.GetFiles("C:\") _
              Select <tr><td><%= file %></td> ,
              <td>
              <%= New IO.FileInfo(file).Length %>
              </td></tr>
           %>
       </table></html>
```

Process relational data with LINQ to SQL. LINQ to SQL allows you to query relational data from a database. To access the database, you must first create a DataContext object. Then you can call this object's `GetTable` method to retrieve a table's rows, or the `ExecuteQuery` method to retrieve selected rows from one or more tables with an SQL query. The result is stored in a class designed specifically for the data you're retrieving via the DataContext object.

Master It Explain the attributes you must use in designing a class for storing a table.

Solution The class must be decorated with the `<Table>` attribute, which specifies the name of the table that will populate the class:

```
<Table(Name:="Customers">Public Class Customers
...
End  Class
```

Each property of this table must be decorated with the <Column> attribute, which specifies the name of the column from which the property will get its value:

```
<Column(Name:="CompanyName")> _
     Public Property Company
...
End Property
```

When you call the GetTable method of the DataContext object, pass the name of the class as an argument, and the DataContext object will create a new instance of the class and populate it.

Chapter 18: Drawing and Painting with Visual Basic 2008

Display and size images. The most appropriate control for displaying images is the PictureBox control. You can assign an image to the control through its Image property, either at design time or at runtime. To display a user-supplied image at runtime, call the DrawImage method of the control's Graphics object.

Master It How would you implement a form that displays a large image and allows users to scroll the image to bring any segment of it into view?

Solution

1. Add a new form to your project.
2. Place a Panel control on the form and set its AutoSize and AutoScroll properties to True.
3. Place a PictureBox control on the Panel control and set its SizeMode property to *AutoSize*.
4. Finally, assign a large image to the PictureBox control. As soon as you assign the image to the control, the necessary scroll bars will be displayed and you can scroll any part of the image into view, even at design time.

Generate graphics by using the drawing methods. Every object you draw on, such as forms and PictureBox controls, exposes the CreateGraphics method, which returns a Graphics object. The Paint event's e argument also exposes the Graphics object of the control or form. To draw something on a control, retrieve its Graphics object and then call the Graphics object's drawing methods.

Master It Show how to draw a circle on a form from within the form's Paint event handler.

Solution The following statements will draw a circle at the center of the *Form1* form:

```
Private Sub Form1_Paint( _
       ByVal sender As Object, _
       ByVal e As System.Windows.Forms.PaintEventArgs) _
       Handles Me.Paint
```

```
        Dim diameter = Math.Min(Me.Width, Me.Height) / 2
        e.Graphics.DrawEllipse(Pens.Blue, _
          New RectangleF((Me.Width - diameter) / 2, _
         (Me.Height - diameter) / 2, _
         diameter, diameter))
    End Sub
```

The circle is being redrawn as you resize the form. To force a redraw when the form is redrawn, insert the following statement in the form's `Load` event handler:

```
    Me.SetStyle(ControlStyles.ResizeRedraw,True)
```

Display text in various ways, including gradient fills. The Graphics object provides the `DrawString` method, which prints a user-supplied string on a control. You can also specify the coordinates of the string's upper-left corner and its font. To position the string, you need to know its dimensions. You can use the `MeasureString` method to retrieve the dimensions of the image when rendered on the Graphics object in a specific font. Text is drawn with a Brush object, and you can use a SolidBrush object to draw the string in a solid color, the Linear-GradientBrush object to fill the text with a linear gradient, the PathGradientBrush object to fill the text with an arbitrary gradient defined by a path, or the TextureBush object to fill the text with a texture.

Master It How will you print a string centered on a PictureBox control?

Solution To determine the coordinates of the string on the PictureBox control, you must first find out the dimensions of the string by using the `MeasureString` graphics method:

```
    Dim str As String = "Print this!"
    Dim fnt As New Font("Verdana", 48, FontStyle.Bold)
    Dim strSize As SizeF = _
             e.Graphics.MeasureString(str, fnt)
```

To actually draw the string, call the `DrawString` method as follows from within the control's `Print` event handler:

```
    Dim strPoint As New PointF( _
         (PictureBox1.Width - strSize.Width) / 2, _
         (PictureBox1.Height - strSize.Height) / 2) _
         e.Graphics.DrawString(str, fnt, _
         Brushes.DarkGreen, strPoint)
```

The first statement calculates the coordinates of the string's upper-left corner by splitting the difference between the control's width/height and the string's width/height on either size of the string.

If you anchor the PictureBox control on all four edges of the form, the text isn't being redrawn. There's no `SetStyle` method for the PictureBox control, but you can call the `SetStyle` method as shown in the preceding section and then call the PictureBox control's `Refresh` method from within the form's `Paint` event handler:

```
    Private Sub Form1_Load(...) Handles MyBase.Load
        Me.SetStyle(ControlStyles.ResizeRedraw, True)
    End Sub
```

```
Private Sub Form1_Paint(...) Handles Me.Paint
    PictureBox1.Refresh()
End Sub
```

Chapter 19: Manipulating Images and Bitmaps

Specify colors. Color values are based on the RGB cube. Each color is a point in the RGB code and is expressed as a triplet of integer values that represent the intensity of the red, green, and blue components of the color. You can also use the named colors of the Color class.

Master It How do you draw with semitransparent colors?

Solution You can call the `FromARGB` method with four arguments, which are a transparency value (alpha channel) and the usual red, green, and blue components of a color. If the transparency value is 255, the color specified with the other three components is opaque. If the transparency value is 0, the color is totally transparent. By specifying any other value between the two extremes, you can superimpose graphics that allow the underlying visual elements to show through. Use this technique to draw 3D-looking text, place watermarks on images, or wash out the colors of a background image. You can also animate an image's colors by varying the alpha channel from totally transparent to totally opaque from within a loop.

Manipulate images and bitmaps. The Framework provides two classes for representing images: the Image and Bitmap classes. The Image class represents images. You use Image objects to read images from files or streams and to store them in memory. You can't use the Image object to create a new image. The Bitmap class also represents images, but you can use the Bitmap object to create new images from within your code.

Master It When will you use an Image object versus a Bitmap object in a graphics application?

Solution Images are stored in Image objects, which are static. You can't edit the contents of an Image object; you can only display it on a Form or PictureBox control and find out the image's attributes (its resolution, size, and so on). The Bitmap object allows you to read and set the image's pixel values, or create new images from within your code, with the `GetPixel` and `SetPixel` methods. Use an Image object to store an image. Use a Bitmap object to store and edit an image, or to create a new image entirely from within your application's code.

Process images. Images are two-dimensional arrays of color values, one value per pixel, arranged in rows and columns. To process an image's pixels, start by reading the image into a Bitmap object. Then set up two nested loops that iterate through each row and each column of pixels. Use the `GetPixel` method to read pixel values, and the `SetPixel` method to change a pixel's value.

Master It Outline the code that processes the pixels of an image.

Solution First you must create a Bitmap object by loading the pixel values of the image:

```
Dim Bmp As Bitmap
Bmp = Image.FromFile(FileName)
```

Then create a new Bitmap object with the same dimensions as the original image, where you'll store the processed pixels. If you overwrite the pixel values of the original Bitmap object, you won't be able to read the original values anymore.

```
Dim newBmp As New Bitmap( _
      Bmp.Width, Bmp.Height, Bmp.PixelFormat)
```

Now you're ready to set up two nested loops to iterate through the image's pixels and process them. In the following sample, the code flips the image vertically and horizontally:

```
Dim row, col As Integer
Dim clr As Color
Dim newBmp As New Bitmap( _
          Bmp.Width, Bmp.Height, Bmp.PixelFormat)
    For row = 0 To Bmp.Width - 1
        For col = 0 To Bmp.Height - 1
            clr = Bmp.GetPixel(row, col)
            newBmp.SetPixel(Bmp.Width - row - 1, _
            Bmp.Height - col - 1, clr)
        Next
Next
```

To view the processed image, assign the *newBmp* variable to the `Image` property of a PictureBox control.

Chapter 20: Printing with Visual Basic 2008

Use the printing controls and dialog boxes. To print with the .NET Framework, you must add an instance of the PrintDocument control to your form and call its `Print` method. To preview the same document, you simply assign the PrintDocument object to the `Document` property of the PrintPreviewDialog control and then call the `ShowDialog` method of the PrintPreviewDialog control to display the preview window. You can also display the Print dialog box, where users can select the printer to which the output will be sent, and the Page Setup dialog box, where users can specify the page's orientation and margins. The two dialog boxes are implemented with the PrintDialog and PageSetupDialog controls.

Master It Explain the process of generating a simple printout. How will you handle multiple report pages?

Solution Both the `PrintDocument.Print` and the `PrintPreviewDialog.ShowDialog` methods fire the `PrintPage` event of the PrintDocument object. The code that generates the actual printout must be placed in the `PrintPage` event's handler, and the same code will generate the preview or the actual printout.

It's your responsibility to terminate each page and start a new one every time you complete the current page — you simply exit the `PrintPage` event handler. If there are more pages to print, set the `HasMorePages` property to True. Any static variables you use to maintain state between successive invocations of the PrintPage event handler, such as the page number, must be reset every time you start a new printout. A good place to initialize the printout's variables is with the BeginPrint event handler.

Master It Assuming that you have displayed the Page Setup dialog box control to the user, how will you draw a rectangle that delimits the printing area on the page, taking into consideration the user-specified margins?

Solution First, set up a few variables to store the page's margins:

```
Dim Lmargin, Rmargin, Tmargin, Bmargin As Integer
With PrintDocument1.DefaultPageSettings.Margins
   Lmargin = .Left: Rmargin = .Right
   Tmargin = .Top: Bmargin = .Bottom
End  With
```

Then calculate the dimensions of the rectangle, where the printing will be confined:

```
Dim PrintWidth, PrintHeight As Integer
With PrintDocument1.DefaultPageSettings.PaperSize
   PrintWidth = .Width - Lmargin - Rmargin
   PrintHeight = .Height - Tmargin - Bmargin
End  With
```

The rectangle we want to draw should start at the point (*Lmargin*, *Tmargin*) and extend *PrintWidth* units to the right and *PrintHeight* units down.
Finally, insert the following statements in the PrintPage event handler to draw the rectangle:

```
Dim R As Rectangle
R = New Rectangle(Lmargin, Tmargin, PrintWidth, PrintHeight)
e.Graphics.DrawRectangle(Pens.Black, R)
```

Print plain text and images. Typical business applications generate printouts with text and a few borders or grids. The DrawString method of the Graphics object can print a string at a specific location on the page. To print images, call the DrawImage method of the Graphics object, passing as an argument the image you want to print and the rectangle on the page where you want the image to appear.

Master It Outline the process of printing the contents of a TextBox control.

Solution An overloaded form of the DrawString method allows you to specify the rectangle in which the string will be printed and let the method break the text into multiple lines that fit in that rectangle. You can also specify how the string will be broken into multiple lines to fit in the given rectangle. Before calling the DrawString method, however, you must call the MeasureString method to find out the number of characters that will fit in this rectangle and pass a segment of the string to the DrawString method. Then you can call the DrawString method with the appropriate chunk of the text for the current page.

Print tabular data. Business applications make heavy use of reports, and you should provide a mechanism to print these out. Printing tabular data isn't a simple task, but after you break the page into rows and columns, you can draw the appropriate string into its corresponding cell.

Master It Describe the process of building a tabular report.

Solution You must first come up with the widths of the columns and then decide whether the rows will have a fixed or variable height. For fixed-height rows, you should call the `DrawString` method to print the appropriate string (or part of it) into its cell. You also know the number of rows that will fit on the page, and this will simplify the logic that detects the end of the page. For variable-height cells, you must call the `MeasureString` method to find out the height of each cell (how many lines it takes to print the appropriate string into a cell of a given width). You must keep track of the tallest cell and then use this value to advance to the following row.

Chapter 21: Basic Concepts of Relational Databases

Use relational databases. Relational databases store their data in tables and are based on relationships between these tables. The data is stored in tables, and tables contain related data, or entities, such as persons, products, orders, and so on. Relationships are implemented by inserting columns with matching values in the two related tables.

Master It How will you relate two tables with a many-to-many relationship?

Solution A many-to-many relationship can't be implemented with primary/foreign keys between two tables. To create a many-to-many relationship, you must create a table between the other two tables and implement two one-to-many relationships. Consider the Titles and Authors tables, which have a many-to-many relationship, because a title can have many authors and the same author may have written multiple titles. To implement this relationship, you must create an intermediate table, the TitleAuthors table, which is related to both the Titles and Authors tables with a one-to-many relationship. The TitleAuthors table should store title and author IDs. The *`TitleAuthor.TitleID`* field is the foreign key to the relationship between the Titles and TitleAuthor tables. Likewise, the *`TitleAuthor.AuthorID`* field is the foreign key to the relationship between the TitleAuthor and Authors tables.

Utilize the data tools of Visual Studio. Visual Studio 2008 provides visual tools for working with databases. The Server Explorer is a visual representation of the databases you can access from your computer and their data. You can create new databases, edit existing ones, and manipulate their data. You can also create queries and test them right in the IDE.

Master It Describe the process of establishing a new relationship between two tables.

Solution To create a relationship, double-click a table's name in Server Explorer to open it in design mode and then choose Table Designer ➢ Relationships, which will display the Foreign Key Relationships dialog box shown in Figure 21.9. To create a new relationship between the selected table and another one, click the Add button. A new relationship will be added with a default name, which you can change. Expand the Tables And Columns Specification entry and click the ellipses in this field to see a dialog box, where you can select the name of the table with the primary key in the relationship in the Primary Key Table column.

Use the Structured Query Language for accessing tables. Structured Query Language (SQL) is a universal language for manipulating tables. SQL is a nonprocedural language, which specifies the operation you want to perform against a database at a high level, unlike traditional languages such as Visual Basic, which specifies how to perform the operation. The details of the implementation are left to the DBMS. SQL consists of a small number of keywords and is optimized for selecting, inserting, updating, and deleting data.

Master It How would you write a `SELECT` statement to retrieve data from multiple tables?

Solution The `FROM` statement should include the names of two or more tables, which must be somehow related. To relate the tables, use the `JOIN` clause and specify the primary/foreign keys of the join:

