CD-ROM includes Visual Basic 2005 Express Edition

Visual Basic®
2005 Express Edition

STARTER KIT

Andrew Parsons

wrox™

Wrox's Visual Basic® 2005 Express Edition Starter Kit

Andrew Parsons

WILEY

Wiley Publishing, Inc.

Wrox's Visual Basic® 2005 Express Edition Starter Kit

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

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

Credits

Acquisitions Editor
Katie Mohr

Senior Development Editor
Kevin Kent

Technical Editor
Todd Meister

Production Editor
William A. Barton

Copy Editor
Luann Rouff

Editorial Manager
Mary Beth Wakefield

Vice President & Executive Group Publisher
Richard Swadley

Vice President and Publisher
Joseph B. Wikert

Permissions Editor
Laura Moss

Media Development Specialist
Angela Denny
Kit Malone
Travis Silvers

Project Coordinator
Michael Kruzil

Graphics and Production Specialists
Carrie A. Foster
Lauren Goddard
Denny Hager
Joyce Haughey
Jennifer Heleine
Barbara Moore

Quality Control Technicians
Leeann Harney
Susan Moritz
Joe Niesen

Proofreading and Indexing
TECHBOOKS Production Services

About the Author

Andrew Parsons has been programming with the Basic language for more than 20 years and with Visual Basic for the last eight years. He has experience with more than a dozen programming languages but keeps coming back to the Basic language because of its easy-to-understand syntax and the powerful features found in the modern versions, and he believes that Visual Basic is the best language to program in no matter what your level of experience.

Andrew has written several books and technical articles about Visual Basic for magazines in Australia and New Zealand and is constantly improving his own skills in Visual Basic with ongoing programming work with companies such as Quicken Software (associated with Intuit Inc.) and APS. When he's not writing code for other people, or books to help people learn how to program effectively, he serves as editor of *MSDN Magazine,* Australia and New Zealand, and still finds time to write add-ins for Microsoft Office at GrayMatter Software (`www.graymatter.com.au`).

You can contact Andrew at `andrewp@parsonsdesigns.com`.

Contents

Contents

Contents

Contents

Acknowledgments

While I would love to claim that this book is the result of only my own work, it just wouldn't be true. Without the help of a number of colleagues, I would not have been able to complete this book at all, let alone with the high quality of examples and the accuracy of code listings that you'll find throughout the chapters.

In particular, I would like to thank the following people from Microsoft who have been continuously available to help out when I was stuck with various beta builds of Visual Basic Express and who gave me excellent feedback that made the book better — Charles Sterling, Frank Arrigo, Ari Bixhorn, and Jay Roxe. There are a heck of a lot of other Microsoft guys in Australia and the United States who have helped out in various ways, too — to all of you, a big THANK YOU!

It also helped that I had an awesome set of fellow developers out there who are as committed to helping people learn how to program as I am, and the following names are just some of the guys who have encouraged me in a myriad of ways while I was writing this book. So, to Tony Gray, Nick Wienholt, Nick Randolph, Greg Low, Mitch Denny, Carl "GoatBoy" Belle, Kevin Johnson, and "uber-boss" Pierre Le Grange: You all know what you did and it was all worth it — thanks for sharing the passion I have to help other people get into programming.

Saving the best for last — I want to thank my family. Without the support of my wife, Glenda, and her understanding and acceptance of the *many* late nights and absences while I slaved away at this book, it just wouldn't have been possible at all. And to my kids, Jacob and Ashleigh, I love you, and thanks for loving me back!

One last note — in a pretty special way, I've written this book for my son, Jacob. He's convinced that he wants to follow in my footsteps as a programmer, and I feel privileged to be able to write a book that will help him learn how to program, too. It's not often that a father has an opportunity to help his children in this unique way, and I'm very thankful that I can do it for him.

Jake, you rock, little buddy!

Introduction

So you want to get a proper start in programming but don't know quite where to begin? You couldn't have chosen a better tool to get you on the ground running than Microsoft's new programming application, Visual Basic 2005 Express Edition. Of course, you'll now need to learn how to use it, maximizing your education while minimizing the impact on your busy life.

That's where this book comes in. Not only do you have a comprehensive introduction to Visual Basic Express as a language and a development tool, but you also have tips, tricks, and additional techniques that will bring you up to speed before you know it.

From installation to building your own programs, customizing existing code, debugging, securing, and deploying solutions, the next few hundred pages will be your guide to the world of Visual Basic Express.

I've been using the Basic programming language in many forms for over 20 years, and I freely and happily admit that this version is the easiest I've ever encountered. Considering that Basic as a language has always been one of the most easily understood, that's saying something.

Who This Book Is For

If you've picked this book up to see what Visual Basic is all about, then I've got a little secret — this book is for you. *Wrox's Visual Basic 2005 Express Starter Kit* comes with Visual Basic Express and other Microsoft products, such as Visual Web Developer Express and SQL Server Express, on a CD — so you don't need anything else other than what you're holding in your hands.

This book has been designed from a practical, task-oriented approach so that the information taught is backed up with solid examples that confirm and extend the text. If you're someone who prefers to get straight into your learning experience, rather than try to wade through thousands of pages of text, this is exactly the book you need. From the first chapter, you will be writing programs and learning how to use Visual Basic to solve common programming tasks.

If you've used the Basic language in its previous forms, you'll appreciate the elegance and simplicity of this latest iteration, which is coupled with the most powerful library of functions and classes Visual Basic has ever been able to access. In addition, if you're new to the language or new to programming, this book will introduce you to the important concepts and information you'll need to get up to speed — by the end of this book, you'll find that Visual Basic Express is so easy to learn that you'll wonder why you haven't been programming already.

What This Book Covers

This book is completely, unabashedly focused on the just released Microsoft Visual Basic 2005 Express Edition. From installation to deployment, everything that you can do in Visual Basic Express is discussed here so you can get up to speed as quickly as possible.

It should be noted that Microsoft has released Visual Basic in a few different editions this time around. First, there is the professional programmer's tool, Visual Studio 2005, which includes Visual Basic 2005 (in both Professional and Enterprise versions). The newcomers to Microsoft's development tool collection are the Express Editions, of which Visual Basic features in two: Microsoft Visual Basic 2005 Express Edition and Microsoft Web Developer 2005 Express Edition. As the name of the latter suggests, Web Developer Express enables you to create applications designed to run over the Internet and enables developers to write their code in the Visual Basic language. However, it is Microsoft Visual Basic 2005 Express Edition that is the focus of this book.

> *Although these other editions of Visual Basic are not covered in detail, Visual Web Developer 2005 Express Edition is featured in Chapter 9, which deals with programming for online applications.*

How This Book Is Structured

To ease your way into the world of Visual Basic programming, I've split the information into three general parts — "Getting Familiar," "Extending Yourself Is Good," and "Making It Hum." As the titles may intimate, I first introduce you to Visual Basic, then describe how to take control of the language, and then finally explain how to fine-tune everything and make all the bells and whistles work.

❑ **Part I, "Getting Familiar"** — Part I covers Visual Basic first as a language, and then as a development environment. The chapters in this section show you how to install Visual Basic Express and navigate around the environment, building your first program as you go, and then it delves into detail about the user interface, event programming, and how to access data.

❑ **Part II, "Extending Yourself Is Good"** — Part II is where things start getting really interesting, showing you how to write proper programming code by creating additional features for your applications, such as multiple users and custom-built controls. You'll also learn how to debug code that isn't functioning correctly.

❑ **Part III, "Making It Hum"** — In Part III of the book, you'll be introduced to topics that previously would have been well out of reach for the beginner and intermediate programmer. XML processing, data encryption, and notification dialogs were all difficult to implement until .NET came along. Using Visual Basic Express smoothes those processes even further so that they become almost as easy as the introductory lessons most programmers learn.

As a bonus to learning each individual task, if you follow the steps outlined in every chapter, you'll end up with the basics of your own personal organizer, complete with DVD library; information about friends and family members, including birthdays and contact information; and a reminder system so you can ensure that you don't forget to do the important things that need doing.

What You Need to Use This Book

Everything you need to use this book can be found on the accompanying CD. You'll need Visual Basic 2005 Express Edition installed, as well as SQL Server Express for some of the later topics, both of which have installers on the CD. Apart from that, everything else you will create yourself by following the examples and exercises found in each chapter. If you're not sure of the best way to tackle an exercise at the end of a chapter, Appendix C has suggested answers for each one so you can be confident that you're learning what you need to know.

Conventions

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

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

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

As for styles in the text:

❑   I *highlight* important words when I introduce them.

❑   I show keyboard strokes like this: Ctrl+A.

❑   I show filenames, URLs, and code within the text like so: `persistence.properties`.

❑   Code is presented in two different ways:

```
In code examples, I highlight new and important code with a gray background.
```

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

Source Code

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

Because many books have similar titles, you may find it easiest to search by ISBN; for this book the 10-digit ISBN is 07-64595-9573-3 (changing to 978-0-7645-9573-8 as the new industry-wide 13-digit ISBN numbering system is phased in by January 2007).

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

Errata

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, such as a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata, you may save another

reader hours of frustration; and at the same time, you will be helping us provide even higher-quality information.

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

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

p2p.wrox.com

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

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

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

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

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

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

Part I
Getting Familiar

1

Basic Installation

Ever wondered where Basic came from? Much maligned but still the most popular programming language in the world, the Basic language has enjoyed a colorful past and many evolutions to get where it is today. In this chapter, you'll discover the origins of this powerful programming language. You'll install Visual Basic 2005 Express Edition along with the supporting applications and frameworks you'll need to write programs effectively. And finally, yes, you will indeed write your first program.

By the time you hit the end of this chapter, you'll be familiar with how Visual Basic is put together and be ready to create programming projects that will form the basis for all of your solutions from here on out.

In this chapter, you learn about the following:

❑   The history of Visual Basic as a language

❑   Installing Visual Basic Express and its dependencies

❑   Creating your first program

Where Did Basic Come From?

If you tell friends or work colleagues who are experienced in programming that you're going to learn Visual Basic, there is a good chance that they'll look at you with a question in their eyes. That questioning glare is usually an indicator that they're in what I call the "other half" of the programming world. This is the group of programmers who still believe that Basic is not a real programming language, and should be reserved for people who don't know how to write a "real" program.

If this happens to you, just look at them and smile. For while Basic has indeed had a rocky history, the last couple of versions of Visual Basic rival the best alternatives in development, and with Visual Basic 2005 Express's extra features that make it even easier to create full-blown solutions, not only will your programs be able to achieve the same results as the best professional coder, but you will also be able to do it in less time — much less time.

However, to be fair, this section provides a quick rundown of where Visual Basic Express came from, just so you know how far it has come. You'll learn that Visual Basic has a rich past that has helped it evolve into a solid, respected language that often leaves the more recent programming languages scrambling for a foundation on which they can be compared against it.

The Basic programming language was first created back in 1964 — more than 40 years ago. Its very inception was meant to make programming easy and more accessible. In fact, the name was actually originally an acronym that stood for Beginner's All-Purpose Symbolic Instruction Code. It was designed as an interim step for students when they were learning programming concepts for more complex languages such as Fortran.

In the 1970s, Bill Gates and Paul Allen got involved and worked with the company MITS (Micro Instrumentation Telemetry Systems) to develop a version of Basic for the Altair PC. From that humble beginning, Gates and company ported Basic to various other computing platforms, and by the end of that decade, most computers had some form of the Basic language. It was from this starting point that both its ease of use and popularity, as well as the disparaging opinions of many hardcore programmers, sprang.

When DOS was first released for the early PCs, versions of a Basic interpreter were distributed along with the operating system. Programming code can be executed in two ways — interpreted or compiled:

❑ When it is **compiled**, it is assembled into the underlying machine code and can execute fast. However, the compilation can take a while, and the program will not execute at all if even one error is present.

❑ An **interpreter**, on the other hand, requires another program to run through the code one line at a time and execute it piece by piece. While this is slower than compiled code, it doesn't require a compilation routine before running, and it can execute working code up to the point where an error occurs. Basic, and Visual Basic in particular, requires some form of a runtime component because of the interpretive nature of the language compilers.

Microsoft took the command-line interpreter to the next step and introduced QuickBasic. QuickBasic did actually compile the code into an executable, but it was still slow in comparison to the professional languages on the market. In the late 1980s and early 1990s, Alan Cooper created a prototype that enabled a developer to dynamically add components, then called *widgets,* to a program running off a small, custom-built language engine. Microsoft bought the concept and combined it with QuickBasic to form Visual Basic 1.

And Then Came Visual Basic

Visual Basic was a revolution to Basic programmers worldwide as it enabled them to drag and drop controls from a toolbox onto their forms without having to write any code at all. It also changed the focus of the actual code to an event-oriented model that reacted to things happening, as opposed to making things happen.

Visual Basic's versatility enabled third-party companies to develop add-ins and additional controls that Visual Basic programmers could use in their own applications, and the popularity of the language grew hugely.

Subsequent versions of Visual Basic introduced database support (ODBC in VB2, and Jet in VB3) and the ability to create your own add-ins and classes (in VB4), and ultimately your own controls (in VB6). While all of this was happening, Basic appeared in other applications such as Access Basic and VBScript for Internet Explorer. This integration of Basic as a way of programmatically accessing features in Windows and applications culminated in Visual Basic for Applications, which first appeared in Microsoft Office 97.

Throughout all these stages of its evolution, however, Visual Basic was still crippled with additional run-time components and a (much) less than perfect implementation of object-oriented programming that hurt its reputation in the performance and pure programming stakes.

That all changed with .NET. Visual Basic .NET was the first fully compiled language and required no extra runtime component other than the one required by all other .NET languages — the .NET Common Language Runtime (CLR). Visual Basic .NET programs compile down to the same assembled code that the other .NET languages do; and because of this, Visual Basic has no performance issues in comparison to C# or C++.

> **In the last few paragraphs, several programming terms have been used that you may not be familiar with. If you are new to programming, then the next few chapters will be extremely useful to you — particularly the information in Chapter 2 that explains the most commonly used object-oriented programming terms that you'll encounter in Visual Basic Express.**

The Old and the New

The beauty of this latest move for Basic is that it has not lost the ease of use and additional features that make it the choice of many programmers — wizards, intuitive user interface design, and some excellent debugging features (although edit-and-continue was removed in the early days of .NET, it lives again in Visual Basic 2005 Express).

In fact, the modern development environment for .NET has more in common with the way Visual Basic 6 worked than the C++ equivalent. The toolbox, Solution Explorer, and properties pages are almost unchanged, and the way of associating code with user interface elements is identical to previous versions. For people with previous experience in Visual Basic programming, the only real hurdle is learning how to handle the new way of actually coding — proper object-oriented programming is admittedly different from the way VB6 did it.

So here we are, with a programming language that has evolved over more than 40 years and through many iterations and somehow has maintained a freshness with each release that has kept programmers faithful to it over all that time. It is a language that possesses an incredibly robust and intuitive framework of objects and programming constructs that ease you, as a programmer, into creating full-blown applications almost without thought, and an environment that can produce applications that rival the professionally built solutions on the market in performance and user interface. Visual Basic 2005 Express — want to use it? Thought so.

Let's Get Started

Obviously, before you do anything else, you're going to need to install Visual Basic Express on your computer. Microsoft has fine-tuned this process over the years, and you'll find the steps to be as easy as 1-2-3.