```
SELECT column1, column2, ...
FROM table1 T1 INNER JOIN table2
 ON T1.primaryKey = T2.foreignKey
  INNER JOIN table3 T3
 ON T2.primaryKey = T3.foreignKey
```

Pay attention to the type of join you specify. An inner join requires that the two columns match and excludes Null values. A left join takes into consideration all the qualifying rows of the left table, including the ones that have Nulls in their foreign key column. A right join takes into consideration all the qualifying rows of the right table, including the ones that have Nulls in their foreign key column. A full outer join is a combination of the right and left joins — it takes into consideration Null values from both tables involved in the query.

Chapter 22: Programming with ADO.NET

Create and populate DataSets. DataSets are data containers that reside at the client and are populated with database data. The DataSet is made up of DataTables, which correspond to database tables, and you can establish relationships between DataTables, just like relating tables in the database. DataTables, in turn, are made up of DataRow objects.

Master It How do we populate DataSets and then submit the changes made at the client back to the database?

Solution To populate a DataSet, you must create a DataAdapter object for each DataTable in the DataSet. The DataAdapter class provides the `SelectCommand`, `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties, which are initialized to the SQL statements that retrieve/insert/update/delete rows from the corresponding database tables. You can use the CommandBuilder object to build the `UPDATE/INSERT/DELETE` commands from the `SELECT` command. After these properties are in place, you can populate the corresponding DataTable in the DataSet by calling the `FILL` method of its DataAdapter object. If the DataSet contains relationships, you must fill the parent tables before the child tables.

Establish relations between tables in the DataSet. The DataSet can be thought of as a small database that resides at the client, because it's made up of tables and relationships between them. The relations in a DataSet are DataRelation objects, which are stored in the `Relations` property of the DataSet. Each relation is identified by a name, the two tables it relates, and the fields of the tables on which the relation is based.

Master It How do we navigate through the related rows of two tables?

Solution To navigate through related tables, the DataRow object provides the `GetChildRows` method, which returns the current row's child rows as an array of DataRow objects, and the `GetParentRow/GetParentRows` methods, which return the current row's parent row(s). `GetParentRow` returns a single DataRow object, and `GetParentRows` returns an array of DataRow objects. Because a DataTable may be related to multiple DataTables, you must also specify the name of the relation. The following statements retrieve the child rows of a specific category and a specific supplier, respectively:

```
DS.Categories(iRow).GetChildRows("CategoriesProducts")
DS.Categories(iRow).GetChildRows("SuppliersProducts")
```

Submit changes in the DataSet back to the database. The DataSet maintains not only data at the client, but their states and versions too. It knows which rows were added, deleted, or modified (the `DataRowState` property) and it also knows the version of each row read from the database and the current version (the `DataRowVersion` property).

Master It How will you submit the changes made to a disconnected DataSet to the database?

Solution To submit the changes made to an untyped DataSet, you must call the `Update` method of each DataAdapter object. You must call the `Update` method of the DataAdapter objects that correspond to the parent tables first and then the `Update` method of the DataAdapters that correspond to the child tables. You can also submit individual rows to the database, as well as arrays of DataRow objects through the overloaded forms of the `Update` method.

Chapter 23: Building Data-Bound Applications

Design and use typed DataSets. Typed DataSets are created with visual tools at design time and allow you to write type-safe code. A typed DataSet is a class created by the wizard on the fly and it becomes part of the project. The advantage of typed DataSets is that they expose functionality specific to the selected tables and can be easily bound to Windows forms. The code that implements a typed DataSet adds methods and properties to a generic DataSet, so all the functionality of the DataSet object is included in the autogenerated class.

Master It Describe the basic components generated by the wizard when you create a typed DataSet with the visual tools of Visual Studio.

Solution The basic components of the class that implements a typed DataSet are as follows: the DataSet, which describes the entire DataSet; the BindingNavigator, which links the data-bound controls on the form to the DataSet; and the TableAdapter, which links the DataSet to the database. The DataSet component is based on the DataSet class and enhances the functionality of an untyped DataSet by adding members that are specific to the data contained in the DataSet. If the DataSet contains the Products table, the typed DataSet exposes the ProductsRow class, which represents a row of the Products table. The ProductsRow class, in turn, exposes the columns of the table as properties.

The BindingSource class allows you to retrieve the current row with its `Current` property, move to a specific row by setting the `Position` property, even suspend temporarily and restore data binding with `SuspendBinding` and `ResumeBinding`.

The TableAdapter class, which is based on the DataAdapter class, provides methods for loading a DataTable (the `Fill` method) and submitting the changes made to the DataSet to the database (the `Update` method). The TableAdapterManager class, which encapsulates the functionality of all TableAdapters on the form, provides the `UpdateAll` method, which submits the changes in all DataTables to the database.

Bind Windows forms to typed DataSets. The simplest method of designing a data-bound form is to drop a DataTable, or individual columns, on the form. DataTables are bound to DataGridView controls, which display the entire DataTable. Individual columns are bound to simple controls such as TextBox, CheckBox, and DateTimePicker controls, depending on the column's type. In addition to the data-bound controls, the editor generates a toolbar control with some basic navigational tools and the Add/Delete/Save buttons.

Master It Outline the process of binding DataTables to a DataGridView control.

Solution To bind a DataTable to a DataGridView control, locate the desired table in the Data Sources window, set its binding option to DataGridView, and drop the DataTable on the form. The editor will create a DataGridView control on the form and map the control's columns according to the columns of the DataTable. It will also add a toolbar with the basic navigational and editing controls on the form.

To bind two related DataTables on the same form, drop the parent DataTable on the form, and then select the child DataTable under this parent and drop it on the form. To modify the appearance of the DataGridView controls, open their Tasks menu and choose Edit Columns to see the Edit Columns dialog box, where you can set the appearance of the control's columns.

Chapter 24: Advanced DataSet Operations

Use SQL to query DataSets. Although DataSets resemble small databases that reside in the client computer's memory, you can't manipulate them with SQL statements. However, it's possible to query their tables by using the `Select` method and SQL-like criteria. The `Select` method filters the rows of the table to which it's applied and returns an array of DataRow objects, which you can use as a data source for data-bound controls.

Master It How would you select the rows of interest from a DataTable?

Solution To filter the rows of a DataTable, use the `Select` method. The simplest form of this method accepts an SQL expression as an argument and uses it to filter the rows of the DataTable to which it's applied. To select the customers from Germany, apply the following `Select` method to the Customers DataTable:

```
DS.Customers.Select("Country = 'Germany')
```

The second, optional, argument of the `Select` method is equivalent to the `Order By` clause of SQL and determines the order in which the selected rows will appear. In addition to the SQL expression, you can limit the scope of the selection by specifying the state of the rows you want to select in the DataSet. This form of the Select method accepts a third argument, which is of the RowState type (Added, Deleted, Modified).

Add calculated columns to DataTables. Sometimes you'll need to update a column's value based on the values of other columns. For example, you may wish to maintain a column in the Orders DataTable with the order's total. For this column to be meaningful, its value should be updated every time a related row in the Order Details DataTable is modified. In the actual database, you'd do this with a trigger, but we want to avoid adding too many triggers to our tables because they slow all data-access operations, not to mention that you'll be duplicating information. To maintain totals in a DataSet, you can add calculated columns to its DataTables. You specify a formula for the calculated column, and every time one of the columns involved in the formula changes, the calculated column's value changes accordingly.

Master It Add a calculated column to the Customers DataTable that combines the first and last columns.

Solution To add a new DataColumn object to a DataTable, use the Add method of the table's Columns collection. One of the overloaded forms of this method allows you to specify not only the column's name and data type, but also its contents. The following statement adds the Name column to the Customers table and sets its type to String and its value to the specified expression shown in this statement:

```
DS.Orders.Columns.Add("Name", _
        System.Type.GetType("System.String"), _
       "First & ' ' & Last")
```

The expression that specifies the new column's value isn't calculated when the DataSet is loaded; it's calculated on-the-fly, every time the column is requested.

Compute aggregates over sets of rows. To calculate aggregates over sets of rows in a DataTable, use the Compute method, which accepts two arguments: an SQL-like aggregate function and a filtering expression, which is similar to an SQL WHERE clause. The aggregate function isn't limited to columns of the table to which it's applied; you can use the Child()function to access the current row's child row in a given relation, and the Parent() function to access the current row's parent row(s) in a given relation, which is passed to the method as an argument.

Master It Show the statement for adding a new column to the Orders table with each order's total.

Solution Each order's total is the sum of the products of the quantities times the prices for all lines in the order. (You must also take into consideration the discount.) The implication is that the column belongs to the Orders table, but it must be calculated over the current order's child rows in the Order Details table. To access the detail lines of an order, use the Child function, passing as an argument the name of the relation, as shown here:

```
DS.Orders.Columns.Add("OrderTotal", _
    System.Type.GetType("System.Decimal"), _
    SUM(child(Orders_Order_Details).UnitPrice * _
    child(Orders_Order_Details).Quantity * _
   (1 - child(Orders_Order_Details).Discount)")
```

`Orders_Order_Details` is the name of the relation between the Orders and Order Details tables.

Chapter 25: Building Web Applications

Create a basic XHTML/HTML page. Building a basic HTML page is a straightforward process using a simple text editor such as Notepad. Knowledge of XHTML/HTML is still a major asset when developing web applications with Visual Studio 2008.

Master It Develop a web page using HTML that features a heading, some text, an image, and a link to another page. Convert the page to XHTML and verify it by using the W3C verification tool at `http://validator.w3.org`. You might find that you will need to run the validation a couple of times to get everything right. If you attach and use the style sheet in the following Master It challenge, you will find that the validation will be less problematic.

Solution Note that this solution includes the style sheet created in the next Master It challenge. Some long lines are wrapped here in print, but you can leave them all on one line in your code.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 _
 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1- _
strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en">
   <head>
      <link rel="stylesheet" type="text/css" href="stylesheet.css" />
      <title>Basic Page</title>
   </head>
   <body>
      <div class="title">
         <h1>Heading</h1>
      </div>
      <div class="content">
         <p>Text</p>
         <img src="myimage.jpg" height="100" _
 width="100" alt="myimage" />
         <br/>
      </div>
      <div class="menu">
         <a href="http://www.microsoft.com">Microsoft</a>
      </div>
   </body>
</html>
```

Format a page with CSS. Cascading Style Sheets (CSS) are a powerful tool for controlling the styles and format of a website. You can manually create style sheets by using a text editor. An understanding of their operation and syntax is a useful skill when manipulating CSS in Visual Studio. 2008.

Master It Create a CSS style sheet that defines the layout of the web page that you developed in the previous task, including a header section, a left-hand navigation section, and a main content section. Include a rollover for the link and apply formatting to the tags that you have used for your heading and text tags. Attach the style sheet to your web page.

Solution The following code represents the CSS style sheet:

```
.title{
 height:80px;
 background:lightblue;
 margin:5px 10px 10px 10px;
 text-align: center;
 }

.menu{
 position: absolute;
 top: 110px;
 left: 20px;
 width: 130px;
 background: silver;
 padding: 10px;
 bottom: 20px;
 }

.content{
 background: lightblue;
 padding: 30px;
 position: absolute;
 top: 110px;
 bottom: 20px;
 left: 180px;
 right: 20px
 }

a {
 text-decoration:none;
 color:blue;
 }

a:visited {
 text-decoration:none;
 color:blue;
 }

a:hover {
 text-decoration:none;
 font-weight: bold;
 color:darkblue;
 }
```

```
a:active {
 text-decoration:none;
 color:blue;
 }
```

Set up a master page for your website. Using master pages is a reliable method of controlling the overall layout, and look and feel of your websites and applications. Master pages enable you to achieve a high level of consistency in your design and are particularly useful if the site has multiple developers working on it.

Master It Create a website with a master page and attached content page. Use appropriate controls to give the master page a page heading, My Page, which will appear on all attached content pages. Use a combination of Button and Label controls on the content page to create a simple Hello World application.

Solution Start a new website and delete the `default.aspx` page. Create a new master page and add a Label control to it above the default ContentPlaceHolder control. Format the Label control appropriately as a heading and add a page heading: My Page.

Add a content page to your project. Name the page `default.aspx`. In the content page, add a Button control and a Label control to the ContentPlaceHolder. (You may need to right-click the ContentPlaceHolder control and choose the Create Custom Content option.) Double-click the button and write the following code to set the text property of the label control to *Hello World*:

```
Protected Sub Button1_Click(ByVal sender As Object, _
 ByVal e As System.EventArgs) Handles Button1.Click
        Label1.Text = "Hello World"
End Sub
```

Use some of the ASP.NET intrinsic objects. ASP.NET objects such as the Response, Request, Session, and Server objects offer a range of important utilities in developing, running, and maintaining your websites and applications. In addition, they also give you access to vital information about the client, the server, and any current sessions at runtime.

Master It Create a simple website with a master page and two attached content pages. Use the `Server.Execute` method attached to a LinkButton control to navigate between the two content pages.

Solution Create a new website and delete the `default.aspx` page. Create a new master page and attach two content pages. Name one of the pages **`default.aspx`** and right-click it in Solution Explorer to set the page as the startup page from the drop-down context menu. Name the second page **`Page2.aspx`**. Place some distinguishing features on the two pages, such as Label controls with appropriate text content.

Add a LinkButton control to the ContentPlaceHolder in `default.aspx`. Double-click the LinkButton control and use `Server.Execute` in the sub for the `Click` method to create a link to page 2.

```
Protected Sub LinkButton1_Click(ByVal sender As Object, _
   ByVal e As System.EventArgs) Handles LinkButton1.Click
        Server.Execute("Page2.aspx")
End  Sub
```

Chapter 26: ASP.NET 3.5

Create cascading style sheets. Even though you may have created a main content area for your page, you will still need to lay out the various text items and images that compose the area. You can employ CSS to create content containers that can be used for layout and styles within the main style sheet areas defined for a page.

Master It Develop a text container using CSS that will occupy the full width of the containing area and expand vertically to accommodate the content. Include style attributes for the `<p>` tag that set the font to Verdana, the text color to dark blue, and the font size to 8 point. Set vertical alignment to top, and horizontal alignment to left.

Solution

```
.contenttextbox
{
   background: lightblue;
   clear:both;
   padding: 5px;
   text-align:left;
   vertical-align:top;
   border:0;
}
.contenttextbox p
{
   font:verdana;
   color:Navy;
   font-size:8pt;
}
```

Use web form controls. Images are an important part of any web page. The Image control from the Standard toolbox gives you a control that combines the ability to display an image with server-side functionality.

Master It Place an Image control onto an ASPX page and use it to display an image.

Solution Prepare a suitable image in GIF, JPG, or PNG format. Open the root directory of your website and create a new subdirectory named **images**. Place the image that you have created inside the images directory.

From the Standard toolbox, drag an instance of the Image control onto the design surface of the ASPX page. Set the `ImageUrl` property to that of your image. Set the `AlternateText` property to a suitable short description of your image.

Create a web user control. Web user controls enable you to create reusable combinations of controls and functionality.

Master It Create a web user control with two TextBox controls and a Label control that calculates a percentage based on amounts entered into the TextBox controls.

Solution Use the Add New Item dialog box to create a new `WebUserControl.ascx` template. In Design view, drop a Table control from the HTML toolbox onto the page. Keep the three default rows and columns.

In the first cell of the top row, type **Enter amount**. In the second cell, type **Enter total**. In the third cell, type **Calculated Percentage**.

From the Standard toolbox, drag a TextBox control into each of the first two table cells in the second row. Drop a Label control into the third cell. Place a Button control in the first cell of the third row.

Delete the default text property value of the Label control. Set the `Text` property of the Button control to **Calculate**. Double-click the Button control to open the code skeleton for the `Click` event for the Button in code-behind. Enter the following code snippet:

```
Label1.Text = CInt(CInt(TextBox1.Text) * 100 / CInt(TextBox2.Text))&"%"
```

This particular example will not do any validation of user input but will perform a basic calculation. Save your work and test it in an ASPX page.

Chapter 27 ASP.Net Web Services

Create a simple ASP.NET web service. Creating ASP.NET web services is straightforward with Visual Studio. ASP.NET web services provide a great method for delivering data and functionality within a distributed environment, including the Internet.

Master It Develop an ASP.NET web service that enables the user to add two numbers.

Solution

1. Open Visual Studio 2008 and choose File ➢ New Web Site.
2. From the New Web Site dialog box, select ASP.NET Web Service and click OK.
3. In `App_Code/Service.vb`, change the default reference of `http://tempuri.org` to something more relevant to your situation.
4. Delete the default HelloWorld `<WebMethod()>` and add the following code snippet:

```
<WebMethod()> _
    Public Function AddNumber(ByVal value1 As Integer, ByVal _
 value2 As Integer) As String
        AddNumber = CInt(value1 + value2)
    End Function
```

Consume an ASP.NET web service. Adding a web reference or service reference to a web service is a key element to creating an application that can consume the web service.

Master It Create a new website and add a service reference to a web service on your machine.

Solution

1. Open Visual Studio to an existing web service project such as MyWebService. Run the application to create a running instance of ASP.NET Development Server.
2. From the Start menu, open a new instance of Visual Studio 2008. Choose File ➢ New Web Site.

3. From the New Web Site dialog box, select ASP.NET Web Site and click OK.
4. In Solution Explorer, right-click the name and path of the website at the top of the Solution Explorer tree, and from the context menu choose Add Service Reference.
5. In the Add Service Reference dialog box, type (or paste) the URL for the web service. Click the Go button.
6. After the service has been discovered, click the OK button. The service should now be registered in Solution Explorer.

Work with AJAX technologies. UpdatePanel controls are used in AJAX implementations to provide partial page refreshes. A control placed within the UpdatePanel will automatically refresh the UpdatePanel with a postback. However, you can also use a control located elsewhere on the page to trigger an UpdatePanel refresh.

Master It Add a Button control to an AJAX web page that is set to trigger an asynchronous update for an UpdatePanel control.

Solution

1. Open Visual Studio 2008 and create a new website. Once in the website, use the Add New Item dialog box to add an AJAX web form to the solution.
2. Drop an UpdatePanel control onto the form from the AJAX Extensions toolbox.
3. Drop a Button control onto the form, below the UpdatePanel control. Keep the default `Button1 ID` property.
4. Select the UpdatePanel and from the Properties window, click the ellipsis in the `Triggers` property to open the UpdatePanelTrigger Collection Editor dialog box.
5. Click the Add button and choose AsyncPostBack. In the AsyncPostBack Properties window, set the ControlID to **Button1**. Set the `EventName` to **Click**. Click the OK button to exit the dialog box.

The UpdatePanel will now respond to the `Click` event of `Button1`.

Appendix B

Error Handling and Debugging

Your basic task as a developer is to write functional, robust applications. To write functional applications, you must keep the interface as simple as possible, use your common sense, and listen to the users. If you take the users' comments into consideration while designing your application's interface, you will produce a functional application. Avoid designing complicated forms and don't try to squeeze too much information onto a single form. If you've been around in this field for a while, you already know that this is an acquired skill and that there's no substitute for experience.

Writing robust applications, however, is not as hard. Although writing functional applications is an art, writing robust applications is a technique that can be taught. It takes a lot of code, but it is well within the average developer's skills. A robust application is one that will continue its operation under adverse conditions. The most typical abnormal condition for an application occurs when users supply the wrong data. Users will enter data that defy any logic, and your code should be able to handle them. At the very least, your application shouldn't terminate without giving users a chance to save their data. It's okay to abort an operation and display a warning, but an application shouldn't crash.

Another related issue is that of debugging applications. No amount of error-handling code will do you any good if your application is producing wrong results. Errors, or bugs, in our code are not always obvious, and any nontrivial application contains quite a few of them — just check out the number of fixes and patches for a major commercial application. Debugging is an important aspect of coding an application, and all modern development tools provide valuable assistance in locating problems in your code and fixing them. The bulk of this appendix is devoted to the debugging tools of Visual Studio.

Understanding the Types of Errors

The errors caused by a computer program (regardless of the language in which the program is written) can be categorized into three major groups:

Design-time errors The design-time error, which is the easiest to find and fix, occurs when you write a piece of code that does not conform to the rules of the language in which you're writing. These errors are easy to find because Visual Studio tells you not only where they are but also what part of the line it doesn't understand.

Runtime errors Runtime errors are harder to locate because Visual Studio doesn't give you any help in finding the error until it occurs in your program. These errors occur when your program attempts something illegal, such as accessing data that doesn't exist or a resource to which it doesn't have the proper permissions. These types of errors can cause your program to crash, or hang, unless they are handled properly.

Logic errors The third type of error, the logic error, is often the most insidious type to locate because it might not manifest itself as a problem in the program at all. A program with a logic error simply means that the output or operation of your program is not exactly as you intended it. It could be as simple as an incorrect calculation or having a menu option enabled when you wanted it disabled. Quite often, logical errors are discovered by the users after the application has been deployed.

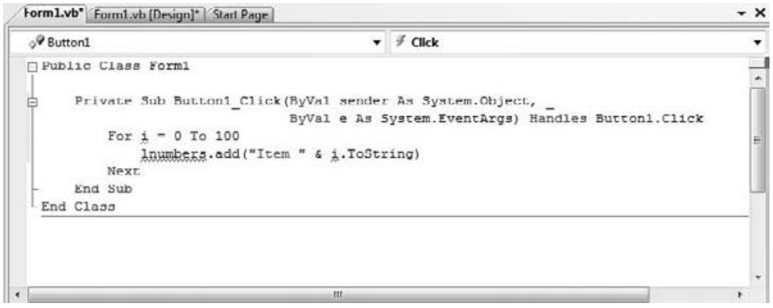
Design-Time Errors

Also called *syntax errors*, design-time errors occur when the Visual Basic compiler cannot recognize one or more statements that you have written. Some design-time errors are simply typographical errors — you have mistyped a keyword. Others are the result of missing items: undeclared or untyped variables, classes not yet imported, incorrect parameter lists in a function or method call, or referencing members on a class that do not exist. By now, you should be familiar with all these types of errors.

A program with as few as one design-time error cannot be compiled and run — you must locate and correct the error before continuing. Fortunately, design-time errors are the easiest to detect and correct because Visual Studio shows you the exact location of these errors and gives you good information about what part of the code it can't understand. What follows is a brief example showing several design-time errors in just a few lines of code.

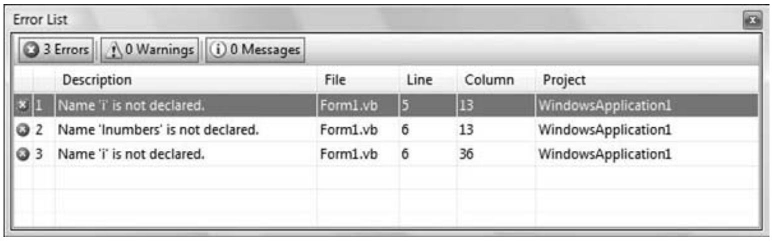
The event code shown in Figure B.1 was typed into the `Click` event of a button named `Button1`. Note the three squiggly lines under various parts of this brief code (under the two instances of the variable `i` and under the term `lbNumbers`). Each of those squiggly lines represents a design-time error. To determine what the errors are, locate the Error List windowin the IDE and bring it forward. The Task List displays the errors seen in Figure B.2 for the code from Figure B.1. If the Infer option is off, the editor will not underline the `i` variable; it will automatically create an integer variable and use it. To make the most of the compiler's syntax error capabilities, turn on the Strict and Explicit options, and turn off the Infer option.