When you go through the Visual Basic 2005 Express Setup wizard, you need to select only a couple of options before the setup process takes over and does the rest for you. After you read and accept the license agreement, the installation program will examine your system and present you with a list of two optional products (see Figure 1-1).
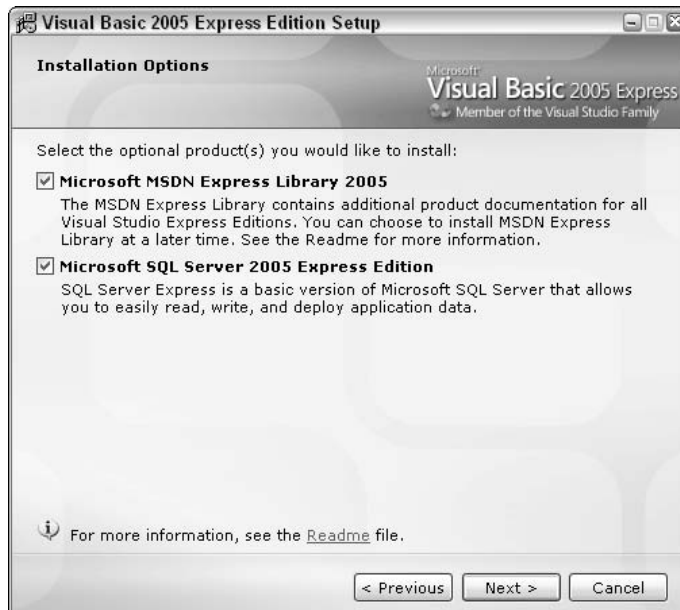
Figure 1-1

As far as I'm concerned, both of the optional components are essential:

❑ The **MSDN Express Library** includes the documentation for both the .NET Framework and Visual Basic 2005 Express. If you do not include this in the installation, you will have only very rudimentary help available to you without going to the Internet.

❑ The second option includes **SQL Server 2005 Express Edition** in the installation process so that you can develop full-blown database-based applications. And if you're still not convinced, you will need it to be installed if you want to complete all of the exercises and tasks set out in this book.

The only exception would be if you already have SQL Server installed on your system. In that case, you could use the existing installation for any database server examples instead, although I cannot guarantee they will work as expected if you are using an older version of SQL Server.

The only other decision that you have to make is to where to install the application. Note that you don't actually get to choose the location of the optional components or the underlying .NET Framework. In addition, this location does not affect the location of the projects you will create — you'll set that location later in this chapter. Once you've made that decision, click Next to start the actual file copy and registration process.

As the installer copies each component over to your computer, it will mark the status on an interim screen. The obvious icons will point out any errors, but most likely you'll encounter nothing but success. In the event of an error, the installation process will advise you as to what steps to take to rectify it before you try again.

Fortunately for you, the rest of the installation is automatic, and while it can take quite some time, you can sit back and have a coffee (and perhaps a Danish) while you wait. When you're presented with the final screen, you have the capability to submit to Microsoft a copy of the installation log so they can check it against what they expect.

While many people believe submitting this information is either pointless or a way for Microsoft to gain access to private data, Microsoft does actually find the information useful in fine-tuning its processes, and anything that improves the speed and efficiency of an installation process is something I am 100 percent behind.

What It Looks Like

Once you have successfully installed Visual Basic 2005 Express, you can start it up by selecting it from your Start menu. Click Start ➪ All Programs ➪ Visual Basic 2005 Express Edition. After the obligatory splash screen identifying the application and version, you'll be presented with an interface much like the one shown in Figure 1-2.

The main program is known as an *Integrated Development Environment*, or *IDE* for short. The IDE of Visual Basic Express has been formed from the experiences of many programmers and many other environments, but it will definitely be familiar to anyone who has programmed in Visual Basic before.

To explore the main elements, you should expand and pin several windows and explorers. As you can see in Figure 1-2, to the right of the Welcome page is an area entitled Solution Explorer. In the top-right corner of this area are three small buttons. The middle one is the *pin*, or auto hide, button. When clicked, this tells the IDE to always show the area, or to automatically hide it when it is not needed. Another window that is currently hidden is the toolbox to the left of the Welcome page.

To better describe the environment, and to start setting it up in a way that will be useful to following the examples in this book, move your mouse over the Toolbox tab on the left, and when the IDE automatically expands it, click the pin button to keep it from automatically hiding. Next, create a basic project by clicking on the File menu, selecting New Project, and clicking OK when the New Project dialog window is shown. This will create an empty form and show it in Design view.
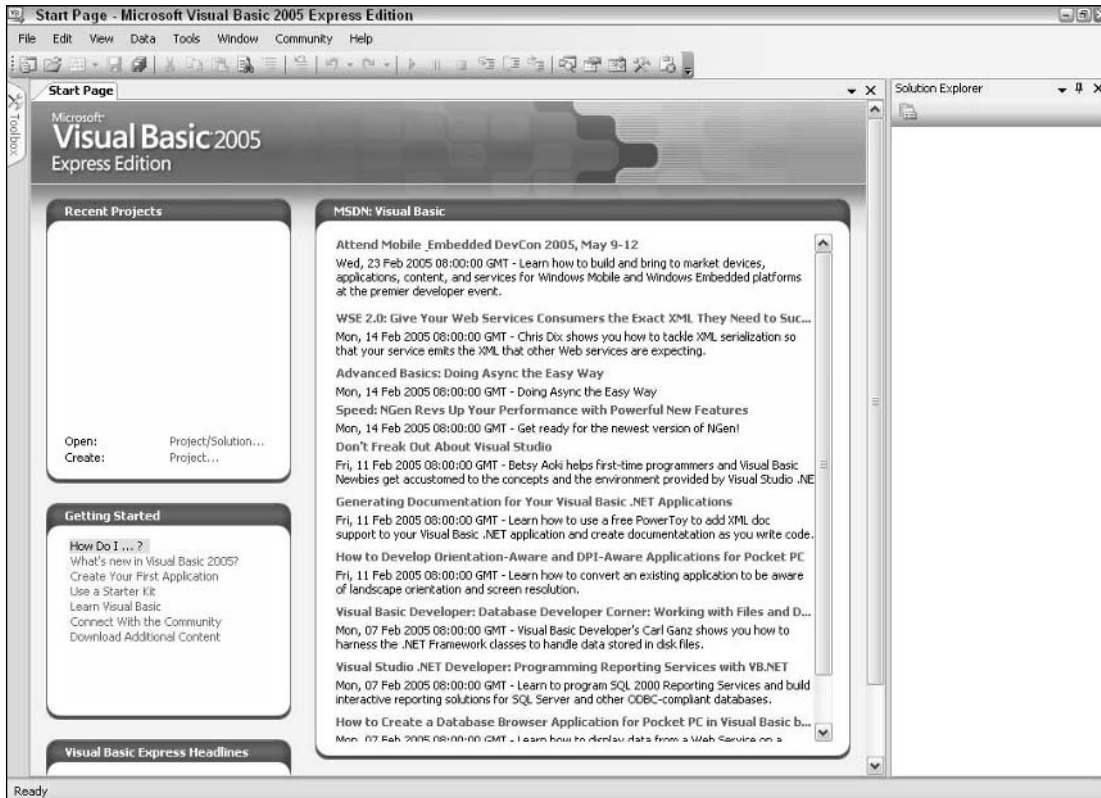
Figure 1-2

To finish setting the scene, double-click on the word Button in the Toolbox window, and the Visual Basic Express IDE will automatically place a button on the form in a default location and with default settings. After you've done all this, the IDE should look like Figure 1-3.
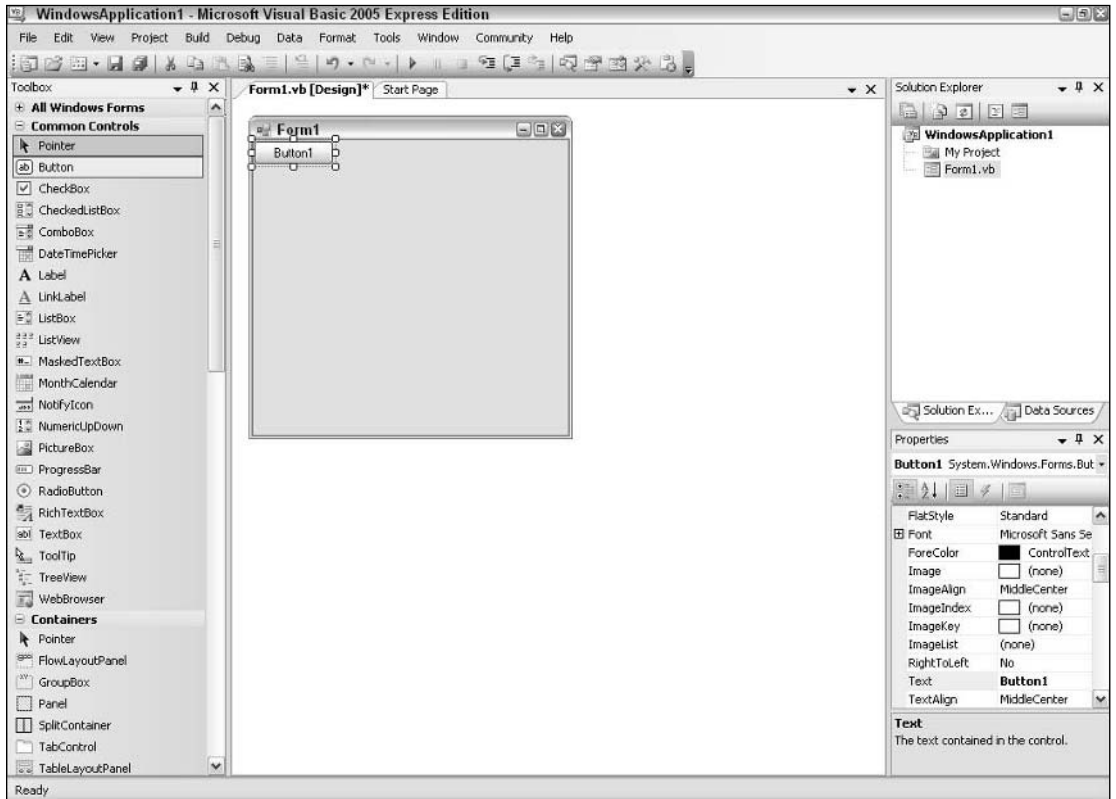
Figure 1-3

The Major Components

Now that the main components of the IDE are visible (and you have even used some!) it's time to tell you what each section does. You should already be familiar with menus and toolbars — they're present in almost every current application. The thing to be aware of in Visual Basic Express is that they're dynamic, and show only the commands that are appropriate to the current context. For example, the Format menu will disappear when you're in a code window, as it doesn't make sense for it to be present when you're writing code. Similarly, the Text Window toolbar won't show when you're designing a form layout.

The next major window is the *Toolbox*. In the next several chapters, you'll use the Toolbox to add various components to your applications, which should give you an idea of what each one does. Every fundamental component you can add to your solution can be found in the Toolbox. To add one to your form layout, you can double-click on it or click-and-drag it to the form.

The objects are grouped into logical sections based on function. By default, you'll find the *Windows Forms* section is expanded and contains many commonly used elements such as buttons and text areas. The other readily available groups deal with data-related components, such as database connections and system components, that give you access to system-level features such as performance monitors and hardware devices.

Moving over to the other side of the main window, you'll find two more essential windows: the *Solution Explorer* and the *Properties window* (both of which are shown in Figure 1-4).
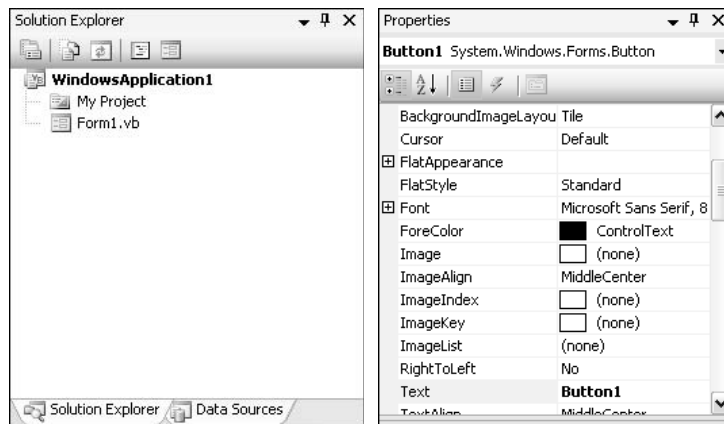
Figure 1-4

❏ The **Solution Explorer** provides you with a way to navigate through your program's structure, with entries for each form, module, and class, along with supporting files such as the application configuration file. The view is structured in a way similar to Windows Explorer, so you should have no problem navigating your way through the program.

❏ The **Properties window** gives you access to the various configurable options available to the currently selected item. This can be a form, a server component, or an individual object (such as the Button object shown in Figure 1-4). By default, the Properties window is organized into categories, but you can click the A–Z button to sort the properties alphabetically instead.

The last major areas to cover are the *Error List* and *Task List* windows at the bottom of the IDE. These two windows will not appear until you have compiled or run an application, but after that point, they will always be present by default:

❏ The **Error List** will be populated with any potential issues with the code and form design of your application. The issues will be broken down into three categories—errors that will stop the program from compiling at all, warnings that indicate a probable runtime error that ought to be investigated before running your program, and informational messages that are purely there for your reference and won't affect the way the program runs.

❏ The **Task List** contains automatically generated tasks, although you can also manually create your own user tasks. You can use this list to keep an eye on what needs to be done, and you can check individual tasks off as you complete them.

Your First Program

You're actually well on the way to creating your first program in Visual Basic 2005 Express. Earlier in the chapter, you created a Windows Application that generated a blank form. On the form, you added a button. To finish the job, you'll need to write a single line of code that will be executed when a user clicks on the button. The following Try It Out walks you through the entire process of creating the project, adding the button to the form, and writing your first line of code.

Try It Out   **Creating Your First Program**

If you didn't create the project in the previous part of this chapter, follow these steps:

1. Start Visual Basic 2005 Express. As mentioned previously, you'll find the link to Visual Basic in your main All Programs list on the Start menu.

2. Create a new Visual Basic project by selecting File ⇨ New Project. This will present you with the New Project window, listing all available project templates (see Figure 1-5).
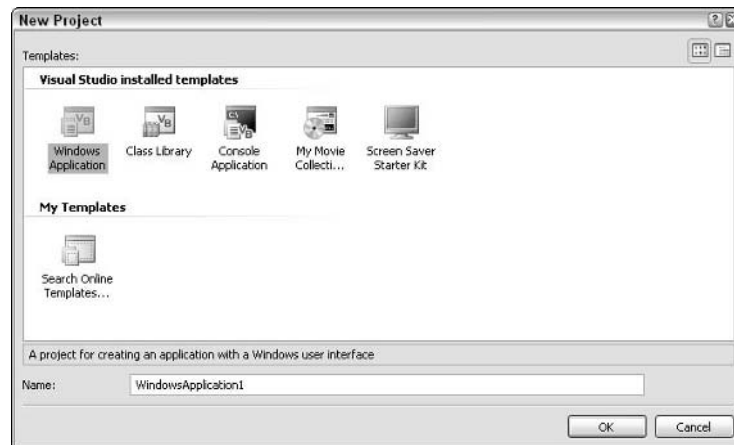
Figure 1-5

By default, Windows Application should be selected. This will create a normal program that runs in Windows. Click OK when you're ready to have the project generated for you.

3. After a moment, you will be presented with a blank form in the center of the IDE. Find the `Button` control in the Toolbox and double-click it to automatically add it to the form in the top-left corner.

4. Select the `Button` object that was added to the form by clicking it once. Locate the `Text` property in the Properties window (it may be easier if you sort it alphabetically) and change it to `Say Hello`. To do this, you should click the right-hand column next to `Text` to access the value (by default it says `Button1`, which is the name of the control).

5. Double-click the button on the form and the IDE will automatically open the code window for this form. It will then create the necessary code to execute your code when the button is clicked, like so:

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

End Sub
```

6. In between the `Private Sub` and `End Sub` lines, write the code `MessageBox.Show("Hello World!")` so that the program appears like this:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
        MessageBox.Show("Hello World!")
End Sub
```

7. Now you can run your program. The easiest way is to simply press the F5 button, but if you find the menus and toolbars easier to use, you'll find the Start command in the Debug menu. Either way, when you run your program, you'll be presented with a simple form with a single button on it. If you click the button, it will display a message box with the words "Hello World!" (as shown in Figure 1-6). Congratulations—you've written your first complete Visual Basic Express program!

Figure 1-6

What you've done is create a Windows Application—a program designed to run on Windows with a base form. You then added a button to it and wrote actual code to generate a message dialog box when the user clicked it.

That Was Too Easy

Yes, I know—that first program seemed a little too easy, didn't it? That you needed to write only one line of code to actually create a program containing a button on a form that produces a message might seem a little crazy, but that's what Visual Basic Express is all about—making life as a programmer incredibly simple.

To show you that this simplicity extends well beyond the age-old Hello World program, I can show you how to create a simple web browser. The intention is to create a form that has a button, a text input area, and a fully functional web browser on it. When the user clicks the button, the web browser will attempt to navigate to the URL entered in the text area.

Try It Out   Your Very Own Web Browser

1. Start a new Windows Application project in the same way you did in the previous Try It Out exercise.

2. Once the blank form is generated, you need to add a `Button` control, along with a `TextBox` control, and a `WebBrowser`. Because you want to be able to see what's on the web page, resize the form to 500 pixels wide by 460 pixels high. To do this, you can select the form in the design window and click and drag the bottom right corner to the desired size, or you can locate the `Size` values in the Properties window. You'll learn more about properties in more detail in subsequent chapters, but for now overwrite the current setting with the value 500, 460.

3. Once the form size is set, click and drag the three controls from the Toolbox onto the form and then resize them — again using either the click-and-drag method or setting the values directly in the Properties window — so they are laid out as shown in Figure 1-7. You should also set the `Text` property of the button control to the word `Go`.

    You'll notice that as you click and drag controls to move them about or to resize them, small helper lines appear. These lines indicate ideal proximity to the edges of the form or to other controls. In some cases, you'll also see small blue alignment lines that make aligning controls with each other easy.
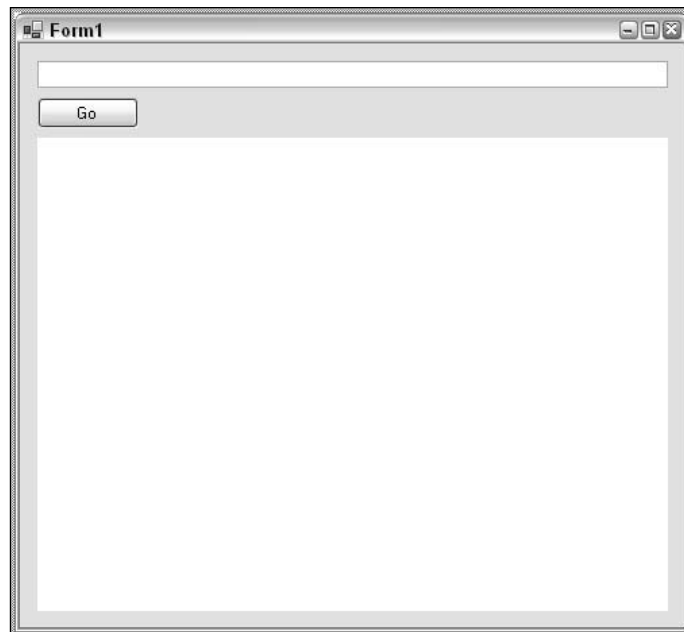
Figure 1-7

4. Now that you're done with design, all you need to do is add the code to make the button react when clicked. Just as you did in the previous Try It Out, double-click on the button to generate the code necessary to hook into the click of the button.

5. The only thing you need to do in the code is tell the `WebBrowser` control to go to the URL specified in the `TextBox` control. The properties you see in the Properties window are also accessible in code. The way you access these properties is by specifying the name of the control followed by a period (.) and then the name of the property. Methods are functions connected to an object, and they execute a certain task. In this case, you need the `Text` property of the `TextBox` control to get the URL text, and the `Navigate` method of the `WebBrowser` control to tell it to go to the URL. This is all achieved with the following line of code:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
        WebBrowser1.Navigate(TextBox1.Text)
End Sub
```

6. Run the program by pressing F5 or selecting the Start command on the Debug menu. When the form is displayed, type a URL in the text area, such as `http://www.wrox.com`, and then click the Go button. After a moment, the web browser area of the form will be populated with the web page associated with the URL, as shown in Figure 1-8.

Figure 1-8

As you can see, creating what appears to be a fairly complex program is made simple in Visual Basic 2005 Express. The controls used to create this program, along with the techniques known as *method* and *property access* in code, are discussed in the next few chapters.

Summary

Creating programs using Visual Basic 2005 Express is an immensely rewarding process. When you need to write only a couple of lines of code to achieve a feature-rich solution, it frees you to think of more complex solutions and helps you harness the power of Windows in ways that previously would be too difficult to contemplate.

In this chapter, you learned to do the following:

❏ Install Visual Basic Express, SQL Server Express, and the associated documentation

❏ Create a simple application that says "Hello World," and another that can browse a website

In the next chapter, you'll find out about starter kits and wizards—more features of Visual Basic Express that make your programming life easier. Along with these wizards, you'll learn about some core programming concepts such as controls, classes, methods, and properties that are essential to programming in any language.

Exercises

1. **Installing Visual Web Developer 2005 Express Edition:** To create applications that run on the Internet, you can still use Visual Basic 2005 as a language, but you will need to install Visual Web Developer 2005 Express Edition. The method for installing Web Developer Express is exactly the same as what has been outlined here, but it will install Web Developer instead of Visual Basic. If you have already installed Visual Basic Express, you'll find that the Web Developer installation process does not include options for MSDN or SQL Server, as it automatically detects that they are already present on your system.

2. **Customizing the Browser Application:** Extend your web browser program so you can both go back to the previous web page you visited and navigate to the default home page of Internet Explorer. You'll need to use two more methods of the `WebBrowser` control—`GoHome` and `GoBack`.

2

Why Do All That Work?

Other programming languages require you to create everything you'll need using code. While that might appear to give you more control over every aspect of your program, that perception is often wrong when it comes to modern languages such as Visual Basic 2005 Express.

Rather than write code, the development environments included as part of the whole package along with the language enable the programmer to click and drag user interface elements around, provide quick access to the various components and their properties, and format the actual code portions of the application in a way that makes creating new subroutines relatively painless.

In addition to these fundamental capabilities, Visual Basic 2005 Express takes it a step further with wizards and starter kits. Both of these walk you through various options and then generate large sections of code designed to do what you require without you needing to know how it was done.

In this chapter, you learn about the following:

❑   The programming fundamentals of an object-oriented world

❑   How to use starter kits

❑   What wizards are and how to take advantage of them

❑   How to set up the environment and options to customize your experience

Object-Oriented Programming 101

When it comes down to it, in order to understand any programming language, you actually need to know how the code fits together. The majority of this chapter walks you through the wonderful features of Visual Basic Express that reduce the amount of code you have to write, sometimes enormously; and along the way, you'll benefit from a basic understanding of how the language works in general.

Visual Basic Express is an object-oriented programming language. What this means is that everything revolves around individual *objects* and how they interact with each other and the rest of the world. A real-world example might be an employee of a company. The employee has a name, date

of birth, and a salary. The employee can start work, finish work, and perform a variety of functions in between. One such function might be to deliver the results of a job to another employee. Another function might be leaving the office if the fire alarm sounds.

All of these functions and descriptive elements about the employee help define it as an object. In programming terms, the employee could be defined as an individual object, with its name, date of birth, and salary being stored as properties, and the various functions defined as methods and events. Other employees can interact with this one using these methods.

It gets a little more involved than that, however. Objects can house other objects. Continuing the employee example, each department in the company could be defined as an object with specific properties and functions. Within the department are a number of employees. This translates directly to the programming concept — a department object can have a collection of employee objects and these employee objects can relate to each other through the functions they expose to each other, or even cross department boundaries and talk to an employee belonging to another department. The only caveat to this is that you as the programmer must implement the code that makes this interaction possible.

In other words, everything in Visual Basic Express is an object, and all code is written to make the objects interact with each other. Every object has a collection of descriptive elements called *properties*. A property is something that defines an aspect of the object, such as its name, color, or size. In addition to properties, objects can have methods. *Methods* are subroutines that perform a section of code. They can do pretty much anything you want them to and are usually defined in two groups — *internal functions* that are called only by other parts of the object, and *external methods* that are invoked by other objects. Finally, objects include events. *Events* are special method subroutines that are connected to set circumstances. For example, a button control could be clicked by the user, and you'll usually want to know about that when it occurs so you can respond accordingly — for this purpose, the `Click` event is exposed by the `Button` object.

All these elements — properties, methods, and events — combine to form the structure of an object. In programming code, this structure is defined and known as a *class*. When you need an instance of a class, you create an object based on the class as a template. This method of creating a class definition enables you to easily create multiple objects of the same kind. In the employee example, you would define an `Employee` class and define the various properties, events, and methods in it. Then you would create an `Employee` object for each employee you want to handle, and it would automatically receive every element you defined in the class.

When creating an object-oriented program in Visual Basic Express, you don't need to know much more than that. All the properties belonging to a particular object are accessible in the Properties window or in the code, by typing the object name, followed by a period (.), followed by the property name. Methods are called using a similar method.

Events work slightly differently, as you need to tell the code how to handle the event when it occurs. You do this by "handling" the event with a function defined in the containing object that owns the object that has raised the event. In the company example outlined so far, there may be a `Building` object that has a `FireAlarmSounded` event. This event is triggered whenever there is a fire and the alarm goes off. The `Employee` object would reference the `Building` object and have a function defined that handles the `Building`'s `FireAlarmSounded` event. This might appear like so:

```
Private Sub FireAlarmSounded Handles myBuilding.FireAlarmSounded
    ExitTheBuilding
End Sub
```

Quick Reference Glossary

As you read through *Wrox's Visual Basic 2005 Express Edition Starter Kit*, you may find yourself encountering the following programming terms. Use the definitions provided here as a cheat sheet to help remember what they all mean:

Class — The definition of something to be used in the programming. The class defines an object's makeup, while an object implements a class structure for an individual instance.

Object — A discrete piece of data that is defined by a class, including public elements and internal data

Property — A descriptive element of an object. Properties are defined in the class and normally describe the object in some way. For example, name, date of birth, and phone number are all properties of a `Person` object.

Method — A function belonging to an object that can be called by other parts of the program. Usually, methods will perform an action or set of actions against the object. A `Person` object may have a `GoToSleep` method, which puts it into a sleep state.

Event — A predefined occurrence that the object knows about and can communicate to other parts of a program. Events are intercepted by event handlers and can convey to the recipient code information that is necessary for it to function properly. The `Person` object could have a `GoneToSleep` event that is raised whenever the object's sleep state is activated.

Here are some other basic programming terms you'll need to remember as you progress through this book:

Function — A subroutine that can accept pieces of data as input and return another data element as output.

Variable — A special kind of object that contains a single piece of data, such as some text or a number. Variables store this information so you can retrieve it later.

Control — A special kind of object that you can put on a window or form that behaves in a specific manner. Examples of controls are `Buttons`, `TextBoxes`, and `ListBoxes`.

The final thing to note at this point is that you can refer to objects within objects as well. If the `Department` object needed to know the name of the building in which a particular employee was situated, it might get that information by concatenating the `Employee` object with the `Building` object and the `Building` object's name, all joined with periods, as shown in the following line of code:

```
sNameOfBuilding = myEmployee.Building.Name
```

Now that the essential theory work is done, you can have some more fun — this time with starter kits.

Starting Out Right

The people at Microsoft have outdone themselves this time around. Normally, programming languages come with an Integrated Development Environment (IDE), a bunch of prebuilt controls that can be dragged and dropped onto a form, and a number of wizards to automate certain tasks. Visual Basic 2005

Express does do all of that, but there's an extra feature that sets a new standard for rapid development — the starter kit.

The main starter kit is the DVD Movie Collection Starter Kit. This project template will automatically create your main form, complete with all necessary controls to create and maintain a simple DVD collection. In addition, it comes with web access calls to retrieve information from Amazon.com and a database setup so all of the information can be retained between program executions.

Once you have the base application generated by the starter kit, you can customize it as much as you need to — everything used to create the program is accessible by you when it is complete. This enables you to check out other coding styles, the programming structure of a working application, and some best practices for project organization.

In the next Try It Out sequence, you'll create a DVD Movie Collection application using the starter kit, and then look at several elements of the program to see how they work.

Try It Out   Using Starter Kits

1. Start Visual Basic 2005 Express and select File ➪ New Project.

2. Starter kits appear right alongside normal project templates, so you create a new application based on one just like any other project type. Find and select the My Movie Collection Starter Kit template. Type a suitable name for the project — you're going to use this project as part of the larger application later, so if you would like to be prepared for the later chapters, name the project `MyOrganizerMovies`. Once you've entered the name, click the OK button to create the starter kit project.

3. After a moment, you will be presented with the main IDE window, which is populated with documentation related to how to use the starter kit; and the Solution Explorer window on the right-hand side, which contains a hierarchical list of all the modules, forms, controls, and objects that form part of the project.

4. Run the application by selecting Debug ➪ Start or by pressing the F5 key. When Visual Basic has completed building the application's executable files, it will display the main form (see Figure 2-1).

5. You'll notice two buttons on the top of the form — View DVDs and Search Online. The screen you can see initially is the View DVDs form, where you can scroll through all the DVDs in your collection. Add a title manually by clicking the Add Title button and entering the details on the right-hand side.

6. Click the Search Online button. The main area of the form will switch over to a search page. Here you can search Amazon's website via their web service to look for any movies that match the words you entered, which if found will be presented in a list. Unfortunately, the version of the starter kit that ships with Visual Basic Express doesn't come with the necessary code behind the user interface to connect to the web, so all you can do is look at the form (which appears in Figure 2-2).
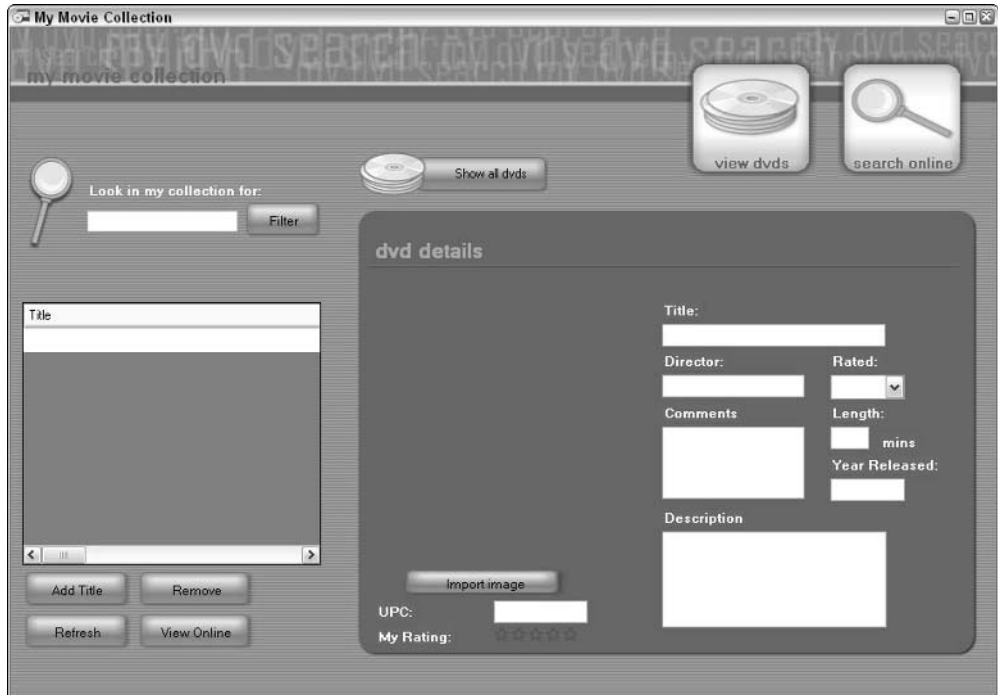
Figure 2-1

Figure 2-2

7. You can see here that you can type in keywords and click the Search button. At that point, the web-enabled version of this starter kit accesses Amazon's website and downloads movies that match the criteria you entered. You can browse through the list (the list is on the left and the details for the selected movie are on the right) and then add the correct DVD to the database with the Add to Collection button.

8. Return to the View DVDs page and add more DVDs manually. Once you're done, click the Close button in the top-right corner to terminate the application and return to Visual Basic Express.

To get the web-enabled version of this project, you'll need to go to the Starter Kit web page on Microsoft's developer site. The URL is `http://lab.msdn.microsoft.com/vs2005/downloads/ starterkits/`. Locate the Amazon-Enabled Movie Collection Starter Kit section of the page and download the Starter Kit for Visual Basic.

Once the .vsi file (a special file type for installing add-ins to Visual Basic Express) is downloaded to your computer, locate it in Windows Explorer and double-click it to start the installation. Visual Basic Express will prompt you for confirmation and then install the new template for the web-enabled Starter Kit.

Restart Visual Basic Express and create a new project. You'll find the new template in the My Templates section with a label of My Movie Collection Starter Kit (Download). When you use this template instead of the one supplied as part of the normal installation of Visual Basic Express, you will be able to perform the online functions, such as searching Amazon.com.

How It Works

Did you notice what you just did? You created a full-blown application that includes a database, custom-built controls, formatted backgrounds, and buttons; and to top it off, if you use the web-enabled version available from Microsoft's website, the system actually accesses the web and communicates with a real web service. And what did you actually do to create all of this magic? Nothing more than a couple of clicks of your mouse!

Even better, the magic doesn't stop there. As mentioned earlier, starter kits not only give you a great head start in creating whole programs like this one, they also give you full access to maintain and modify (and potentially break if you're not careful) the application to suit your needs.

If you delve into the Visual Basic Express development environment, you'll find that all of the features used in the Try It Out are easily accessible. First take a look at the main form in Design view. Locate the `MainForm.vb` file in the Solution Explorer and double-click it. When the Design view is shown, you'll see that the form itself is quite empty — it has the two navigation buttons on the side, but the main part of the page is empty except for a blank object called `TargetPanel`. *Panels* are special objects that are often used to design the layout of a form, and are kept as placeholders for other objects.

The View DVDs and Search Online buttons each load a different custom-built control into the `TargetPanel`. These custom-built controls are where the remainder of the user interface design can be found. Double-click the `ListDetails.vb` entry in the Solution Explorer to show the Design view of the control. Here you can see and modify the various controls that make up the View DVDs page.

You can access the code as easily. Locate the `SearchOnline.vb` entry in the Solution Explorer and right-click it to bring up its context menu. Select View Code to show the Visual Basic code that drives the various functions and events for this control.

The code generated by a starter kit often contains best practices and more efficient methods of achieving the result you're after, so it's a good idea to take a look at it.

As an example, when the Search button is clicked, the `SearchButton_Click` subroutine is executed, which in turn simply calls a privately accessible subroutine called PerformSearch — the code for this appears as follows:

```
Private Sub PerformSearch()
    'object responsible for containing dvd search results
    Dim searchResults As New List(Of DVD)

    'simple wrapper object responsible for handling requests and responses
    'from the Amazon.com Web service
    Dim amazonService As New SimpleAmazonWS

    'show hour glass during the search to tell users that work is being done
    Me.Cursor = Cursors.WaitCursor

    Try
        'request search results from the Web service passing in the user's search
        'criteria
        searchResults = amazonService.SearchDVDs(Me.SearchTextBox.Text)

        'data bind the search results to the form UI
        Me.DVDBindingSource.DataSource = searchResults
    Catch ex As Exception
        MsgBox(String.Format("There was a problem connecting to the Web service. " _
          " Please verify that you are connected to the Internet.  Additional " _
          "details: {0}", ex.Message))
        My.Application.Log.WriteException(ex)
    End Try

    'set cursor back to the default now that work is done
    Me.Cursor = Cursors.Default

    'tell the user how many results were found.  Use String.Format feature to concat
    'strings in a Localization-friendly way
    Me.Label2.Text = String.Format("{0} results found.  ", _
        searchResults.Count.ToString)
End Sub
```

This code is self-describing through the use of meaningful names for variables and well-placed comments that communicate less obvious commands. It first creates an empty list of DVDs along with a copy of the Amazon web service object. The routine then attempts to retrieve the list of DVDs from Amazon using the search text that was entered. Once it obtains the list, it passes it over to the database objects so they can populate the rest of the control. If there is a problem, a message dialog will be displayed for the user.

To see how easy it is to modify the code to suit your own requirements, follow along with this Try It Out to change some code along with some of the user interface design.

Try It Out   **Modifying Starter Kit Projects**

1. Return to the `MainForm` in Design view (double-click on the `MainForm.vb` entry in the Solution Explorer list). You'll change the caption of the form to better suit the rest of the application you'll be creating.

2. Click on the caption bar to select the form itself, and then locate the Text entry in the Properties window. If the Properties window is not visible, press the F4 key to display it first.

3. Highlight the current text and replace it with `Personal Organizer – DVDs`.

4. Locate the My Project entry in the Solution Explorer and double-click it to open the project's properties. This special set of pages (shown in Figure 2-3) contains projectwide settings, including how to compile the application. Click the Resources tab on the left to display the list of currently included images.

Note that when you are viewing the Resources tab in the Project properties, you can also access other resource types such as icons and audio files.
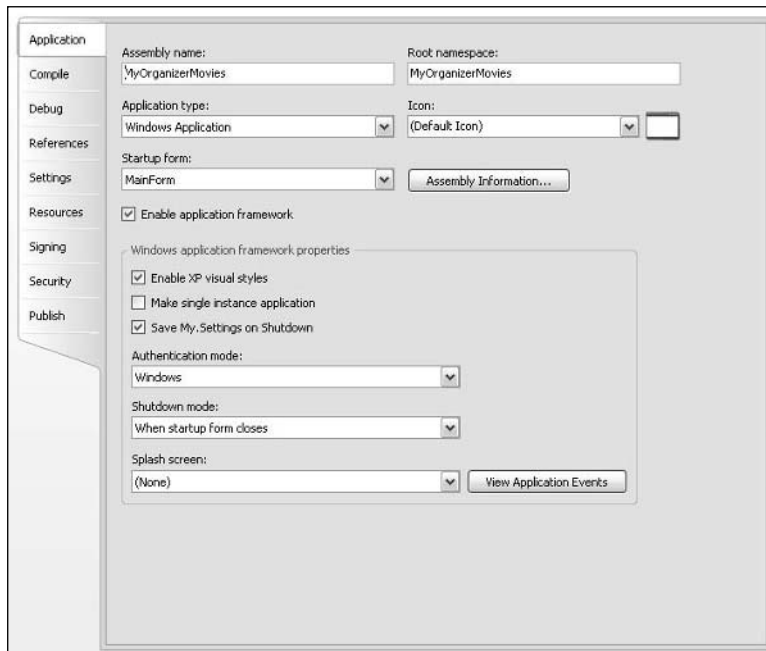
Figure 2-3

5. Click the small down arrow next to the Add Resource button and select Existing File. Locate a picture file on your computer that you would like to use as a background. I chose the `winnt.bmp` file found in the main Windows directory because it is commonly found on most systems. Once you have found the file, click the Open button and Visual Basic Express will import the file into the Resource library for your project.

The image is now in the Resource library, but before you can use it in the rest of the program, you'll need to save the Resources section. Do this by selecting File ⇨ Save Selected Items.

6. Now that you have the new image, you'll want to tell the program to use it as a background image. Open the `ListDetails` control by double-clicking the `ListDetails.vb` entry in the Solution Explorer. Click anywhere on the background of the form to make sure the Properties window is referring to the form and not any of the objects on it, and locate the `BackgroundImage` property in the Properties window.

7. Click the ellipsis button in the `BackgroundImage` property, and Visual Basic Express will display a dialog window that enables you to change the image to another one in the Resources library. Scroll through the Project resource file list until you find the `winnt` entry and select it. Click OK to save that image as the background image for the form.

8. Select the `BackgroundImageLayout` property (which is listed immediately below the `BackgroundImage` property you just changed) and, using the drop-down list to choose from the available options, select Stretch so that the image is resized to fit the form size.

9. Run the program again. Notice how the caption of the window has changed to your new title, and the background of the `ListDetails` area has been modified so that it shows the new image (see Figure 2-4). Click the Search Online button to confirm that the background of that control has remained unaltered.

Figure 2-4

10. Once you're satisfied, end the program, and save the project in Visual Basic Express by selecting the File ⇨ Save All command. Visual Basic Express will prompt you for a location for your project. Choose somewhere you'll remember later, as you'll need to call this application from the main Personal Organizer application you will build in the rest of the book.

In just a few short minutes, you updated an application by changing a property on the form, adding a new resource to the project, and then referencing that resource in the design of a control.

Wizards, Too

Starter kits aren't the only aids you have to remove some of the burden of actually writing code — they're just the most glamorous. Their older and humbler cousins, wizards, have been around for a long time and aren't restricted just to the programming world. When you create a new account in Outlook or set up your home network, you'll most likely use a wizard to do so. In a nutshell, a wizard is a multistep

process that walks you through a (typically difficult) task. At the end of the process, the wizard takes the information it has collected from you and produces the desired result. In Visual Basic Express, this result is usually lovely code ready to use.

In fact, in Chapter 12, you'll create your own wizard as part of the Personal Organizer application to export data from your database. The next Try It Out shows you a very popular and useful wizard that is included with Visual Basic Express — the Visual Basic Upgrade Wizard. This wizard is automatically fired up if you attempt to open a Visual Basic 6 project in Visual Basic Express, and it attempts to auto-matically create a .NET version of the project for use in Visual Basic Express.

This project, and a number of others throughout the book, need the code download available from www.wrox.com *for this book. Refer to the Introduction or Appendix A to find out how to locate and download this code.*

Try It Out   Using a Wizard

1. Start Visual Basic 2005 Express and select the File ➪ Open Project command.

2. When the Open Project dialog window appears, browse to the location where you extracted the code downloaded from Wrox's site and find the Chapter 2/VB6Calc folder. In here you will find a file called Project1.vbp. Select this file and click the Open button.

3. Visual Basic Express will detect that the Visual Basic project was created in Visual Basic 6 and start the Visual Basic Upgrade Wizard.

4. The wizard goes through five steps. At each window, simply click Next (you can optionally change the location of where the new project will be built on page 3 — see Figure 2-5). When you reach the last page, the wizard takes over and begins to build the new project by analyzing the forms and modules in the old project and converting the various design elements and code subroutines to run in Visual Basic Express.
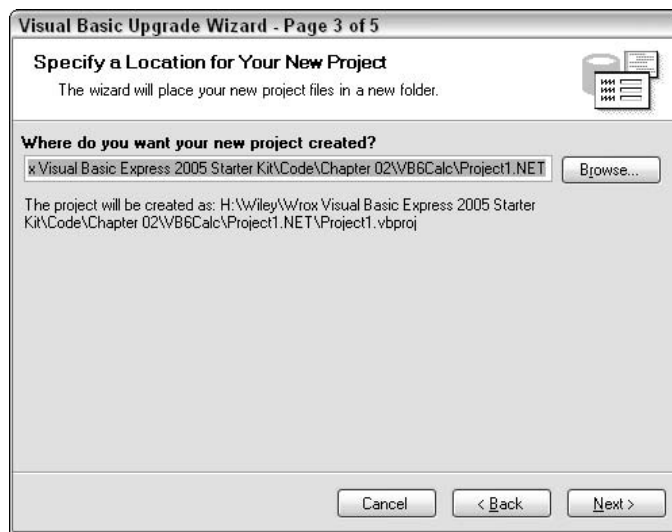
Figure 2-5

5. When the wizard is complete, it will close the wizard form and display an upgrade report. If the upgrade report is not shown by default, it usually means the upgrade worked completely. You can still view the report by locating and opening the `_UpgradeReport.htm` file in the Solution Explorer.

This sample project should upgrade and build without any errors. See the sidebar "Upgrading Visual Basic" for more information on upgrading from Visual Basic 6.

Upgrading Visual Basic

Visual Basic 2005 Express is part of the latest release of Visual Basic from Microsoft and can automatically convert projects developed in previous versions of Visual Basic, often with minimal human intervention required.

The Visual Basic Upgrade Wizard does an enormous amount of work for you by converting the old language syntax to the new style of doing things, and replaces various controls and classes as much as it can.

The project used as an example in this chapter is cleanly converted completely — all of the controls are converted to their Visual Basic 2005 Express equivalents, and none of the underlying code needs to be changed other than event handler connections (which you'll find out about in the second part of this book).

However, many issues can arise when upgrading older Visual Basic 6 projects, and many require unique solutions to deal with the problems that the converter has encountered. These issues fall into two categories — the known and the unknown.

The known issues are problems that the Upgrade Wizard encountered as it converted the code and design to the new format. For every single issue that the Upgrade Wizard finds, it will insert comments in the code to highlight the problem as well as a new entry in the Task List. In both locations, you will also find a link to the appropriate place in the Microsoft help documentation that describes why the Upgrade Wizard was unable to convert the code and what steps you can do to fix the problem yourself.

While these can be a pain to fix, it is the unknown problems that are more of a concern. These are caused by the subtle differences between the ways in which the two different versions of Visual Basic work, and they are not found by the Upgrade Wizard. You will not encounter many of these, and rather than being strictly language-specific problems, they are usually related to the way the original code was written.

Because these issues don't cause compilation errors or show up in the Upgrade Wizard process, they won't be seen until the application is running. Admittedly, they will not occur frequently, but because there is always the potential for this kind of logical error, you should test any project you've upgraded from Visual Basic 6 thoroughly before changing it further.

On the other end of the spectrum, it's worth noting at this point that any project you create in Visual Basic 2005 Express is automatically compatible with Visual Basic 2005 as well. Therefore, if you've been developing applications in Visual Basic 2005 Express but then upgrade to the full version of Visual Basic to take advantage of the enterprise and web features found in that product, you can be sure that your work will translate seamlessly.

You reverse isn't necessarily true, however. If the Visual Basic 2005 project contains references or code constructs that are available only in the full version, you won't be able to open it in Visual Basic 2005 Express without encountering errors.

Everything Is Optional

Besides the starter kits and wizards, Visual Basic Express has other ways of making your experience in programming more enjoyable. While the standard settings that are installed with Visual Basic 2005 Express are pretty good, there's always the chance that they won't suit your own personal style. Fortunately, Microsoft has outdone itself in creating ways to customize the interface and your experience in using the IDE. As mentioned in the last chapter, menus and toolbars are dynamic depending on the context of your situation. However, if you would like to show (or hide) a particular toolbar that isn't part of the default settings, you can choose to show it using the Customize command found in the Tools menu.

From here, you can select which toolbars should be shown in the current situation, along with which commands are to be accessible from each toolbar. This level of customization should be familiar to you if you've used other Microsoft products such as Word or Excel. You can create your own toolbars, and add, delete, or move commands around in the menus to suit your own personal style of working. Moreover, the IDE can be changed in a number of other ways that will likely be new to you.

Not only can the various windows and panels that are situated around the main editing space be automatically displayed and hidden as described in the last chapter, they can also be moved to a more convenient location. To aid you in the process, as you drag one of these windows around the design surface, snap and alignment icons will appear.

In Figure 2-6, the Code Definition window is being dragged over the main editor area. The Visual Basic Express IDE pops up snap buttons to move it automatically to one of the four sides of the editor space, or to the very edge of the entire window. As it is dragged over another window, the icons change to enable it to piggyback the space used in a tabbed display.

To customize the IDE further, Visual Basic Express has an extensive Options dialog. To display the Options window, use the Tools ⇨ Options menu command. In here you can affect the view by something as simple as changing the font to a more legible typeface, showing a grid to more easily align controls when editing a form in Design view, or changing the style of the core IDE from the tabbed environment to a more recognizable MDI layout (MDI stands for multiple document interface and is common in applications such as Microsoft Excel).

Some programmers prefer to use line numbers in their code, and Visual Basic Express allows for that kind of customizing, too, in the default view of the Options window (see Figure 2-7). However, numerous configurable settings are hidden in the normal view. Clicking the Show All Settings checkbox at the bottom of the Options dialog window will display these additional settings.
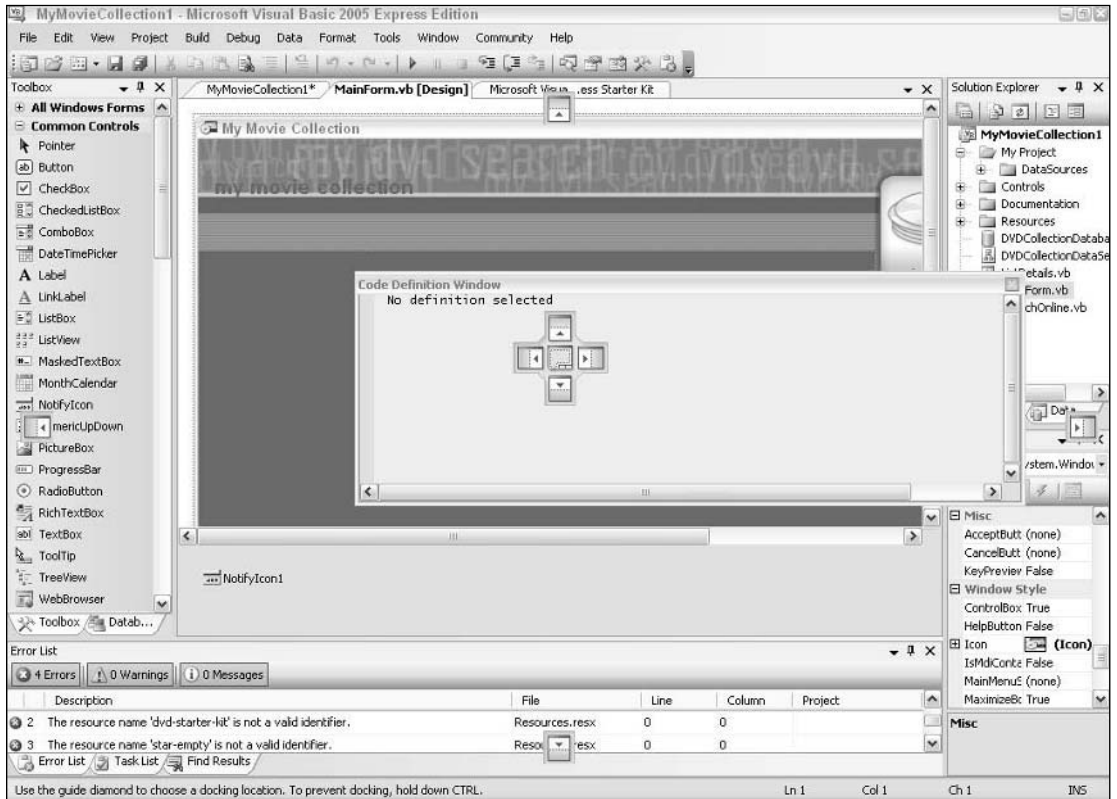
Figure 2-6
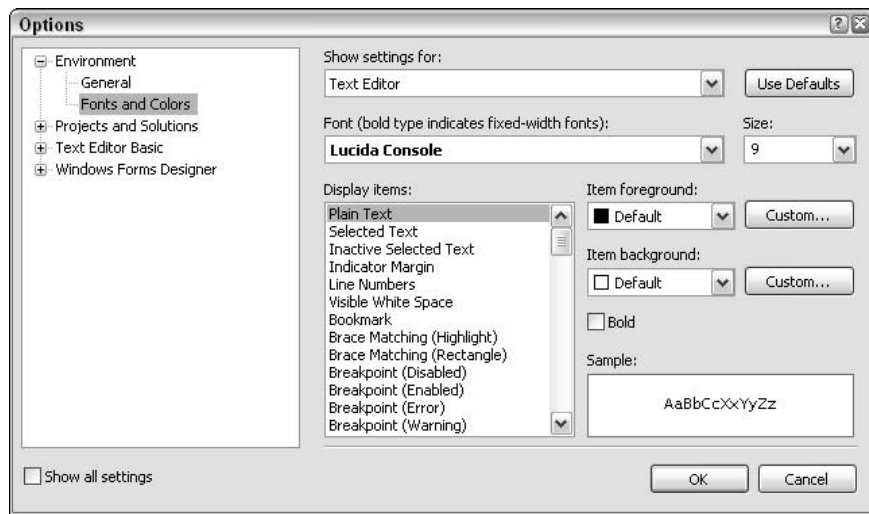
Figure 2-7

In the next Try It Out, you'll set a variety of options designed to make your experience with Visual Basic Express both more efficient and safer from unexpected errors.

Try It Out   Customizing the Options