FIGURE B.1
VB.NET identifies the locations of design-time errors.



```
Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, _
                              ByVal e As System.EventArgs) Handles Button1.Click
        For i = 0 To 100
            lnumbers.add("Item " & i.ToString)
        Next
    End Sub
End Class
```

FIGURE B.2
Corresponding errors in the Task List



	Description	File	Line	Column	Project
1	Name 'i' is not declared.	Form1.vb	5	13	WindowsApplication1
2	Name 'lnumbers' is not declared.	Form1.vb	6	13	WindowsApplication1
3	Name 'i' is not declared.	Form1.vb	6	36	WindowsApplication1

You can determine which squiggly line corresponds to which design-time error in the Task List by double-clicking the error in the Task List. The corresponding error will become selected in the code window.

Note that two of the errors are the same: They state, *Name 'i' is not declared*. In this case, these errors are telling you that you've referenced a variable named `i`, but you have not declared it. To fix these two errors, you need to declare the variable `i` as Integer. As a reminder, you can declare the counter of a `For`. . .`Next` loop right in the `For` statement:

```
For i As Integer = 0 To 100
```

The only error remaining now is *Name 'lnumbers' is not declared*. As the programmer of the application, you would probably have some type of idea what `lnumbers` is. In this case, I was attempting to add 100 items to a ListBox, and `lnumbers` is supposed to be the name of the ListBox on the form. This error tells me that I do not have a ListBox on the form named `lnumbers`. I've either forgotten to put a ListBox on the form entirely, or I did add one but did not name it `lnumbers`. To correct the problem, I can either make sure that a ListBox is on my form with the correct name, or change this code so that the name matches whatever I've named the ListBox.

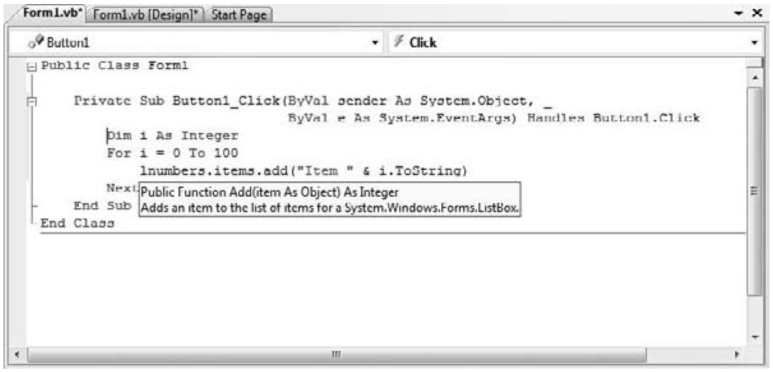
I added a ListBox named `lnumbers` to my form. After doing so, however, I'm still left with a syntax error on the line, this time a different one. The description of the new error is ''*The name 'add' is not a member of 'System.Windows.Forms.ListBox'*.'' This is telling you that it now recognizes that `lnumbers` is a ListBox object, but there is no member (property, event, or method) named `Add` on a ListBox. So what's the correct method to add an item to a ListBox? Some brief research in the help should yield the correct line of code — the one shown in Figure B.3. The statement:

```
lnumbers.Add ("Item " & i.ToString)
```

was changed to

```
lnumbers.Items.Add("Item " & i.ToString)
```

FIGURE B.3
This syntax is correct.



```
Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, _
                              ByVal e As System.EventArgs) Handles Button1.Click
        Dim i As Integer
        For i = 0 To 100
            lnumbers.items.add("Item " & i.ToString)
        Next
    End Sub
End Class
```

Public Function Add(item As Object) As Integer
Adds an item to the list of items for a System.Windows.Forms.ListBox.

Notice that all the squiggly lines are now gone, and the Task List should be empty of errors as well. This means that our program is free of syntax errors and is ready to be compiled and run.

Even if you believe there's nothing wrong with your code, a design-time syntax error is really simple to fix. Do not make any assumptions, read the error message carefully, and consult the documentation. More often than not, it's something almost obvious that you fail to see immediately.

Runtime Errors

Runtime errors are much more insidious to find and fix than design-time errors. Runtime errors are problems encountered by your program while it's running, and they usually happen at the client's site. Runtime errors can take on dozens of different shapes and forms. Here are some examples:

- Attempting to open a file that doesn't exist
- Trying to log in to a server with an incorrect username or password
- Trying to access a folder for which you have insufficient rights
- Accessing an Internet URL that no longer exists
- Dividing a number by zero
- Users entering character data where a number is expected

As you might imagine, runtime errors can be many degrees harder to diagnose and fix in comparison to design-time errors. After all, any error you make in design time is right there on your own development PC. Not only that, but the Visual Studio compiler also goes ahead and tells you right where a design-time error is and why it's an error. The runtime error, by comparison, might manifest itself only in strange computing conditions on a PC halfway across the world. We'll see in later sections how runtime errors can be detected and managed. Basically, we'll insert some special sections in our code to handle possible errors in a code segment with a structured exception handler. Structured exception handlers are discussed in detail later in this appendix.

Logic Errors

Logic errors also occur at runtime, so they are often difficult to track down. A logic error occurs when a program does not do what the developer intended it to do. For example, you might provide the code to add a customer to a customer list, but when the end user runs the program and adds a new customer, the customer is not there. The error might lie in the code that adds the customer to the database; or perhaps the customer is indeed being added, but the grid that lists all the customers is not being refreshed after the add-customer code, so it merely appears that the customer wasn't added.

Here are some actual Visual Basic .NET code snippets that produce logic errors. Consider the following code snippet:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles Button1.Click
   Dim i As Integer
   i = 1
   Do While i > 0
      i += 1
   Loop
End Sub
```

Here we have an integer variable set to 1 and incremented by 1 in a loop. Each time the loop iterates, the number gets bigger. The loop will continue to iterate as long as the variable is greater than 0. See any problem with this? The problem is that the value of the variable will *always* be greater than 0, so the loop will never terminate. This is called an *infinite loop*, and it's a very common error. Of course, this loop isn't exactly infinite; after 2 billion iterations, an overflow condition will cause the application to crash with an overflow exception, which is a good indication about what happened.

Here's another simple example of a logic error:

```
Private Sub ColorTheLabel(ByVal lbl As Label)
   If CInt(lbl.Text) < 0 Then
      lbl.ForeColor = Color.Green
   Else
      lbl.ForeColor = Color.Red
   End If
End Sub
```

This routine was intended to color the text of a label red if the label text contained a negative number, and green if it contained a positive number (or 0). However, I got the logic backward — the label text is green for numbers less than 0, and red otherwise. This code won't produce any design-time errors or runtime crashes. It simply does the opposite of what I intended it to do.

Note finally that logic errors might or might not manifest themselves as program crashes. In the preceding logic error examples, the programs wouldn't have crashed or produced any type of error message — they simply did not perform as intended. It goes without saying that you should test your application thoroughly and discover such errors on your own.

Some logic errors might indeed produce a program crash, at which point the line between a logic error and a runtime error becomes blurry. The fact that a new customer doesn't appear in a grid might cause a crash if your program tries to highlight that new customer in the grid but the customer row isn't there. In this case, we made a logic error (not adding the customer to the grid) that's caused a runtime error (the program crashes when it tries to highlight a row in a grid that doesn't exist). Fixing the logic error would automatically fix the runtime error.

Logic errors are often discovered by the users of the application, although rigorous testing will reveal most of them. When such an error is reported, you must debug the application: Discover the statements that produce the error and fix them. Discovering the statements responsible for a logical error is usually much more difficult than fixing them. We'll discuss the tools for debugging applications in the last section of this appendix.

Working with Exceptions and Structured Exception Handling

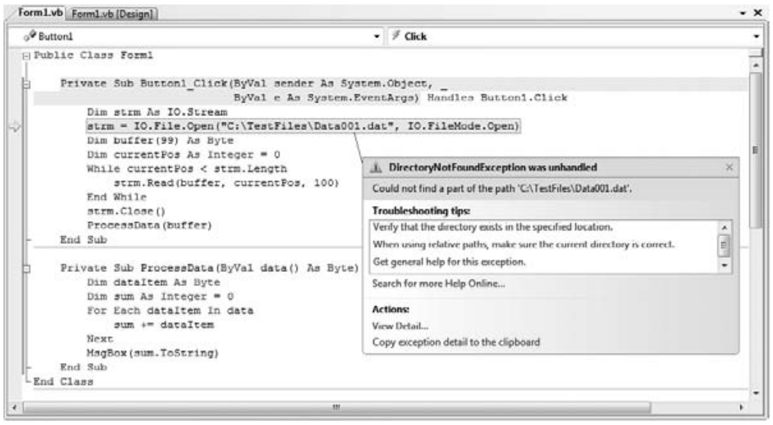
A good deal of the code we write handles errors. It shouldn't come as a surprise, but more than half of a professional application's code validates data and handles possible errors. Most of the error-handling code we write will never be executed. Users aren't supposed to enter a discount percentage that exceeds 100 percent or a future birth date. This isn't supposed to happen and might never happen. However, you must validate the discount's value from within your code and not proceed with your calculations until the user supplies a valid value. We should make an important distinction here: A valid value is not necessarily the correct value, but there's nothing you can do about that.

Consider an application that performs static calculations. The individual parameters supplied by the user might be correct, but when they're combined, the calculations might fail (that is, the calculations might produce results that don't make sense). No parameter is in error, but they're incompatible with one another. For example, they might result in a division by zero, the calculation of the square root of a negative value, and so on. Our task is to detect this condition in our code and allow users to revise their data, rather than allow the application to crash.

The situation we just described can't be handled with data validation. We'll discover a problem with our data only after we attempt to use them in some calculations. Another type of error you can't prevent with data validation is an error caused by the hardware itself. The disk might be full when you attempt to save a large file, or the drive you're accessing might be disconnected. When your application runs into a situation like this, it should be able to detect the error and handle it from within your code. Many of these errors will never happen during the testing phase, unless you're really thorough in your test and take it as far as disconnecting drives and unplugging cables while you're testing your application.

To handle errors that surface at runtime, we use *structured exception handlers*. Exceptions that are handled from within the application's code are called *handled exceptions*, and they result in robust applications. Exceptions that are not handled from within your code are called *unhandled exceptions*, and they lead to program crashes. (In effect, it's the Common Language Runtime, or CLR, that handles these errors in a rather crude manner.)

Figure B.4 shows an example of an exception message. This is the dialog box that appears when you are running your program in the IDE. The statement in error is highlighted, and a box with the error's description appears. The troubleshooting tips that appear in this dialog box provide hints as to what might have caused the exception and, in many cases, suggestions about how to correct the error. If the same error were to be encountered by a user running your program, the dialog box would look slightly different, as seen in Figure B.5. Usually the Details section of the dialog box isn't shown, but I clicked the Details button before capturing the figure. If you scroll the details text to the right, you'll see the number of the line in the source code that caused the runtime exception.



```
Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, _
                              ByVal e As System.EventArgs) Handles Button1.Click
        Dim strm As IO.Stream
        strm = IO.File.Open("C:\TestFiles\Data001.dat", IO.FileMode.Open)
        Dim buffer(99) As Byte
        Dim currentPos As Integer = 0
        While currentPos < strm.Length
            strm.Read(buffer, currentPos, 100)
        End While
        strm.Close()
        ProcessData(buffer)
    End Sub

    Private Sub ProcessData(ByVal data() As Byte)
        Dim dataItem As Byte
        Dim sum As Integer = 0
        For Each dataItem In data
            sum += dataItem
        Next
        MsgBox(sum.ToString)
    End Sub
End Class
```

DirectoryNotFoundException was unhandled

Could not find a part of the path 'C:\TestFiles\Data001.dat'.

Troubleshooting tips:

Verify that the directory exists in the specified location.

When using relative paths, make sure the current directory is correct.

Get general help for this exception.

Search for more Help Online...

Actions:

View Detail...

Copy exception detail to the clipboard

FIGURE B.4
Design-time error message

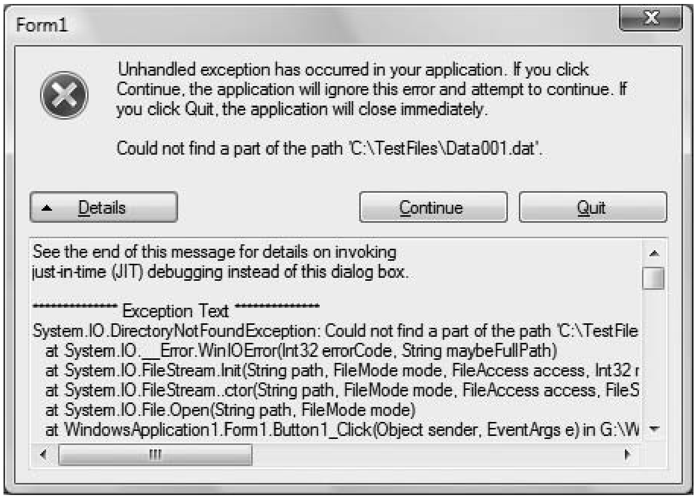


FIGURE B.5
Runtime error message

```
Unhandled exception has occurred in your application. If you click
Continue, the application will ignore this error and attempt to continue. If
you click Quit, the application will close immediately.