1. Start Visual Basic 2005 Express and bring up the Options dialog by selecting Tools ➪ Options. By default, you should see the basic view (similar to Figure 2-7).

2. Change the Editor Font from `Courier New` to `Lucida Console` (on some systems, this might already be selected, and on others the default font will be Courier New). Lucida Console is a more modern font and is usually easier to read on higher-resolution displays. If you think the size is a little small, you can increase it using the combo box next to the font name list.

3. Expand the Projects and Solutions group and then select VB Defaults to display default settings for any new projects you create. Turn Option Strict on by checking the box. By default, Option Explicit is turned on, which means that any variables you use in your code must first be declared. If they're not, the program will not compile.

    While this is great protection from unexpected results, Option Strict is even better. With Option Strict, Visual Basic Express will not allow you to set one variable from another variable if they are different types. This is known as *implicit type conversion* and is a common source of errors.

4. Check the Show all settings box to display all of the available options. This gives you access to settings that are otherwise hidden from view.

5. Expand the Projects and Solutions section and first click the General set of options. In this area, you'll find options related to creating solutions and what Visual Basic Express will do when you create and compile them. Set the Visual Studio projects location to a folder that you will be able to find later. While you can overwrite this as you create each new project, it's handy to set this to a default location so you don't have to keep on browsing to find it.

6. Select the Build and Run set of options. Review the Before Building option and ensure it is set to one of the Save options. There's nothing worse than ending your programming session and forgetting to save the edited files.

7. Click the OK button to save the changes you've made to the settings of Visual Basic Express.

Your Visual Basic Express environment is now set up in a way that will ensure you have cleaner programs (that is, less bugs) and easier code to follow.

It's All There in the Documentation

The last aid in ensuring that your experience with Visual Basic Express is as enjoyable as possible is the extensive documentation that comes with the development environment. Not only do you get explanations of every control and every class in the .NET Framework, but also you're provided with extensive examples as well.

Visual Basic Express comes with a new form of the MSDN[1] library. It incorporates a redesigned search engine that helps you identify the topics that are best suited to your needs. This contrasts with the old MSDN search capability, which would often return hundreds of obscure results that hid the one or two that actually answered your query.

---
[1] MSDN stands for Microsoft Developer Network and represents a number of things depending on the context. In this book, MSDN refers to the MSDN library — the documentation that accompanies Visual Basic Express — unless stated otherwise.

In Figure 2-8, the user has searched for `BackgroundImage` (the property you changed in the DVD Collection project) with a filter of Visual Basic. Each result is listed with a brief paragraph and a set of icons representing the technologies covered by the article. In addition, the search results provide summaries of the articles that can be found on the MSDN website and the Code Wise Community.

Besides the normal table of contents on the left and the much improved search engine, the new documentation application also comes with a special How Do I section. This area provides quick links to common programming tasks, separated into intuitive categories. To access this enormous set of help documentation, simply press F1 anywhere within the Visual Basic Express development environment, or select from the various menu items in the Help menu.
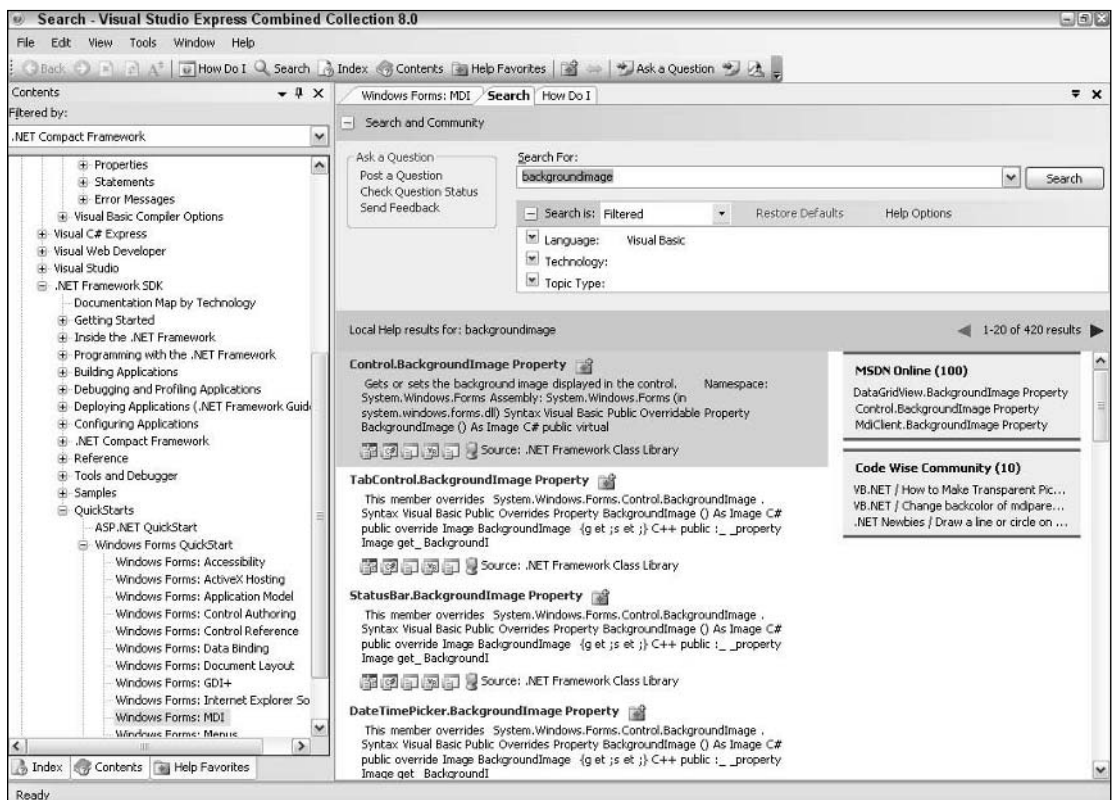
Figure 2-8

Summary

Although there are hardcore programmers out there who insist on writing every single line of code to achieve their goals, you can see from this chapter that getting some help from the development environment can make you a lot more efficient in reaching those same goals. In fact, using a combination of options and IDE customizations to make the environment suit your own style of programming, coupled with the use of starter kits, wizards, and the examples found in the documentation, you've got the best head start on creating your own applications than you could ever imagine.

In this chapter you learned to do the following:

❑   Understand the concepts behind object-oriented programming including classes, methods, properties, and events

❑   Create an entire application simply by using a starter kit

❑   Use the Visual Basic Upgrade Wizard to convert a Visual Basic 6 application

❑   Customize the environment to suit your own personal taste

Exercises

1.   **Customize the DVD Collection application:** Re-open your `MyOrganizerMovies` project and change the images for the View DVDs and Search Online buttons. You'll need to set three properties for each in the Properties window — `NormalImage`, `HoverImage`, `PressedImage` — and you will need to edit the code so that the proper Resource objects are used.

2.   Look up the documentation for an example of how to use the `BackgroundImage` property of a control.

3

Using Databases

One fundamental requirement of most applications is a way to store the information that is processed. The program may need to know things before it can do its job. Alternatively, you might need to keep track of data in between runs. Another possibility is that the program needs to save the information generated while it was executing so another application can use it.

Regardless of the need, you have several ways of keeping track of the information a program uses. Database technology has been around for almost as long as computing, and fortunately for Visual Basic Express users, the language and development environment come with a number of tools that make it easy to use. In fact, using a database to store information is so straightforward in Visual Basic Express that you might find yourself using databases instead of alternatives such as the Windows Registry or normal files that traditionally have been easier to access.

In this chapter, you learn about the following:

❏   The database technologies that Visual Basic Express supports by default

❏   Creating and editing a database

❏   Adding databases to a Visual Basic Express project

SQL Server Express

Microsoft has had a long history with database technology. On the Windows desktop, they have had at least two different database technologies available for quite a few years now:

❏   **Microsoft Access** is a permanent part of the Microsoft Office suite that enables you to build not only some fairly complex database definitions but also forms, queries, and other components in order to be able to create whole applications that use Access as their driver.

❏   **SQL Server** is the Windows server database that is used for high-end, robust database solutions. Previously, SQL Server needed to run on a server operating system, and because of this, it includes a number of advanced technologies that enable it to run with a much better fail-safe approach than Access.

A few years ago, Microsoft decided to release a product entitled MSDE for use on desktop systems. MSDE stands for Microsoft SQL Desktop Edition or Microsoft Database Engine, depending on who you talk to, but either way it represents the same thing: a scaled-down version of SQL Server to enable developers to build those same robust and performance-based databound applications for standalone desktops.

One problem with MSDE is that not too many people know about it; and because the name is an acronym, it isn't clear what it's for and how it's tied to SQL Server. That issue won't be around for much longer because MSDE's replacement, now available, has support built directly into Visual Basic Express — and this new database product is called SQL Server 2005 Express.

Microsoft worked on the latest version of the full server product, SQL Server 2005, for a very long time (the last version was SQL Server 2000!) and decided to release its scaled-down version in the Express range to enable programmers creating applications on the Windows desktop to use the latest database technology.

SQL Server Express is freely available and is included in the installation of Visual Basic Express (as shown in Chapter 1). It uses a simplified management environment that borrows from the functionality found in the main SQL Server 2005 environment, including a Computer Manager for checking the different services relating to SQL on the machine, and the Express Manager for maintaining the individual databases registered in the SQL Server engine.

However, as you'll see later in this chapter, Visual Basic Express has all the tools you need to create and maintain the databases for your applications built right into its own development environment. And SQL Server Express has borrowed a leaf from the Microsoft Access book, storing each database in its own easily accessible file, which can then be easily deployed as part of your application.

SQL Server 2005 Express uses the same engine as the server-based SQL Server 2005, with ADO.NET support (the database component of .NET and Visual Basic Express), Transact-SQL (the normal language to interact with SQL data), and a SQL Native Client. In fact, it has only the following differences when compared to the full version:

❑   Databases can be a maximum of 4GB in size.

❑   The internal buffers can use only 1GB of memory.

❑   It runs on only one CPU (that is, it does not take advantage of dual-processor technology or spanned computing).

❑   It does not have any of the enterprise features, such as business intelligence.

None of these restrictions prevents SQL Server Express from functioning as a web or database server engine, and in fact it can be used in either of these scenarios. However, the main purpose for it is exactly what this book is about — easily creating applications that run on a Windows desktop PC in a stand-alone environment.

Data to Database

Databases store information in a structured fashion. SQL Server Express is what is known as a *relational database*, meaning that each group of information is connected to another group through identified relationships. For example, if you have a group of information about the buildings in a city and another group of people in the same city, you might have a relationship identifying which people are in which building.

The information in each group must be defined with very precise structures if it is to be used effectively. People have a name and a birth date, but should the names be split into first and last names? They might have an address indicating where they live — is it important that the address be split into different components — house number, street name, city, postal code, and so on? For each piece of data, the database needs to know how it should be stored and what kind of information will be kept in it. If it's a piece of text, how long will it be? If it's a number, should it be storing decimal places? If so, how many? After you answer these questions, you might end up with a grouping of information like that in the following table:

Information	Name	Type	Length
Person First Name	FirstName	Text	35 characters
Person Last Name	LastName	Text	35 characters
Person Date of Birth	DateOfBirth	Date	Not relevant
Person Address	Address	Text	200 characters
Extra Notes	Notes	Text	As big as it can be

SQL Server Express uses tables to store a group of information. At its simplest, a *table* has a name and a collection of pieces of information. These chunks of data are called *columns* (or *fields*), and the preceding table would define the basic structure for a table of five fields.

Tables and their corresponding fields are what structure the information kept within the database. When actual data is stored in a table, each discrete collection of information is kept in a separate *row*, the term SQL Server Express uses for each of the information records. In the preceding example, you have a table with five fields relating to a person. If the database were to store information about a person named Trevor Greenstein, then his information would be kept in an individual row. If another person named Hayley Thomas were also to be stored in the table, her information would be saved in a separate row.

The information can be easily represented in tabular form (which comes in handy in Visual Basic Express because the editors for viewing the data within a database table use the same kind of format), so the sample data can be viewed like so.

Table Name: Person				
FirstName	**LastName**	**DateOfBirth**	**Address**	**Notes**
Trevor	Greenstein	09/23/1955	unknown	Likes plants
Hayley	Thomas	06/12/1973	123 Rainbow Parade	Ex-girlfriend

While you could access the entire table of information and look through the collection for the person's details you need, SQL Server Express enables you to define a way of accessing information directly — using a *key*. A key is exactly what it sounds like — a component of the table definition that helps find a particular row within the table's data.

Each table you create can have a *primary key* that is unique for that table. In the Person table example, the primary key might be a combination of first name and last name. However, there is a chance you could have multiple people with the same name, so SQL Server Express enables you to define special *Identifier* fields that are used specifically to create a unique index for each row. The preceding table could be modified to include such an identifier.

Table Name: Person					
ID	**FirstName**	**LastName**	**DateOfBirth**	**Address**	**Notes**
1	Trevor	Greenstein	09/23/1955	unknown	Likes plants
2	Hayley	Thomas	06/12/1973	123 Rainbow Parade	Ex-girlfriend

You can also use a special database language known as *SQL* (yes, this is why the product is called SQL Server Express), which stands for Structured Query Language. Using SQL, you can find individual rows of information by specifying the criteria you need to follow. SQL queries can be used to filter the information stored in the table and return only the rows that match the criteria; for example, a query could be written to find all rows that have a value in the Address column of unknown like so:

```
Select * From Person Where Address = 'unknown';
```

You'll see more about SQL queries later in this chapter and throughout the rest of this book, but for now, a quick definition: A SQL query is a type of search mechanism that can be performed against a database. It can be as simple as "get all the records" or quite complex, including merging several tables of information together and filtering out certain excluded search criteria.

Once you have the definition ready to go, you need to create the database and tables to store it. SQL Server Express has Manager applications that enable you to do this, but Visual Basic Express has its own components within its development environment so why go elsewhere?

The Project ➪ Add New Item command is used to add a new, empty database to the current project. The Add New Item dialog box has several templates from which to choose, including SQL Database. The name field is used to specify the actual filename of the database (see Figure 3-1).
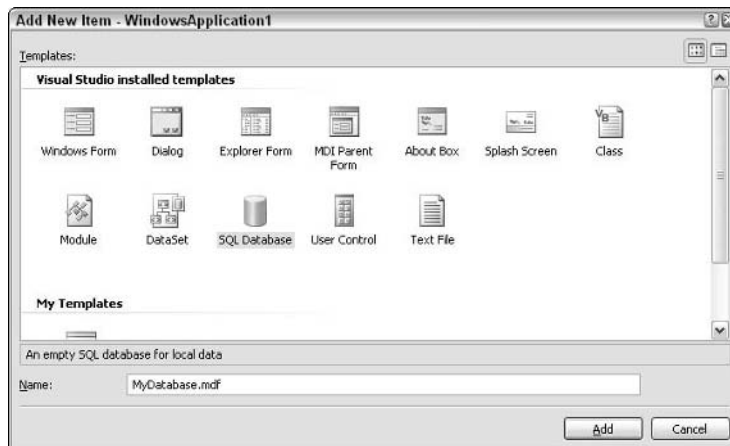
Figure 3-1

Once you have the database file, it can then be used in other applications by pointing them to the disk file that contains the database.

When you click OK, Visual Basic Express automatically creates an empty SQL Server Database and displays a Data Source Configuration Wizard to select individual components within the database. Because this is useful only when the database has definitions within it, you can safely cancel the wizard at this point (the wizard will be used in later chapters when you connect an existing database to an application).

The empty database file is then added to the Database Explorer. The Database Explorer normally shares space in the IDE with the Toolbox, but if you cannot find it, you can show it by selecting View ⇨ Database Explorer (see Figure 3-2).

When you first add the database, it may appear in the Database Explorer window with a small red X to indicate it is currently disconnected. Just click the icon and Visual Basic Express will go through the process of connecting it to SQL Server Express and displaying the contents of the database.
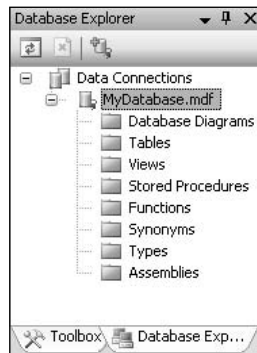
Figure 3-2

To create a new table within the database, right-click the Tables folder and select the Add New Table menu command. The main part of the IDE will show a specialized editing form for database tables with three columns of information.

Each field is represented by a row in this editor, with the columns representing the main pieces of information that are required — the name of the field, the type of data that is to be stored in it, and whether the database should allow the field to store nothing, or null, for any given row. In addition to this basic information, a Properties window is displayed below the field list with more advanced settings that can be applied to each individual field.

For identifier fields, the Properties window includes an Identity Specification group of properties. The field that should uniquely identify the rows within the table should have the `Is Identity` field set to `Yes`. This will tell SQL Server Express to keep track of the data stored in this field, making sure each row stores a unique value. In fact, SQL Server Express will actually automatically increment the `Identity` field so you don't even need to worry about making sure they're unique.

Creating a primary key is a matter of selecting the fields that make up the key, right-clicking the header button in the row, and selecting Set Primary Key from the context menu. If you're using an identifier

field, usually you'll set the primary key to that data field. Figure 3-3 illustrates how the preceding sample data definition could be represented in the table editor. Note that the Column Properties area is displaying the information about the selected field — ID — and includes the Is Identity = Yes setting.
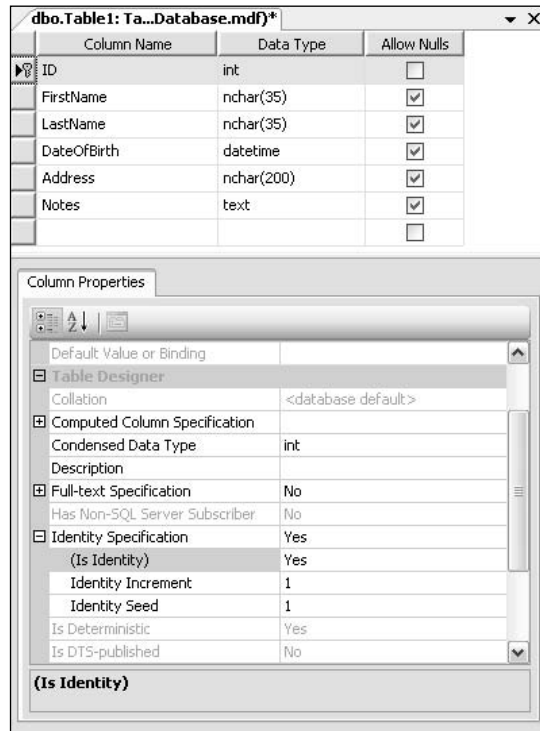
Figure 3-3

Once you're done adding fields and setting the corresponding properties, save the table to the database by using the File ⇨ Save command. At this point, Visual Basic Express will prompt you for the name of the table and add it to the list in the Database Explorer.

When tables are available in the Database Explorer, you can take a look at the information stored within the table by right-clicking the list and selecting Show Table Data from the context menu. Visual Basic Express displays the rows within the table in a fashion very similar to the table you saw earlier in this chapter (see Figure 3-4).

Figure 3-4

The information returned from the database can be edited directly in this window, including creating new rows of data and deleting existing ones. To create a new row, select the first editable field in the bottom row of the table that has all `NULL` values. In this case, because the `ID` field is an identifier that is automatically maintained by SQL Server Express, the first field would be `FirstName`. Type the information required and tab to the next field. Repeat this process and then navigate off the row to save the information.

Delete a row, or rows, by selecting the rows to be removed (by clicking their row header buttons), right-clicking, and selecting Delete.

Additional tables can be created by repeating this process. As mentioned earlier, SQL Server Express is a relational database, which means you can tell the database how tables relate to each other. If there were another table called Pet that stored the information about various pets owned by people, you might want to show that the two tables are linked. A person might own no, one, or many pets, so you need a way to connect this information.

To achieve this connection, you can define an additional field in the Pet table that identifies the Person row that "owns" each Pet row. As you look through the definition for the Person table, the obvious choice is the ID field because you know this is unique. As a result, the Pet table definition might look like this:

ID	PersonID	Name	Type	Breed
1	1	Amy	Dog	Silky Terrier
2	1	Muffin	Dog	Maltese Terrier
3	2	Tiddles	Cat	Siamese

The PersonID column identifies the person to which the individual Pet rows belong—Trevor owns two dogs, named Amy and Muffin, while Hayley owns a Siamese cat named Tiddles. You can write SQL queries to retrieve the information in the Pet table for a specific Person record, but a potential issue exists—there is no database-defined relationship.

Even though you can look at the database tables and see the connection between the two, SQL Server Express cannot do the same. This means you could potentially add rows of information in the Pet table with a `PersonID` value that doesn't match any rows in the Person table. To solve this, you need to explicitly define a *relationship* between the two tables.

A relationship is defined by specifying a field as a different kind of key—a *foreign key*. A foreign key indicates that this field is uniquely identified within a different table. To create a foreign key relationship, click the Relationships button on the Table Designer toolbar. A list of relationships is displayed in the Foreign Key Relationships dialog box (see Figure 3-5). Click the Add button to add a new relationship and then click the ellipsis button on the Tables and Column Specification property to bring up the Tables and Columns dialog box.
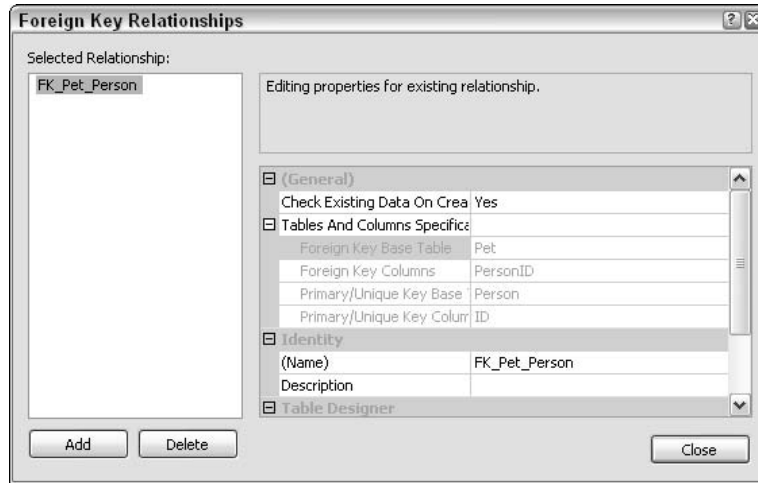
Figure 3-5

This dialog (see Figure 3-6) enables you to name the relationship and select the tables that should be linked. One table is designated as the Primary key table, which means the columns you are selecting define the key in that table that uniquely identifies the row. The other table is the Foreign key table, which specifies the table that will be linked to the primary table.

Each field specified in the Primary key table must map to a corresponding field in the Foreign key table. In Figure 3-6, the Person table is defined as the Primary table and the ID field has been selected as the identifier column. The Foreign key table is the Pet table and the PersonID field is selected to map to the ID field in the Person table.
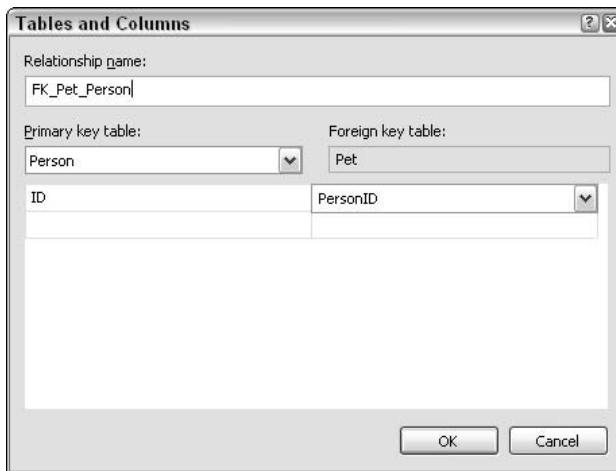
Figure 3-6

Once this relationship is saved to the database definition, whenever a row is added to the Pet table, the PersonID value is checked against the rows of the Person table. If no row in the Person table with a matching ID value is found, an error is generated and the information is not saved to the database.

Relationship definitions can also include what action to take, if any, when certain events arise. For example, if a program deleted a row from the Person table, you can automatically delete any corresponding rows in the Pet table. In the Foreign Key Relationships dialog box, select the relationship you want to control and change the Update Rule and Delete Rule properties to tell SQL Server Express what to do to rows that are connected to the Primary key table row. `Cascade` will automatically update or delete the connected rows, while `Set Null` and `Set Default` will not delete the rows, but set them to the respective values of `Null` or the default value for each type.

Throughout this book, you'll be creating an application that keeps track of your friends and family members. It will store and maintain their names, addresses, phone numbers, birthdays, and other information to help you remember their likes and dislikes. The following Try It Out walks you through the creation of the database structure for the information you'll need for this application.

Try It Out   Creating the Database

1. Start Visual Basic Express and create a new project by selecting File ➪ New Project. Select the Windows Application template from the New Project dialog. Name it **Personal Organizer Database** and click OK. Having a separate project for the database design is nice because it enables you to work on the database structure without having the user interface and code in the way.

2. Add an empty database by selecting Project ➪ Add New Item. Choose the SQL Database template and name the file `PO-Data.mdf`. Click Add to add the database to the project. Because the database is empty and this project is going to be used exclusively for editing the database structure, click the Cancel button in the Data Source Configuration Wizard.

3. The core of the Personal Organizer application is the information about each family member and friend. For this application, you'll need to keep track of their names, addresses, e-mail addresses, birthdays, what things they like, and their phone numbers. (In addition, you will include fields related to finding gifts for the person, which will be used in Chapter 9.) Breaking this down into workable chunks, you get the following table.

Information	Column Name	Type	Length
First Name	NameFirst	Text	35
Last Name	NameLast	Text	35
Home Phone	PhoneHome	Text	20
Cell Phone	PhoneCell	Text	20
Address	Address	Text	255

Table continued on following page

Information	Column Name	Type	Length
Email Address	EmailAddress	Text	100
Birthday	DateOfBirth	Date	
Favorites	Favorites	Text	255
Types of Gifts	GiftCategories	Integer	
Additional Notes	Notes	Text	As big as it can be

You should also include a unique identifier field at the beginning of the table.

4. Open the Database Explorer by selecting View ⇨ Database Explorer. By default, it will share space with the Toolbox on the left-hand side of the IDE, and it can be pinned open so it isn't automatically hidden away. Expand the PO-Data.mdf entry in the list to display the different types of elements that can be kept in the database.

5. Right-click the Tables folder and select Add New Table. In the table editor, select the first empty field in the Column Name column and enter the information in the following table.

Column Name	Data Type	Allow Nulls
ID	int	Unchecked

Scroll through the Column Properties window until you find the Identity Specification group and change the Is Identity property to Yes so that SQL Server knows to use this field as the unique identifier that is automatically incremented for new rows in the table.

Right-click the ID column and select Set Primary Key. This tells Visual Basic Express and SQL Server Express that this is the field to use by default when searching the table.

6. Go to the next row in the table editor and repeat the process for each of the fields defined in the table in step 3. Text fields can use the nchar() type, with the number of characters allowed being specified within the parentheses — for example, NameFirst would have a data type of nchar(35).

The DateOfBirth field should use a data type of datetime. The normal text field type — nchar — can store only around 4K of information. This might not be enough for the Notes column, so use the text data type.

The text data type requires more processing by SQL Server and so is not used unless necessary.

7. Save the table and call it Person. The definition and the Person table entry in the Database Explorer will look like Figure 3-7.
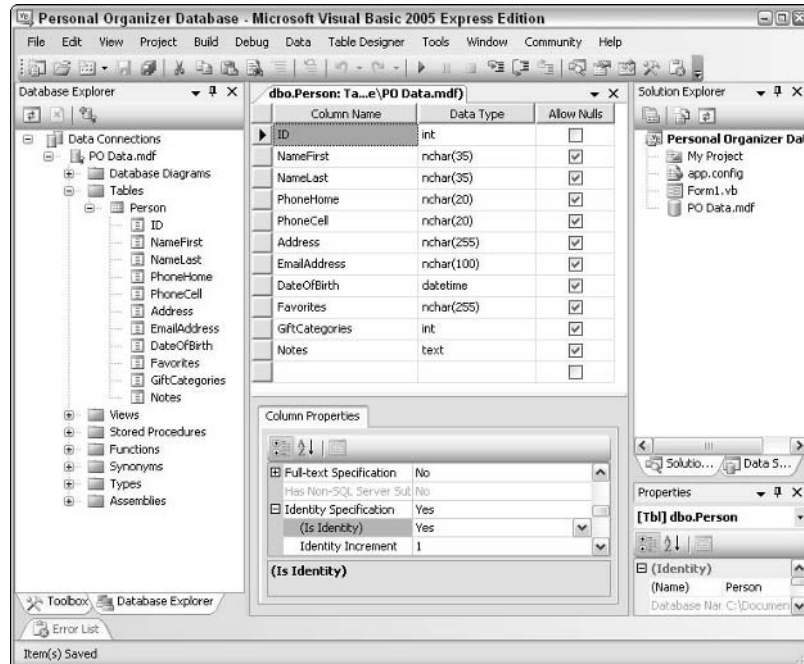
Figure 3-7

8. In later chapters, you'll restrict the information presented in the Personal Organizer application to the particular user who is accessing the data. To do this, you'll need another table in the database that stores the user's details. The user information will include the user's name, a system name, a password, a date field to indicate the last time they were logged on and another to indicate when the account was created, and a log entry to keep track of the number of failed attempts to log in as the particular user, as shown in the following table.

Information	Column Name	Type	Length
ID	ID	Integer	
System Name	Name	Text	255
Name	DisplayName	Text	20
Password	Password	Text	20
Created Date	DateCreated	Date	
Last Logged In	DateLastLogin	Date	
Login Failures	FailedLoginAttempts	Integer	

9. Add another table to the database by right-clicking the Tables folder in Database Explorer and choosing Add New Table. Add the preceding columns, remembering to include an identifier column as well so they can be uniquely identified, and set it to be the primary key. While the System Name should actually be unique, the database will perform better if there is a numeric identifier.

Remember to use `nchar()` *as the data type for Text fields, and* `datetime` *as the data type for Date fields.*

Ensure that the `Name` and `DisplayName` columns must have data by unchecking the Allow Nulls property. Save the table to the database and name it `POUser`.