Could not find a part of the path 'C:\TestFiles\Data001.dat'.

See the end of this message for details on invoking
just-in-time (JIT) debugging instead of this dialog box.

************** Exception Text **************
System.IO.DirectoryNotFoundException: Could not find a part of the path 'C:\TestFile
   at System.IO.__Error.WinIOError(Int32 errorCode, String maybeFullPath)
   at System.IO.FileStream.Init(String path, FileMode mode, FileAccess access, Int32 r
   at System.IO.FileStream..ctor(String path, FileMode mode, FileAccess access, FileS
   at System.IO.File.Open(String path, FileMode mode)
   at WindowsApplication1.Form1.Button1_Click(Object sender, EventArgs e) in G:\W
```

You could have avoided this type of error by making sure the file exists before attempting to use it. Yet, a missing file is not the only reason for the `Open` method to fail. The user may not have permission to open the specified file. The truth is that you can avoid most runtime exceptions with the appropriate validation code, but validation can't prevent all possible errors. Besides, to write the appropriate validation code, you must foresee all possible errors, and in most cases this is simply not feasible.

Note that this dialog box gives the user the opportunity to continue the program. In some rare cases, this might be desirable, but in most cases you probably would not want your users attempting to continue after a program exception has occurred. Think about it — your program has just encountered some form of data that it cannot handle correctly, and now it is asking the user whether it should attempt to ignore that bad data and continue. It is difficult to predict what type of further problems might result as the program continues on and attempts to handle the bad data. Most likely, further exceptions will be generated as the subsequent lines of code attempt to deal with the same unexpected data.

Ideally, users should never see a dialog box like the one shown in Figure B.5; we'll simply have to handle it ourselves with a structured exception handler. An error handler is a section of VB code that allows you to detect exceptions and take the necessary steps to recover from them. What follows are some exception-handling code examples.

Studying an Exception

The exception dialog boxes shown in Figures B.4 and B.5 were generated by a statement attempting to open a file that doesn't exist:

```
strm = File.Open("C:\TestFiles\Data001.dat", FileMode.Open)
```

This error is rather trivial to fix. Sometimes the description of the error isn't as obvious. Let's consider the statements of Listing B.1, which will also result in a runtime exception.

Listing B.1: An Unhandled Exception

```
Private Sub Button2_Click(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles Button2.Click
   Dim s As String
   s = "answer"
   Label1.Text = s.Substring(10, 1)
End Sub
```

This code is attempting to display the 11th character in the string `answer`. Seeing as the word `answer` contains only six characters, you can imagine how an exception might be generated. Let's examine the exact phrasing of the exception to learn as much as possible about this particular error:

```
startIndex cannot be larger than length of string. Parameter name: startIndex
```

The first thing to notice is that the message box's title is `ArgumentOutOfRangeException`, and it's referred to as an *unhandled* exception. This means that the line of code that generated this error is not contained within an exception-handling block.

The second interesting piece of information is that this exception is of type System.Argument-OutOfRangeException, whatever that means. What's important to note is that the different types of errors can be classified in groups. The previous error message is telling us that the exception object instance generated is of class (type) System.ArgumentOutOfRangeException, which is a descendent of class Exception.

The ''additional information'' block gives us some specific notes on the nature of the error. It tells us that the index and length parameters of the `Substring` method must both lie within the boundaries of the string. In our case, we attempted to retrieve the 11th character of a six-character string, clearly outside the boundary.

Getting a Handle on This Exception

Listing B.2 is the same defective code statement as Listing B.1, but with a simple exception handler wrapped around it.

Listing B.2: Handling an Exception, Version 1

```
Private Sub Button2_Click(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles Button2.Click
   Dim s As String
   s = "answer"
   Try
      Label1.Text = s.Substring(10, 1)
   Catch
      Label1.Text = "error"
   End Try
End Sub
```

This code attempts to do the same thing as the preceding code, but this time the faulty `Substring` statement is wrapped around a `Try...Catch...End Try` block. This block is a basic exception handler. If any of the code after the `Try` statement (and before the `Catch` statement) generates an exception, program control automatically jumps to the code after the `Catch` statement. If no exceptions are generated in the code under the `Try` statement, the `Catch` block is skipped. When this code is run, the `System.ArgumentOutOfRangeException` is generated, but now the code does not terminate with a message box. Instead, the text property of `Label1` is set to the word *error*, and the program continues along.

Listing B.3 handles the same error in a slightly different way.

Listing B.3: Handling an Exception, Version 2

```
Private Sub Button2_Click(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles Button2.Click
   Dim s As String
   s = "answer"
   Try
    Label1.Text = s.Substring(10, 1)
   Catch ex As Exception
      Call MsgBox(ex.Message)
   End Try
End Sub
```

In this example, the exception generates an instance of the Exception class and places that instance in a variable named `ex`. Having the exception instance variable is useful because it can give you the text of the exception, which we display in a message box here. Of course, displaying the exception message in a message box is pretty much the same thing that your program does when an unhandled exception is generated, so it's doubtful that you would do this in your own program. However, you could log the exception text to the event log or a custom error file.

Note that the preceding exception handlers do not differentiate between types of errors. If *any* exception is generated within the `Try` block, the `Catch` block is executed. You can also write exception handlers that handle different classes of errors, as seen in Listing B.4.

Listing B.4: Handling an Exception, Version 3

```
Private Sub Button3_Click(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles Button3.Click
   Try
      Label1.Text = lbStates.SelectedItem.ToString
   Catch exNull As System.NullReferenceException
      Call MsgBox("Please select an item first")
   Catch ex As Exception
      Call MsgBox("Some other error: " & ex.Message)
   End Try
End Sub
```

This code attempts to read the selected item in a ListBox named `lbStates` and display it as the caption of a Label control. If no item is selected in the ListBox, a `System.NullReferenceException` will be generated, and we use that information to tell the user to select an item in the ListBox. If any other type of exception is generated, this code displays the text of that error message.

In the list of exceptions shown in Listing B.4, the more specific exception handler comes first, and the more general exception handler comes last. This is how you'll want to code all your multiple `Catch` exception handlers so that they are handled in the correct order. If you put your more general `Catch` handlers first, they will execute first and override the more specific handlers.

Note that because the Exception instance is declared in each `Catch` block, its scope is limited within that block. The code in Listing B.5 is illegal for scoping reasons.

LISTING B.5: Handling an Exception, Version 4 (Illegal)

```
Private Sub Button3_Click(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles Button3.Click
   Try
      Button3.Text = lbStates.SelectedItem.ToString
   Catch ex As System.NullReferenceException
      MsgBox("please select an item first")
   Catch ex As Exception
      MsgBox("some other error")
   End Try
   MsgBox(ex.message)
End Sub
```

The final `MsgBox()` function call is not valid because the `ex` variable that it attempts to display is not in scope at this point of the procedure. The scope of the two `ex` variables is restricted in their `Catch` blocks.

By the way, you can avoid the `NullReferenceException` altogether by making sure that the user has selected an item in the ListBox control with a few statements like the following:

```
If lbStates.SelectedItems.Count = 0 Then
    MsgBox("Please select a state on the States list!")
    Exit Sub
End If
```

The preceding `If` statement doesn't attempt to process the state unless the user has already selected a state in the `lbStates` control. If not, the event handler is terminated without taking any action.

Wrapping Up with the *Finally* Clause

When an exception is generated and handled by a `Catch` statement, the code execution is immediately transferred to the first relevant `Catch` exception handler block and then continues on out of the `Try...Catch...End Try` block. Sometimes it might be necessary to perform some cleanup before moving out of the exception-handling block. Consider the procedure demonstrated in Listing B.6.

LISTING B.6: A Possible Exception

```
Protected Sub ReadFromATextFile(cFilename as string)
   Dim s As StreamReader
   Dim cLine As String
   Dim bDone As Boolean = False
   lbresults.Items.Clear()
   s = New Streamreader(cFilename)
   Try
      While Not bDone
         cLine = s.ReadLine()
         If cLine Is Nothing Then
            bDone = True
         Else
            Call lbresults.Items.Add(cLine)
         End If
      End While
      s.Close()
   Catch ex as Exception
      Call MsgBox("some error occurred")
   End Try
End Sub
```

This method attempts to read the contents of a text file and put the results into a ListBox, line by line. Most of the reading code is wrapped within a generic exception handler. If an exception is encountered in the main loop, the `s.Close()` line will not be executed. This means that our file stream will never be properly closed, possibly leading to a resource leak.

Fortunately, there is an additional type of block available in exception handlers that specifically allow us to avoid this type of problem. This new block is called the `Finally` block. The code within a `Finally` block always executes, whether an exception is generated or not. The code in Listing B.7 is the same as the method in Listing B.6, but is now modified to wrap the `s.Close()` method inside a `Finally` block.

LISTING B.7: Handling an Exception with a *Finally* Block

```
Protected Sub ReadFromATextFile(cFilename as string)
   Dim s As StreamReader
   Dim cLine As String
   Dim bDone As Boolean = False
   lbresults.Items.Clear()
   s = New Streamreader(cFilename)
   Try
      While Not bDone
         cLine = s.ReadLine()
         If cLine Is Nothing Then
            bDone = True
         Else
            Call lbresults.Items.Add(cLine)
         End If
      End While
   Catch oEX as Exception
      Call MsgBox("some error occurred")
   Finally
      s.Close()
   End Try
End Sub
```

The `Finally` clause of a structured exception handler allows you to guarantee that certain resources or handles are properly disposed of when they are no longer needed.

VB 2008 at Work: The ReadWriteFile Project

In this section, we'll implement two structured error handlers to deal with runtime errors that might occur while opening, reading from, and writing to files. First, we must determine the operations that might throw exceptions at runtime. We'll use the `Open` method of the File class to open a file for reading. The file's name will be supplied by the user through the Open common dialog box. If you look up the `Open` method in the documentation, you'll find a list of all the exceptions that this method might raise; they're listed in Table B.1.

TABLE B.1: Exceptions for the *Open* Method

EXCEPTION	CONDITION
`ArgumentException`	The path argument is a zero-length string, contains only white space, or contains one or more invalid characters as defined by InvalidPathChars.
`ArgumentNullException`	The path argument is a null reference (Nothing in Visual Basic).
`ArgumentOutOfRangeException`	The value of the `mode` argument is invalid.
`PathTooLongException`	The specified path, filename, or both exceed the system-defined maximum length. For example, in Windows-based platforms, paths must have fewer than 248 characters, and filenames must have fewer than 260 characters.
`IOException`	An I/O error occurred while opening the file.
`DirectoryNotFoundException`	The specified path is invalid, such as being on an unmapped drive.
`UnauthorizedAccessException`	The path specified is a directory, or the calling application does not have the required permission to write to or read from the file.
`FileNotFoundException`	The file specified in the path was not found.
`NotSupportedException`	The path is in an invalid format.

Some of these errors can be easily prevented; we can easily make sure that the path is not an empty string and the file is opened in a valid mode. Because we're using the Open built-in dialog box, these errors will never occur. However, another application might read the file's name from another source and it might run into a bad filename or pathname.

The statements in the Try clause of Listing B.8 attempt to open a file for reading. If any of the `Open` method's possible exceptions occurs, the program displays a warning and exits the procedure without opening the file.

LISTING B.8: Handling Exceptions with the Basic File I/O Operations (1)

```
Private Sub bttnFileIOExceptions_Click( _
                          ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
                          Handles bttnFileIOExceptions.Click
Dim FName As String
Dim txt As String
    OpenFileDialog1.CheckFileExists = True
    OpenFileDialog1.CheckPathExists = True
    OpenFileDialog1.Filter = "Text Files|*.txt"
    Dim stream As FileStream
    If OpenFileDialog1.ShowDialog = DialogResult.OK Then
        Try
'            Dim excptnDirectory As System.IO.DirectoryNotFoundException
'            Throw excptnDirectpry
            FName = OpenFileDialog1.FileName
            stream = File.Open(FName, FileMode.Open)
        Catch authorizationException As UnauthorizedAccessException
            MsgBox("The file is read-only")
            Exit Sub
        Catch excptnDirectory As System.IO.DirectoryNotFoundException
            MsgBox("Specified directory not found")
            Exit Sub
        Catch excptnFile As System.IO.FileNotFoundException
            MsgBox("Specified file not found")
            Exit Sub
        Catch IOExcptn As System.IO.IOException
            MsgBox("Couldn't open the file")
            Exit Sub
        Catch excptn As Exception
            MsgBox("Failed to open file." & vbCrLf & excptn.Message)
            Exit Sub
        End Try
        Dim b As Byte
        Dim buffer(stream.Length - 1) As Byte
        Try
            stream.Read(buffer, 0, stream.Length)
        Catch excptnIO As System.IO.IOException
```

```
                MsgBox("Error in reading file")
                Exit Sub
            Catch excptnNotSupported As System.NotSupportedException
                MsgBox("Can't read from file")
                Exit Sub
            Catch excptn As Exception
                MsgBox("The following error occurred while reading" & _
                        vbCrLf & Excptn.Message)
            Finally
                stream.Close()
            End Try
            txt = UTF7.GetString(buffer)
            Console.WriteLine(txt)
        End If
    End Sub
```

If the file is opened successfully, the program attempts to read its contents into an array of bytes by using the FileStream object's `Read` method. This method might also throw several exceptions, which are listed in the `Read` method's entry in the documentation. The two most likely ones are the `IOException` and the `NotSupportedException`. The `IOException` exception occurs when the operating system can't read from the file (either because it's locked by another user or because the media is bad); the `NotSupportedException` occurs when we attempt to perform an illegal operation on the file (such as moving to the beginning of a forward-only stream). The structured error handler handles these two exceptions separately and then handles all other exceptions. In the `Finally` clause, we close the stream.

The first two statements in the `Try` clause are commented out. You can insert statements to throw exceptions in your code to test your error handler. Declare variables to represent an exception type and use them to throw any exception with the `Throw` method.

The code for writing to a file is a little more challenging. The user might select a read-only file, and this is an exception we can handle from within our handler. If you attempt to open a read-only file for writing with the `Open` method, the `UnauthorizedAccessException` exception is thrown. In this exception's handler, you can verify that the file is read-only and prompt the user to reset the file's read-only attribute. If the user agrees to change the read-only attribute, the code executes the `Open` method again. Notice the nested exception handler embedded in the `UnauthorizedAccessException Catch` clause. This handler takes care of any exceptions thrown by the statements that call the `SetAttributes` and `Open` methods.

Listing B.9 shows the complete code for a simple operation. As you can see, the core of the code consists of a few statements (the statements that open the file and write a string to it). The remaining statements handle possible errors and make the program easier to use.

LISTING B.9: Handling Exceptions with the Basic File I/O Operations (2)

```
    Private Sub Button2_Click(ByVal sender As System.Object, _
                              ByVal e As System.EventArgs) _
                              Handles Button2.Click
        Dim path As String
```

```
Dim FS As FileStream
Dim cException As ArgumentNullException
Dim RepeatOperation As Boolean = True
While RepeatOperation
    RepeatOperation = False
    If SaveFileDialog1.ShowDialog <> DialogResult.OK Then
        Exit Sub
    End If
    Me.Refresh()
    path = SaveFileDialog1.FileName
    Try
        FS = File.Open(path, FileMode.OpenOrCreate)
    Catch exPath As PathTooLongException
        MsgBox("Invalid path name")
        RepeatOperation = True
    Catch exPath As DirectoryNotFoundException
        MsgBox("The folder you specified does not exist")
        RepeatOperation = True
    Catch exFile As FileNotFoundException
        MsgBox("The file you specified does not exist")
        RepeatOperation = True
    Catch exArgumentNull As ArgumentNullException
        MsgBox("You have not specified the file to open")
        RepeatOperation = True
    Catch AccessException As UnauthorizedAccessException
        Dim reply As DialogResult
        If File.GetAttributes(path) And _
              FileAttributes.ReadOnly = FileAttributes.ReadOnly Then
            reply = MsgBox("File is read-only. Reset it?", _
                            MsgBoxStyle.YesNo)
            If reply = DialogResult.Yes Then
                Try
                    File.SetAttributes(path, _
                            File.GetAttributes(path) And _
                            (Not FileAttributes.ReadOnly))
                    FS = File.Open(path, FileMode.OpenOrCreate)
                Catch ex As Exception
                    MsgBox("Could not reset read-only attribute!")
                    Exit Sub
                End Try
            End If
        Else
            MsgBox("Can't access file!")
            Exit Sub
        End If
    Catch GeneralException As Exception
        MsgBox(GeneralException.Message)
```

```
                Exit Sub
            End Try
        End While

        Dim b(1024) As Byte
        Dim temp As UTF8Encoding = New UTF8Encoding(True)
        Dim buffer() As Byte
        Try
            buffer = System.Text.Encoding.UTF8.GetBytes( _
                          "Write this string to the file")
            FS.Write(buffer, 0, buffer.GetLength(0))
        Catch exUnsupported As NotSupportedException
            MsgBox("The stream doesn't supported the requested operation")
            Exit Sub
        Catch IOExc As IOException
            MsgBox("Error writing to file." & vbCrLf & _
                   "Please make sure the file isn't " & _
                   "read-only and the disk isn't full")
        Catch GeneralExc As Exception
            MsgBox("Error in application")
            Exit Sub
        Finally
            FS.Close()
        End Try
        MsgBox("Data saved successfully")
    End Sub
```

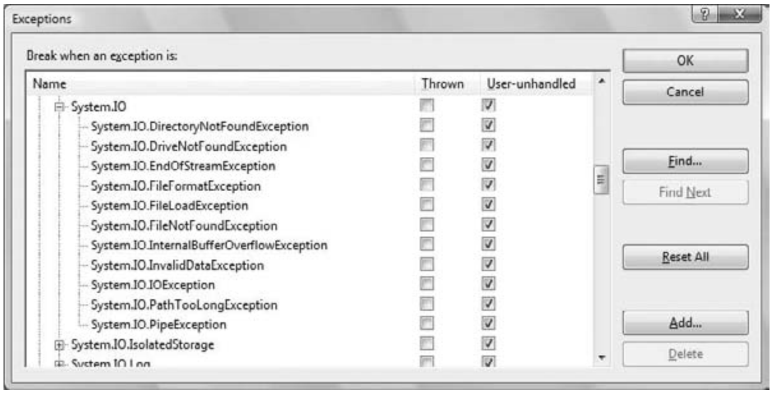
Our error-handling code doesn't do a lot, except for displaying specific error messages that will help the user understand the condition that prevented the successful completion of the operation. However, the `Catch` clauses can be as complicated as you can make them.

To read the attributes of a file, use the `GetAttributes` method of the File class; to set an attribute, use the `SetAttributes` method of the File class. Both methods might throw exceptions, which you should handle with a nested error handler. In the `UnauthorizedAccessException` handler's code, we attempt to reset a read-only file. If the operation succeeds, we repeat the statements that write a string to the file. If the file's read-only attribute can't be reset (as is the case for a file on a website), the subroutine is terminated.

One excellent method of testing software is to ask users who have no preconceived notions about its functionality to test it for you. For example, if you're writing software for the accounting department to use, ask members of the marketing department to test it for you. These users will have much less familiarity with the goals of the software, as well as the expected inputs and outputs. This increases the chance of entering unexpected data, which can lead to unhandled exceptions in your code.

Customizing Exception Handling

There are hundreds of exception classes built into the Framework, and you might not want to handle all of them the same way. You can customize the way certain exceptions are handled by bringing up the Exceptions dialog box (see Figure B.6). The Debug ➢ Exceptions menu opens this dialog box, which allows you to specify how the CLR will handle each exception. This dialog box contains a list of all exceptions, organized in five categories: C++ Exceptions (you can safely ignore this section for your VB or C# applications), Common Language Runtime Exceptions (this is the section of interest to VB developers), Managed Debugging Assistants, Native Runtime Checks, and Win32 Exceptions (for handling exceptions in Win32 functions). For each exception, you can specify whether the IDE will interrupt the execution of the application if an exception occurs. If you want to break the execution of the application every time an exception is thrown, regardless of whether your code handles it, select the Thrown check box. When we debug an application, we frequently want to stop and examine the conditions that caused the exception, rather than let a generic exception handler take over. If you want to break the execution of the application only on unhandled exceptions, select the User-Unhandled check box.



Break when an exception is:

Name	Thrown	User-unhandled
System.IO	☐	✓
System.IO.DirectoryNotFoundException	☐	✓
System.IO.DriveNotFoundException	☐	✓
System.IO.EndOfStreamException	☐	✓
System.IO.FileFormatException	☐	✓
System.IO.FileLoadException	☐	✓
System.IO.FileNotFoundException	☐	✓
System.IO.InternalBufferOverflowException	☐	✓
System.IO.InvalidDataException	☐	✓
System.IO.IOException	☐	✓
System.IO.PathTooLongException	☐	✓
System.IO.PipeException	☐	✓
System.IO.IsolatedStorage	☐	✓
System.IO.Log	☐	✓

FIGURE B.6
The Debug ➢ Exceptions dialog box

The exception shown in the figure is one we saw in the earlier examples: `System.IO.FileNotFoundException`. When this exception is first encountered, the system is currently set to do whatever the parent setting specifies. Tracing up the tree in this dialog box, we eventually find that all .NET Framework exceptions are set to continue when they are first encountered, but to break into the debugger if they are not handled in a `Try...Catch...Finally...End Try` block. This is consistent with what we saw in the earliest exception examples — a dialog box is displayed when an exception was encountered, but it disappears when the proper exception-handling code is in place.

Throwing Custom Exceptions

Although it's relatively easy to handle errors in an application's code (after all, your code can interact with the user), things are very different when you write your own components. When you write a class, for example, you might run into a situation that can't be handled from within the class's code. In this case, you must throw an exception from within your class's code and let the calling application handle it. If the arguments passed to a function that performs a series of calculations, for example, are invalid, there isn't much we can do in our code. We can't interact with the user because the component might be running on a remote machine or a web server. What good are error messages displayed on the application server's monitor? They will remain there indefinitely because no user will see them. In an application that runs on a web server, you should log the errors with the `My.Application.Log.WriteEntry` method, which will append a string describing the error to the system's event log.

The solution is to throw an exception from within our code. The arguments themselves need not be invalid; only their combination results in an impossible situation. Consider a class that exposes a number of properties that must be set before calling a method that acts on these properties. If the calling application attempts to call the method without setting the properties first, the method must pass some information back to the calling application. The most robust technique of passing error information back to the calling application is to throw an exception. As you will see, you can throw a generic exception or a custom exception. A custom exception is an object that inherits from the ApplicationException class and can convey additional information to the calling application, besides an error message.

To raise an exception in a class, use the `Throw` method followed by an Exception object. The simplest method of passing an Exception object to the application that uses your custom class is to create an Exception object by calling its constructor. The Exception object's constructor accepts as an argument a string, which is a description of the error. The following statement will cause an exception when executed:

```
Throw New Exception("Age can't be negative")
```

The description of the error should be as specific as possible, and different conditions should produce different errors.

Consider the `BDate` property, which stores a person's birth date. Obviously, the birth date can't be a future date, so we insert some validation code in the Property procedure's code:

```
Public Property BDate() As Date
    Get
        Return _BDate
    End Get
    Set(ByVal Value As Date)
        If DateDiff(DateInterval.Year, Now, value) > 0 And _
           DateDiff(DateInterval.Year, Now, _PersonBDate) < 100 Then
            _BDate = Value
        Else
            Throw New Exception("Invalid date of birth")
        End If
    End Set
End Property
```

You can also define your own exceptions with a class that inherits from the ApplicationException class. Your class should contain three constructors: one without arguments, another with a description, and a third with a message and another exception object. Let's implement a custom exception for the `BDate` property: the AgeException class. Listing B.10 shows the implementation of the AgeException class.

LISTING B.10: Defining a Custom Exception

```
Public Class AgeException
    Inherits ApplicationException

    Public Sub New()
    End Sub
    Public Sub New(ByVal message As String)
        MyBase.New(message)
    End Sub

    Public Sub New(ByVal message As String, ByVal inner As Exception)
        MyBase.New(message, inner)
    End Sub
End Class
```

Now we can revise our `BDate` property's code and raise an exception of the AgeException type, when an attempt is made to set this property to a future date (see Listing B.11).

Listing B.11: Throwing a Custom Exception