10. Now you need to connect the two tables, so return to the Person table by right-clicking it in the Database Explorer and choosing the Open Table Definition command. You'll add an additional column that stores the `POUser` ID so the information can be filtered later in the application. Right-click the `NameFirst` row and select Insert Column from the context menu.

11. Name the column `POUserID`, make it an `int`, and uncheck Allow Nulls. Save the table definition. The tables are now set up, but no explicit relationship is specified. You'll do that next.

12. Click the Relationships button on the toolbar, or select the Table Designer ➪ Relationships menu command. When the Foreign Key Relationships dialog is displayed, click the Add button to create a new relationship.

13. Click the ellipsis button on the Tables and Columns Specification property to bring up the Tables and Columns dialog. Select `POUser` as the Primary key table and notice that the Foreign key table is already set to `Person`, as that is the table you were editing when you clicked the Relationships button.

In the columns area, choose `ID` from the `POUser` column and `POUserID` from the `Person` column and click OK to set the foreign key. Click Close to return to the table editing view. Click the Save button again to save the relationship to the database. Because this affects multiple tables, Visual Basic Express will display a confirmation dialog, as shown in Figure 3-8. Click Yes to force it to save the relationship.
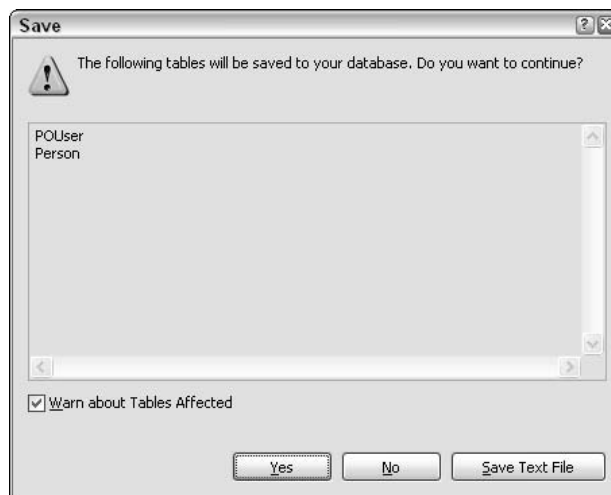
Figure 3-8

14. If you try to add a record to the Person table, it will enforce the relationship, not allowing any rows to be added without a corresponding entry in the POUser table with a matching ID to the POUserID. Go ahead and try to add a row to the Person table first, by right-clicking the Person table in the Database Explorer and selecting Show Table Data.

Enter information in all the columns and navigate off the row. Visual Basic Express will display an error dialog informing you that it was unable to commit the information to the database because it conflicted with the foreign key definition (see Figure 3-9). Press Escape to cancel the changes.
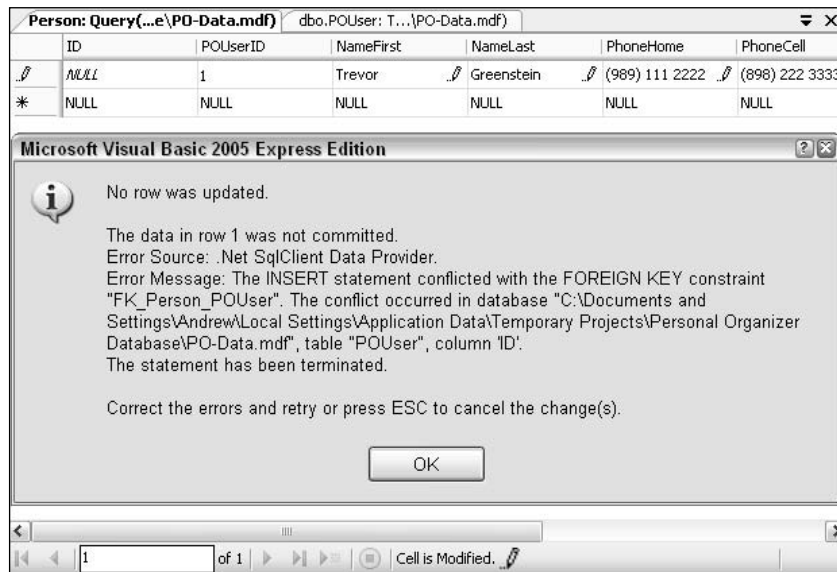
Figure 3-9

15. Open the POUser table and enter a row there to identify yourself. For now, anything will do, but later in the book, the System Name will be used to compare to the currently logged on user in Windows.

When you navigate off the row, SQL Server Express will automatically assign the next available number to the ID column. Take note of this number, return to the Person table, and re-enter all the information for a new Person row. This time, enter the number you noted in the POUserID column. When you save the row, SQL Server Express will accept the change to the database because the POUserID value matched an existing ID value in the POUser table.

16. Save the project so you can return to it later. The database file will be included in the project location, so note where you save the project for future chapter exercises and Try It Out examples.

Connecting Database to a Project

When a database is added to a project, it is not automatically connected to the rest of the project's components. If you take a look at the Data Sources window for a project to which you added a database file, you'll see that it is empty except for two things: a message that says the project does not currently have any data sources associated with it and a link to add a new data source.

The Data Sources window shares space with the Solution Explorer and is accessible through the Data ⇨ Show Data Sources menu command.

To add a database to the project, click the Add New Data Source link to start the Data Source Configuration Wizard. This is the same wizard that you canceled out of when you added the empty database, but this time it begins with the Data Source Type page. Select the Database option and click Next.

If the database is already in the project, it will be displayed in the existing data connection list. If you want to use this database or any other existing data connection, select it from the list and click Next. Alternatively, if the project doesn't have any data connections, click the New Connection button to add a connection to the project.

Visual Basic Express defaults to using SQL Server connections, so all you should need to do is specify the database filename. Click the Browse button to navigate to the location of the database file and click Open to select it. Again, by default, Visual Basic Express assumes you will use standard Windows authentication, but if you've changed your SQL Server Express setup to require SQL Server authentication, you'll need to specify a user name and password here.

Click the Test Connection button to ensure that the database is accessible and then click the OK button to return to the wizard. The next page of the wizard enables you to optionally save the connection string to your application settings. This is handy, as you don't need to remember the often hard to understand properties required to connect to the database; and once set, you won't need to worry about it again.

The next page in the wizard will be one you're familiar with from the previous times the wizard has been displayed. This time, however, the Tables node will have children entries for each table defined (and the Views, Stored Procedures, and Functions nodes will, too, if the database has those kinds of objects). Select the tables you want to include in the data connection and click Finish to finalize the wizard and add the data source to the project.

The Data Sources window will be populated with the `Dataset` object, including the tables and the individual fields within each table in a tree view. A sample of this can be seen in Figure 3-10. Note that the `POUser` table in this example includes a child reference to the `Person` table because of a defined relationship between the two.

Once the information is available in the Data Sources window, you can use it to bind user interface components to database elements, and write code to access the database through the Data Source objects. One very simple way of presenting the information to the user is by dragging the table directly onto a form.

Visual Basic Express will automatically add a tabular control known as a `DataGridView` to the form, along with a navigation bar with New, Delete, Save, and movement buttons. It will also define the required data objects for the form (shown in the tray area below the main user interface design).

Chapter 7 goes into a lot more detail about the various controls that you can use to automatically connect to the data sources you have defined in the project. For now, the following Try It Out uses the Personal Organizer Database project to illustrate how you can quickly view the data in the tables.
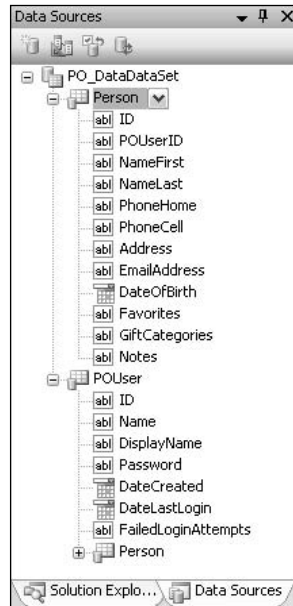
Figure 3-10

Try It Out   Connecting a Database and Project

1. Return to Visual Basic Express and the Personal Organizer Database project. If you closed the project at the end of the last Try It Out, you can re-open it by selecting it from the File ⇨ Recent Projects submenu.

2. Show the Data Sources window by selecting its tab next to the Solution Explorer tab. If it is not visible, select the Data ⇨ Show Data Sources menu command. Click the Add New Data Source link to start the Data Source Configuration Wizard.

3. Select Database and click Next to show the data connection page. By default, the wizard will detect that a database is defined in the project and populate the existing data connection list with the name of the database file. Click Next to go to the Save Connection String page.

Because you don't want to worry about the connection string later, leave the checkbox enabled so that the connection details are saved to the application configuration file and click Next.

Be aware that when you add the database locally, Visual Basic Express creates a fresh copy of the database each time the program runs. This means any changes you make to the information in the database while you're running the application won't be there the next time you run it. If you want to keep the changes between executions, you need to save the database file externally to the project.

4. In the Tables list, select both Person and POUser and click Finish. After a few moments, the Data Sources window will be populated (similar to the one shown Figure 3-10). Initially, the tables might not be expanded, so click the expand buttons to show the individual fields.

5. Go to the Form1.vb Design view by selecting its tab along the top of the main editing area. Drag the POUser table from the Data Sources window and drop it onto the form. Visual Basic Express will automatically add all the required objects and user interface controls.

6. Run the application by pressing F5 or selecting Debug ⇨ Start Debugging. When the application starts, the form will be shown with the DataGridView control populated with the information you added to the database in the previous Try It Out (see Figure 3-11).
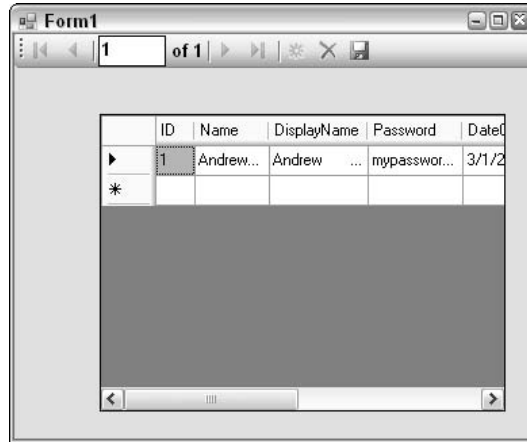
Figure 3-11

Stop the application by clicking the Close button on the form, and save the project so you can preview the data later.

Alternatives to SQL Server Express

While SQL Server Express is the normal way of doing database storage in a Visual Basic Express application, you have several alternatives available within Visual Basic Express. *OLE DB* (it stands for *Object Linking and Embedding Database* but is almost always simply referred to as the acronym) is Microsoft's way of providing a generic database standard. Programs created in development environments such as Visual Basic Express can use OLE DB to access different database types without needing to know specific methods to do so.

As long as the manufacturer of the database distributes an OLE DB interface to their database, you can access it using common commands, classes, and methods. Visual Basic Express defaults to SQL Server as the data provider, but if you want to use a non-SQL Server database, you'll need to switch to OLE DB.

Microsoft Access databases can be used via OLE DB, and Visual Basic Express includes the data provider for Access as one of the starting options. The rest of the settings are the same, and once you add the database to the project, you can interact with it in a similar fashion to a SQL Server database.

This means you can navigate through the database and table structure in the Data Sources window, and you can view the tables and preview the data in the Database Explorer (although you cannot edit the table definitions of an OLE DB-based database within Visual Basic Express). Dragging the table or fields from the Data Source onto a form will automatically create the user elements needed to access the database, just as a SQL Server database will.

The differences become apparent when writing code. Whereas SQL Server databases are accessed through the `System.Data.SqlClient` set of classes, OLE DB database files are processed using the `System.Data.OleDB` classes. This is because the various methods and properties differ for each database type, and Microsoft has made a concerted effort to fine-tune the performance of SQL Server databases.

Summary

Storing and accessing information in a database is an essential part of programming. Visual Basic Express, with the aid of SQL Server Express, makes the process of creating a database straightforward, and then continues the ease of development by enabling you to add database information to a Visual Basic application through wizards and simple drag-and-drop functionality. With these tools at your fingertips, you can ensure that your application is synchronized with the data that drives it.

In this chapter, you learned to do the following:

❑ Create and maintain database definitions for an SQL Server Express database

❑ Look at the different types of databases Visual Basic Express can use

❑ Add a data source to a Windows Application project and add a simple data view to a form

In the next chapter, you'll look at the user interface of an application and how Visual Basic Express helps you create solid designs that your users will appreciate.

Exercise

1. Create a database that uses the Person and Pet tables defined at the beginning of this chapter. Make sure they are linked through a foreign key relationship so that each Pet record must be owned by a Person record.

4

What the User Sees

Creating a program can be divided into three discrete parts. First is the data, which is the whole reason for the application. This is normally stored in a database and has tables and queries defined so that the program code can access the information in a logical manner. You saw how databases work in Visual Basic Express in Chapter 3. The second component is the program code that operates on the data and controls what the user can (and cannot) do. This programming logic will be the subject of the next chapter. This chapter deals with the third section of creating an application—designing the user interface.

Getting the user interface right is vitally important because if users do not like what they see, or can't figure out how to use your program, then it doesn't matter how good the code is underneath it, or how meaningful the data.

In this chapter, you learn about the following:

- ❏   The importance of good user interface design
- ❏   The common controls used to create user interfaces
- ❏   Building menus and toolbars

User Interface Basics

When creating an application, many programmers leave the user interface to the last minute, which results in an ill thought-out layout that hampers the end user's experience. If the design isn't intuitive, the people using your program will have difficulty accessing the functionality and may decide to use someone else's program instead, even if the feature list is not as impressive.

This has happened on numerous occasions in the software industry. Companies release a version of their software with all new bells and whistles but overcomplicate the user interface design. As a result, their competitors release their own software with a more elegant interface, which helps their market share.