```
Public Property BDate() As Date
Get
    Return _BDate
End Get

Set(ByVal Value As Date)
    If DateDiff(DateInterval.Year, Now, value) > 0 And _
        DateDiff(DateInterval.Year, Now, _PersonBDate) < 100 Then
            _BDate = Value
    Else
         Throw New AgeException("Invalid date of birth")
    End If
End Set
```

The last constructor of the custom Exception class allows you to pass the exception raised in the class to the calling application. The following structured exception handler might appear in a class's code and catches any error that occurs in the `Try` block. Instead of handling the error in your class's code, you can pass the exception to the calling application by passing it as an argument to the CustomException class's constructor:

```
Try
    ' statements
Catch Exc As Exception
    Throw New customException("Error in XXX class", Exc)
End Try
```

You can also throw any of the existing exceptions from within your code. Imagine writing the code for an integer property that can accept a value in a given range. If a fellow developer is using your class and attempts to set the property to a value beyond this range, you would probably want to inform the developer that he is entering an invalid value by throwing an exception. That way, the developer using your class can choose to handle this error in his own way by writing an exception handler in his code. Listing B.12 is an example of ''throwing'' an exception of the OverflowException type.

Listing B.12: Throwing a Built-in Exception

```
Private FValue As Integer = 0
Property Value() As Integer
   Get
      Return FValue
   End Get
   Set(ByVal iValue As Integer)
      If iValue <= FMax Then
         FValue = iValue
      Else
         FValue = FMax
         Throw New OverflowException(_
            "Cannot set ProgressBar value to greater than maximum.")
      End If
      Invalidate()
   End Set
```

An Ounce of Prevention

Preventing errors is even better than catching them. Sometimes we can eliminate all sources of error by validating the data we'll use in our calculations. Let's say you're calculating loan payments, which depend on the loan amount, interest rate, and loan duration. Instead of embedding all the calculations in a structured exception handler, you can examine the values of the parameter's loan before you perform the calculations. If you make sure that the interest rate is a value between 1 and 20 (or 0.01 and 0.2, depending on how the interest rate should be expressed) and that the loan's amount and duration are positive values of a reasonable size, you can perform the calculations and be reasonably certain that they will not fail. Data validation is extremely important, and you should always validate the data you're going to process. Sometimes, invalid data might not cause runtime exceptions, but they will certainly produce incorrect results. Your loan payment calculation routine might very well accept a negative interest rate, but what does this really mean? No bank will ever pay you to get a loan. If you validate your data, you can display descriptive error messages to users and help them correct their mistakes as early as possible. A structured exception handler might display a generic message, but your code that validates individual values will provide specific descriptions for all possible errors.

The best approach is a hybrid one. Start with a generic error handler and perform thorough tests, or let inexperienced users test your application. Note the exceptions that are thrown and see how many of the exceptions you can handle with proper validation techniques. These errors need not be caught by an exception handler; you can handle them from within your code before it enters the exception handler. An advantage of this approach is that you can display descriptive error messages and (if possible) use default values in your calculations.

The code in Listing B.12 is taken from a ProgressBar control. It is the code that implements the `Value` property of the ProgressBar control. A check is done to make sure that the value that the property is set to is less than or equal to the value of the `Max` property, because you can't set the current value to be bigger than the maximum-defined value. If the calling application is trying to set the property to a value larger than the `Max`, an exception is generated via the `Throw` statement. This statement instantiates an exception of class OverflowException and produces a custom error that the fellow developer can see in her own exception handler.

The Exception Class

The Exception object represents an exception, and there are different exception classes for different exceptions — all deriving from the Exception class. Each exception object provides information about a specific exception. The `Message` property is a string with the exception's description. The `InnerException` property is an Exception object that represents an exception thrown while an error handler was in effect. If an exception occurs in an exception handler's code, the `Message` property describes the current error, whereas the `InnerException` property represents the error that activated the error handler. The `Source` property is another string with the name of the object that caused the exception or the name of the assembly where the exception occurred. Finally, the `StackTrace` property holds a stack trace, which is a list of all the called methods preceding the exception and the line numbers in the source file(s). The `TargetSite` property returns the name of the method that threw the current exception.

Even if the exception isn't handled in the subroutine where it occurred, you can still recover the line that caused the exception through the `StackTrace` property, which is a long string that contains the chain of procedure calls from the start of the application to the procedure that threw the exception. A typical value of the `StackTrace` property is the following:

```
at Exceptions.Form1.Proc2() in
             C:\Toolkit\Exceptions\Form1.vb:line 168
at Exceptions.Form1.Proc1() in
             C:\Toolkit\Exceptions\Form1.vb:line 147
at Exceptions.Form1.Button1_Click(Object sender, EventArgs e) in
             C:\Toolkit\Exceptions\Form1.vb:line 139
```

This information isn't of much use to your code; you can view the chain of procedure calls to the offending procedure in the Call Stack window, which is discussed shortly. However, it can be a great help when troubleshooting applications that have already been deployed. You can dump this information to a file before you quit the application and use this file's contents as your starting point when you're called to service your application at the customer's site.

The following are some of the most common exceptions, which are represented by individual objects that derive from the Exception class. I first list a general class and its descriptions, followed by more-specific classes that represent specific exceptions of the same type. An `ArgumentException` exception, for example, is thrown every time you call a method with an argument that doesn't match the argument list of the method. An exception of this type might be caused because one of the arguments is Nothing, because you specified more arguments than the method accepts, or because you specified an enumeration member that doesn't exist. These three exceptions are represented by the classes listed in the Derived classes section under the `ArgumentException` entry.

System.ArgumentException Represents the exceptions that occur when one or more of the arguments passed to a method are not valid.

Derived classes: `ArgumentNullException`, `ArgumentOutOfRangeException`, `ComponentModel.InvalidEnumArgumentException`

System.ArithmeticException Represents errors resulting from invalid arithmetic, casting, or conversion operations.

Derived classes: `DivideByZeroException`, `NotFiniteNumberException`, `OverflowException`

System.ArrayTypeMismatchException Represents the exception thrown when you attempt to store a value of the wrong type to an array element.

Data.DataException Represents exceptions generated by the `ADO.NET` components, such as the `ReadOnlyException` exception, which derives from the DataException class and represents the exception that's thrown when a statement attempts to set the value of a read-only field.

Derived classes: `Data.ConstraintException`, `Data.DeletedRowInaccessibleException`, `Data.DuplicateNameException`, `Data.InRowChangingEventException`, `Data.InvalidConstraintException`, `Data.InvalidExpressionException`, `Data.MissingPrimaryKeyException`, `Data.NoNullAllowedException`, `Data.ReadOnlyException`, `Data.RowNotInTableException`, `Data.StringTypingException`, `Data.TypedDataSetGeneratorException`, `Data.VersionNotFoundException`

Data.DBConcurrencyException Represents exceptions that occur during update operations because of concurrency violations.

Data.SqlClient.SqlException Represents the exceptions returned by SQL Server during the execution of a query or stored procedure. You should catch the `SqlException` exception when you call one of the Command class's `Execute` methods.

Data.SqlTypes.SqlTypeException Represents exceptions that occur when you attempt to assign a value of the wrong type to a field or parameter.

Drawing.Printing.InvalidPrinterException Represents exceptions that occur when you attempt to access a printer using invalid settings.

InvalidCastException Represents exceptions that occur during invalid casting or conversion operations.

IO.InternalBufferOverflowException Represents the exception that occurs when a file buffer overflows.

IO.IOException Represents an I/O exception (there are several specific I/O errors).

Derived classes: `IO.DirectoryNotFoundException`, `DriveNotFoundException`,`IO.EndOfStreamException`, `IO.FileLoadException`, `IO.FileNotFoundException`, `IO.PathTooLongException`

MemberAccessException Represents an exception that occurs when you attempt to access a class member that doesn't exist.

Derived classes: `FieldAccessException`, `MethodAccessException`, `MissingFieldException`, `MissingMemberException`, `MissingMethodException`

RankException Represents the exception that occurs when you pass an array with the wrong number of dimensions to a method.

Runtime.Serialization.SerializationException Represents exceptions that occur during the serialization or deserialization process.

Security.Crytography.CryptographicException Represents exceptions that occur during cryptographic operations.

Security.XmlSyntaxException Represents an exception that's thrown when parsing an XML document that contains syntax errors.

System.StackOverflowException Represents the exception that's thrown when the execution stack overflows because of too many nested method calls (usually in recursive procedures).

A Generic Exception Handler

An application should also include a handler for all exceptions that aren't handled in their respective procedures. Because unhandled exceptions propagate upward in the call stack, you can catch them all at the beginning statements of the application. To include an exception handler for all unhandled exceptions, start the application with the `Run` method, as shown in the following code segment:

```
Sub Main()
    Try
        System.Windows.Forms.Application.Run(New Form1())
    Catch ex As SqlClient.SqlException
        MsgBox("SQL server responded with the following message:" & _
                ControlChars.CrLf & ex.Message & ControlChars.CrLf & _
                "Application will terminate!", _
                MsgBoxStyle.Exclamation, "Error!")
    Catch ex As Exception
        MsgBox("APPLICATION ERROR ! " & vbCrLf & _
                ex.Message & vbCrLf & ex.StackTrace())
    End Try
End Sub
```

The code shown here handles two types of exceptions: the ones thrown by SQL Server (the SqlClient class) and general exceptions. When exceptions are caught in this level, it's too late to do anything about them, so you should include error handlers in your procedures as close to the source of the error as possible.

The generic exception handler shown here can be used as a logging tool. For example, you can dump the call stack to a file and then examine this file to find out where the exception occurred and the exception's type. At the very least, you can give users a chance to save their data before quitting the application.

You can also implement the `MyApplication_UnhandledException` event handler, which is fired when an unhandled exception occurs in the application. Here's an outline of the event's definition:

```
Private Sub MyApplication_UnhandledException(ByVal sender As Object, _
            ByVal e As Microsoft.VisualBasic.ApplicationServices._
            UnhandledExceptionEventArgs) Handles Me.UnhandledException

End Sub
```

To access the original exception that caused this event handler to be invoked, use the expression `e.Exception`. The `UnhandledException` event's handler isn't the place to handle any exceptions;

it's too late to remedy the situation that led to the exception and continue the execution of the application. Use this event handler to catch the (rare) exceptions for which you haven't provided proper exception handlers in your code.

Debugging

Encountering errors is nearly a certainty when developing a piece of software. Although syntax errors are easy to fix, logic errors are not so easy. The first implication is that the logic error's source isn't the statement that fails to execute. The logic error has occurred already, and the statement that fails to execute is the manifestation of the error. Many logic errors may not even cause an exception; they will simply produce incorrect results. Your task is to discover such errors in your code (quite often we rely on users to discover such subtle errors) and track them to their source. Fortunately, Visual Studio provides you with a fine selection of tools to detect and remove the errors in your program. The act of hunting for and eliminating errors is called *debugging* because your goal is to remove the bugs (or *debug*) the program.

Setting Breakpoints

The *breakpoint* is your first and most important weapon in the war against bugs. When you set a breakpoint in your program, you're telling Visual Studio to stop executing the program when it reaches a certain line in the code. After the program is stopped, you can examine its state, including the values of the variables, the procedure stack, and the contents of the memory.

Before we can look at debugging essentials, we need some buggy code. Let's write a program to count all the vowels in a string. To set up this program, start a new Windows project; then add a button named `bttnCount` and a TextBox named `txtPhrase` to the form. Add the code from Listing B.13 to the project.

LISTING B.13: Bug-Filled Code

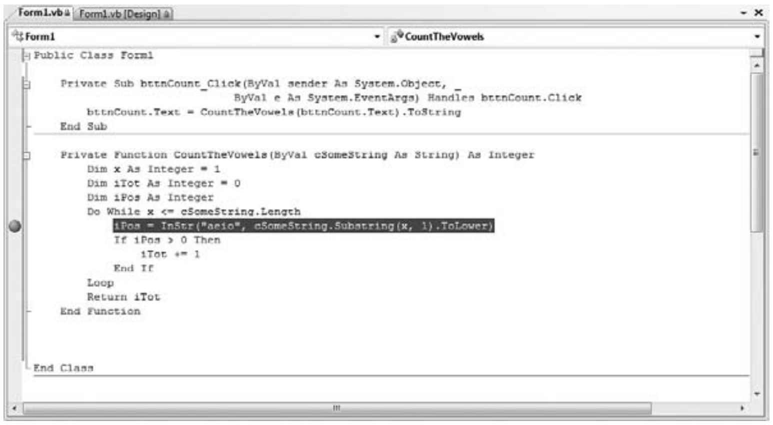
```
Private Sub bttnCount_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles bttnCount.Click
   bttnCount.Text = CountTheVowels(bttnCount.Text).ToString
End Sub

Private Function CountTheVowels(ByVal cSomeString As String) As Integer
   Dim x As Integer = 1
   Dim iTot As Integer = 0
   Dim iPos As Integer
   Do While x <= cSomeString.Length
      iPos = InStr("aeio", cSomeString.Substring(x, 1).ToLower)
      If iPos > 0 Then
         iTot += 1
      End If
   Loop
   Return iTot
End Function
```

The button's `Click` event handler passes the contents of the text box into the function `CountTheVowels()`, which is where all the dirty work will be performed. When the count is obtained, the caption of the button should be replaced with the vowel count. After you type in the code for the program exactly as seen previously, try running the program, entering some text into the text box, and clicking the button. Nothing will happen for a while, and you'll have to stop the application by pressing Ctrl+ Break (or choosing Debug ➤ Stop Debugging). If you wait long enough, an overflow exception will occur because the value of the variable `iTot` has exceeded the maximum value you can represent with an Integer.

Obviously, this little function shouldn't take long to run, so something screwy must be going on, like an infinite loop. Let's set a breakpoint in the function and see if we can spot it.

To set a breakpoint, place the cursor on a line of code in the function where you want the program to stop, and press F9. The line of code should become highlighted, as seen in Figure B.7.

FIGURE B.7
Setting a breakpoint



```
Public Class Form1

    Private Sub bttnCount_Click(ByVal sender As System.Object, _
                                ByVal e As System.EventArgs) Handles bttnCount.Click
        bttnCount.Text = CountTheVowels(bttnCount.Text).ToString
    End Sub

    Private Function CountTheVowels(ByVal cSomeString As String) As Integer
        Dim x As Integer = 1
        Dim iTot As Integer = 0
        Dim iPos As Integer
        Do While x <= cSomeString.Length
            iPos = InStr("aeio", cSomeString.Substring(x, 1).ToLower)
            If iPos > 0 Then
                iTot += 1
            End If
        Loop
        Return iTot
    End Function


End Class
```

After a breakpoint is set, you can begin the program, type some text into the text box, and click the button. The debugger will stop when it reaches that same line of code, this time highlighted in yellow (this means that the program has stopped execution on that exact line of code). Now we can start looking around. First, take the cursor and hover it over some of the areas of code. You should be able to see a ToolTip displaying the value of the various variables you rest the mouse over, such as the one shown in Figure B.8.

The figure displays the first of the errors I made coding this program. The value of the variable `cSomeString` is ''Count the Vowels,'' but this is not the string I typed into my text box. Why is this string being passed into the function? A quick examination of the function call reveals this problem:

```
bttnCount.Text = CountTheVowels(bttnCount.Text).ToString
```

I inadvertently passed the `Text` property of the button `bttnCount`, when my intention was to pass in the value of `tbPhrase.Text`. This is a perfect example of a logic error. The code works fine (well, it will work fine after we find the rest of these bugs), but it won't count the vowels in the string that we intended to count. The fix for this first bug is easy. First, stop the program from running by choosing Debug ➤ Stop Debugging (or by pressing Shift + F5). After the program is stopped, change the `CountTheVowels()` function call as follows. Now we're passing in the string we intended:

```
bttnCount.Text = CountTheVowels(txtPhrase.Text)
```

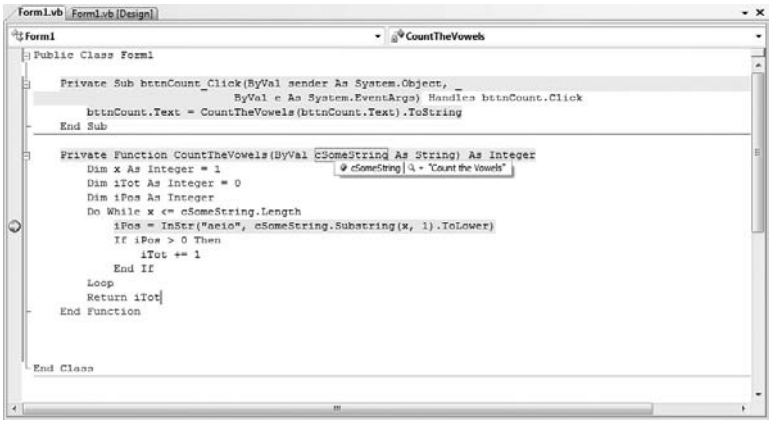


FIGURE B.8
ToolTips display the value of variables.

```
Public Class Form1

    Private Sub bttnCount_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) Handles bttnCount.Click
        bttnCount.Text = CountTheVowels(bttnCount.Text).ToString
    End Sub

    Private Function CountTheVowels(ByVal cSomeString As String) As Integer
        Dim x As Integer = 1
        Dim iTot As Integer = 0
        Dim iPos As Integer
        Do While x <= cSomeString.Length
            iPos = InStr("aeio", cSomeString.Substring(x, 1).ToLower)
            If iPos > 0 Then
                iTot += 1
            End If
        Loop
        Return iTot
    End Function


End Class
```

Stepping Through

As it often happens, we started looking for an infinite loop but found another unrelated bug first. Now that we squashed that bug, we can go back to running the program and looking for the original problem. Start the program again, type some text into the text box, and click the button. Once again, the program should stop at the breakpoint.

Let's watch a few of the program lines run in sequence and see whether they tell us anything. To make the program step through the current line of code, press F10. Each time you press F10, one line of code will execute, and the yellow highlight will move to the next line of code that is about to be executed.

Single Stepping-Through Procedures

The F11 key also steps through the code, but it will step into any procedures that are called. The F10 key steps over the procedure calls, running them all at once and returning back to the original spot. This allows you to skip over the line-by-line tracing of procedures that you are not currently debugging.

You can continue to trace through the loop line by line and examine variable values with the ToolTips. Can you figure out the cause of the infinite loop? Perhaps it's time to bring in some more debugging tools.

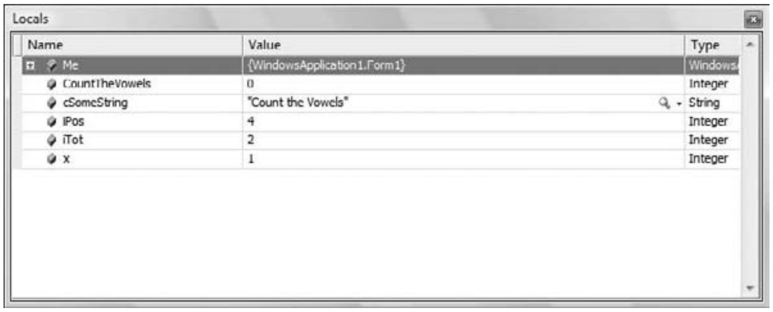
While at a breakpoint, you can even edit the code and then press F5 to continue. This feature is known as *edit and continue*, and it's been the most popular feature of Visual Basic for a decade now. You can edit the next statement to be executed and then execute it by pressing F10. You can also set the value of a local variable by typing an assignment statement in the Command window. After you're finished editing your code, you can execute the highlighted statement by pressing F10. You can also execute any other statement in the current procedure, not necessarily the highlighted one.

Just grab the yellow mark in the left margin of the highlighted statement and drop it in front of the statement you want to execute next. Then press F10 to execute the selected statement.

Using the Local and Watch Windows

While still stopped in debug mode, choose Debug ➤ Windows ➤ Locals from the menu. The Locals window should be displayed in the lower section of the IDE (see Figure B.9). This window shows you the current value of all of the locally declared variables. Now we can see the value of the variables changing as we step through the program.

FIGURE B.9
The Locals window

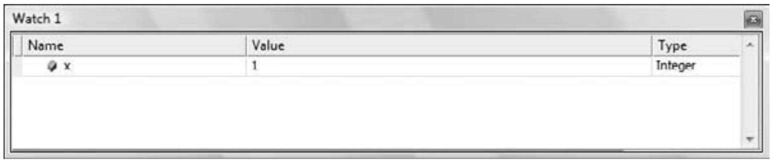


Locals

Name	Value	Type
Me	{WindowsApplication1.Form1}	Windows
CountTheVowels	0	Integer
cSomeString	"Count the Vowels"	String
iPos	4	Integer
iTot	2	Integer
x	1	Integer

Try stepping through the loop a few more times. What you might notice is that the values of the variables aren't changing. To get even more information, highlight the entire phrase `cSomeString.Substring(x, 1).ToLower`, right-click the name of the *x* variable in the statement, and choose Add Watch from the context menu. This brings up the Watch 1window, as seen in Figure B.10.

FIGURE B.10
Watch window



Watch 1

Name	Value	Type
x	1	Integer

The Watch 1 window is similar to the Locals window, but it allows you to look at the value of complex expressions such as the one we just placed in it. Once again, try stepping through the loop a few times. You might expect that the `Substring` method would be incrementing the letter in the string as the loop iterates, but that isn't happening. The only logical reason for this is that the value of counter variable *x* isn't changing. Let's look at the loop again:

```
Do While x <= cSomeString.Length
   iPos = InStr("aeio", cSomeString.Substring(x, 1).ToLower)
   If iPos > 0 Then
      iTot += 1
   End If
Loop
```

I made one of the classic looping blunders here: I set up a counter variable *x* to loop through the string character by character, but I never added any code to increment the counter! That's as sure a recipe for an infinite loop as anything. Fixing that problem is an easy remedy (see the highlighted code):

```
Do While x <= cSomeString.Length
   iPos = InStr("aeio", cSomeString.Substring(x, 1).ToLower)
   If iPos > 0 Then
      iTot += 1
   End If
   x += 1
Loop
```

Okay, that bug is squashed, so it's time to remove the breakpoint and rerun the program to see whether it works. This time, however, the program crashes and burns with the following error:

```
An unhandled exception of type
   'System.ArgumentOutOfRangeException' occurred in
    mscorlib.dll. Additional information: Index and
    length must refer to a location within the string.
```

We've seen that error before — it means that we tried to look at a character beyond the length of the string. By checking the Locals window, you should be able to eventually track down this problem. The problem here is that the loop counter *x* starts at character 1 and ends at value `cSomeString.Length`. Although that range is correct for VB version 6 and earlier, .NET strings are indexed *starting at 0*. Oops. We need to modify our procedure as shown by the two highlighted items here:

```
Private Function CountTheVowels(ByVal cSomeString As String) As Integer
   Dim x As Integer = 0
   Dim iTot As Integer = 0
   Dim iPos As Integer
   Do While x < cSomeString.Length
      iPos = InStr("aeio", cSomeString.Substring(x, 1).ToLower)
      If iPos > 0 Then
         iTot += 1
      End If
      x += 1
   Loop
   Return iTot
End Function
```

This modified loop starts at 0 and ends at `cSomeString.Length – 1`, which is the correct way to iterate through a string. Once again, you can try to remove all breakpoints and rerun the application. This time, our code will produce a value. If you examine the string and the number of vowels reported by the application, you'll realize that it's not the correct answer! A manual count gives 11 vowels in the test string *The quick brown fox jumps over the lazy dog*, but the program reports 10 vowels.

It looks like there's another logic error, so let's use the debugging tools to squash it. Look at the actual comparison of the character to the vowel list:

```
iPos = InStr("aeio", cSomeString.Substring(x, 1).ToLower)
```

Where's the *u*? It looks as if I simply forgot to include it in the vowel list. After adding the *u* back in, the final working code looks like Listing B.14.

Listing B.14: Bug-Free Code

```
    Private Sub bttnCount_Click(...) Handles bttnCount.Click
       bttnCount.Text = CountTheVowels(txtPhrase.Text)
    End Sub

    Private Function CountTheVowels(ByVal cSomeString As String) As Integer
       Dim x As Integer = 0
       Dim iTot As Integer = 0
       Dim iPos As Integer
       Do While x < cSomeString.Length
          iPos = InStr("aeiou", cSomeString.Substring(x, 1).ToLower) If iPos > 0 Then
iTot += 1 End If x += 1 Loop Return iTot End Function
```

This is the type of error you can catch with exhaustive tests — and often it's the users of the application who catch such errors, despite our tests. You can actually test the program with a string that doesn't contain the character *u*, see that the code works nicely, and distribute it. Soon you will receive messages indicating that your application doesn't work. Yet this application has been tested and seems to work fine. The tests, however, were not exhaustive.

You should also try to test your applications with extreme situations (a blank string, for example, or a very large one, an invalid numeric value, and so on). The final test, of course, is to pass the application to users and ask for their comments. Unfortunately, we don't write software for each other. We write software for people intelligent enough to crash an application in minutes, but not intelligent enough to keep it running.

Index

Note to the Reader: Throughout this index **boldfaced** page numbers indicate primary discussions of a topic. *Italicized* page numbers indicate illustrations.