In some cases, the reverse is true. For example, an application released in Australia was designed for simplicity with no obvious buttons on the main window. Instead, everything was done using keystrokes; and for those who tried it, it worked really well. However, the main competition was a piece of software that more closely resembled other commonly used programs such as Microsoft Excel. Even though the functionality of this competitive product wasn't as detailed, customers preferred it because it was something they were familiar with. It was frustrating for the company that released the first product because they knew their application was better. Only when they implemented an optional add-on to the product that enabled users to customize the interface with buttons and commands did their sales pick up, and now the add-on is installed by default.

Remember: If the users of your program can't figure out how to use it, they won't be your users for long. Documentation in the form of manuals and help files can be useful, but they should assume the role of supporting information, rather than being a required tool to use your application.

The best thing about good user interface design is that it doesn't take a lot of thought to actually make it happen.

User Interface Fundamentals

While whole books could be written (and have been) on the importance of a good user interface and the methodologies you can use to achieve the best results for your application, if you follow these simple guidelines (they're guidelines, not rules because there are always exceptions) your application will look clean and be user friendly:

❏   **Let the user decide.** First and foremost, think carefully before changing colors and fonts. By default, controls in Visual Basic Express incorporate the standard system colors of the operating system along with the fonts the user has chosen to use. If you leave these settings as is, users can customize their experience of your application by changing their system setup. Forcing certain color schemes or fonts on users of your application may be detrimental to the usability of your program.

❏   **Be consistent.** Make the design of your objects uniform. If you set the height of the buttons on your main form to 300 pixels, then make sure you set the height of the buttons on any other forms you might use to the same. If you use a `TextBox` control to display information instead of a label, then do the same for other informational panels you may add.

Failing to use consistent styles and settings when adding and customizing your components reduces the cleanliness of the interface, making it harder for the user's eye to find what it's looking for. Obviously, you'll come across exceptions to this guideline, but ensure that the exceptions are few and not the majority.

❏   **Differentiate your objects.** When you need to display one particular element differently than the others, make sure you use enough contrast to distinguish it. If you do not distinguish it well enough, not only can it be hard to determine whether it is indeed different, but because the difference is subtle, the user's eyes can stumble.

Consider the example shown in Figure 4-1; the Bigger button has a slightly larger font — can you tell? The Italic button is obviously different from the Normal button but because of the screen it can be hard to read. The Bold button is obviously different from the Normal style and so is the best option out of these three for contrast.

Figure 4-1

When deciding on contrasting styles, remember that a significant number of people are color-blind to some degree (with red being the most common). If you choose to use color — which is certainly acceptable — remember to also alter the style in some other way as well.

When considering the use of color in your application, there is an excellent option that should be explored. Rather than select specific color values, you can choose system-defined colors such as `ButtonFace` *and* `ActiveBorder` *that are controlled by the user through the Windows Control Panel. This enables users to select the colors most effective for their own situation and gives your application the flexibility it requires to cater to different needs.*

❑ **Align elements clearly.** If you don't position the elements on your form with respect to each other, your interface is doomed. Even a 1-pixel difference in the left-hand side of two controls can show up in the final design as being sloppy and unprofessional. Visual Basic Express does help you with this by providing guidelines and snap-to markers, but you should always take care to properly align your controls.

❑ **Position elements logically.** Group those elements that belong together so that users can navigate easily. Note how most applications have menus in which the commands are divided by function, and toolbars that group like actions together. This helps users find the function they're looking for. The same applies to your own design, and this rule extends to the main section of your form. If you were to create a form with information about a person, you would keep the first name and last name fields together, rather than split them up, for example.

These guidelines may seem self-evident to you. If so, great! As you create your user interface, keep them at the forefront of your decision process and you'll find the layout of your windows is easy.

Adding and Customizing Controls

Visual Basic 2005 Express makes it easy for you to create forms that follow good user interface design practices. Guidelines help you to position and size each control, and components inherit certain properties, such as the font, from their parent component. This enables you to change a property in one place and have the new setting reflected by all of the child components, ensuring that you maintain a clean consistent design.

Before you take a look at a selection of the controls you can use to create your program's user interface, it will be beneficial to review how to add a control to a form and then customize its properties. You'll find that almost everything about a component you add to your application's forms is accessible in Design view, which lessens the need to write code. In the following Try It Out, you'll see how the Visual Basic Express IDE helps you when adding controls to the forms in your projects.

Try It Out   Adding a Control to a Form

1. Start Visual Basic 2005 Express and create a new Windows Application project. Name it
    `MyFirstProgram`.

2. From the Toolbox, click and drag the `Button` control to the form. Notice the help indicators as
    you drag the button over the form's design surface? As this is the first control being added to
    the form, the lines will guide you to the preferred distance from the edges of the form. Position
    the button so that it is the ideal distance from the top of the form and in the middle horizontally.

3. Now add another `Button` control to the form in the same way. This time, as you drag the new
    object over the form, not only will the indicators show the distances from the edges of the form,
    but also the ideal distance from the other controls on the form. In addition, new lines (this time
    colored blue), will show when the control is aligned horizontally or vertically with the existing
    components. Both the ideal positioning and alignment guidelines can be seen in Figure 4-2.

    Another point to note is that as you drag the control around the form, it will snap to these lines
    when you get close enough. This takes away the "guess factor" in positioning the controls.
    Position the second button directly below the first one.
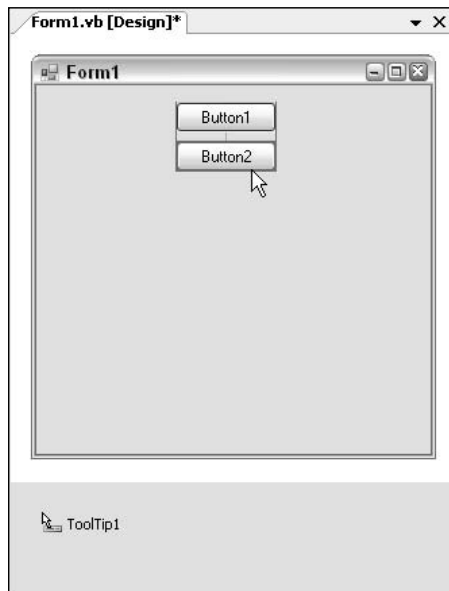
Figure 4-2

4. Resize the second button by clicking over the bottom boundary and dragging it down so that it
    is roughly square. Some controls are not added to the form's viewable area when you drop
    them on it. These controls normally work in the background and will appear in a small tray area
    below the form's design surface. The form shown in Figure 4-2 has had a `ToolTip` object added
    to it.

5. Every control has a number of properties that define its appearance and control how users can
    interact with it. Some of these properties, such as `Width` and `Height`, can be controlled by the
    mouse on the design surface to set the size of the control. However, these properties, along with

most of the others, can be accessed in the Properties window (some controls have properties that cannot be changed at design time — these need to be set in code). Select the second button you added to the form and scroll through the properties to see what's available.

6. Change the Text property to `Say Hello`. For a button object, this changes the caption that the user sees on the button. Change the `ForeColor` property to red and note how the text changes color to match.

Controls can easily be added to your form by clicking and dragging them to the desired position. Alternatively, you can double-click the control name in the Toolbox, and a new object will automatically be added to the form. Once on the form, you can set most of the properties through the Properties window.

The Controls

The following sections describe the various controls that Visual Basic Express provides you with to create your program. Every one of these accelerates the development process because you need to write less code and can do more with simple click-and-drag actions. While this list isn't exhaustive, it will introduce you to the major components that make up a program. As you progress through the rest of this book, you'll use these controls to build programs as you learn about useful properties and events.

Basic Controls

Ultimately, creating your application comes down to presenting users with information and enabling them to tell it to perform some functionality. The basic, or fundamental, controls enable you to do just that. With these half-dozen or so components, you can create just about any interface you can imagine.

Label

The `Label` control displays read-only information for the user. The text that is displayed can be changed only by the program and is usually used to convey the purpose of other controls. For example, you may position a label with text of `First Name` to the left of a `TextBox` to inform the user that the information in the `TextBox` is to be the name of a person.

Button

The most common method to enable users to tell the program to do something is using a button. `Button` controls can be used in many different ways, but the basic button contains a descriptive word or two and when clicked performs some function.

`Button` controls can have images as well as, or instead of, the text. A good example of buttons used in this way is the toolbars in many applications, such as Microsoft Word. You can customize the button so that it shows only the icon, only the text, or both. In Visual Basic Express, you have control over the positioning of the text and image and can even set a background image as well as the icon image.

The only event worth looking at for the `Button` control is the `Click` event. This is raised whenever the user clicks the button.

There is another control called the `LinkLabel` that combines the display style of the `Label` control with the clickable nature of the `Button` control.

TextBox

Most applications require that users be able to enter information at some point, and this is where the `TextBox` control steps in. `TextBox` objects are just that—boxes that contain text. The `TextBox` can either contain a single line of information such as a user name or password field or be used to display many lines at a time, such as you might need in a memo area.

`TextBox` controls can be customized in many ways—the setting to toggle between a single line and multiple lines is called `Multiline`. Other useful properties include `Alignment`, to control where the text should be positioned, and `CharacterCasing`, to automatically convert the text to uppercase or lowercase.

The contents of the `TextBox` can be locked in two ways. The first way is to use the `Enabled` property, which is common to all user interface elements. If `Enabled` is set to `False`, then the control cannot be interacted with. A disabled button cannot be clicked, and a disabled text field cannot be changed or selected. The alternative to this for `TextBox` controls is to use the `ReadOnly` property. While this still prevents users from changing the contents of the field, they can select the text and scroll through the contents.

MaskedTextBox

The `MaskedTextBox` is a `TextBox` control with additional functionality built in. You could actually create the same functionality in code, but if you need to create a text field that allows information only in a set format, then let Visual Basic Express do the work for you and use a `MaskedTextBox` control. The additional property used to control how the user can enter the information is the `Mask` property. You could set this directly in the Properties window, or choose from a preset number of styles by clicking the smart tag arrow on the control and selecting the Set Mask link. This will display a dialog box containing predefined masks, along with the capability to create your own, as shown in Figure 4-3.
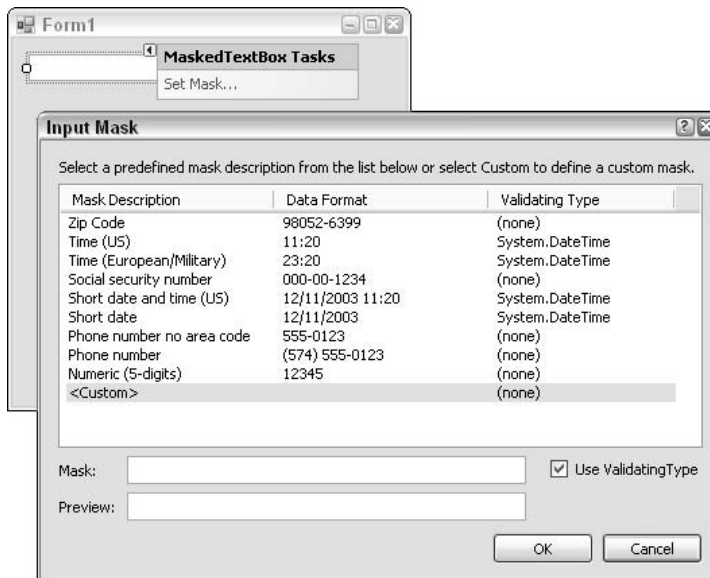

Figure 4-3

CheckBox

The `CheckBox` control is normally used to display options that can be turned on and off. You can change the label by setting the `Text` property and control whether it is checked or not using the `CheckState` property. The `CheckState` property has a third possible value — `Indeterminate`. This is normally used when the program cannot give a definitive yes or no answer. An example of this might be when the `CheckBox` has a number of other fields below it, some of which are checked and some of which are unchecked (the sample form in Figure 4-4 shows how this can work).

Figure 4-4

RadioButton

`RadioButton` controls are sometimes known as option buttons. They represent a set of information whereby only one option can be chosen. Each option is displayed as a separate radio button and users click one to select it. When one radio button is selected, all other radio buttons are deselected automatically.

Because `RadioButton` controls are automatically grouped in this fashion, it's common to keep them in a layout container control such as a `GroupBox` or `Panel` so they don't affect other options groups. To achieve the separate color and clothes selections shown in Figure 4-5, two `Panel` controls were first placed on the form and then the `RadioButton` controls were added to the `Panel`s.

Figure 4-5

ComboBox

When you have a large number of options from which to choose, `RadioButton` controls may get a little messy. Or, if you have limited space on the form, it may be impossible to position the `RadioButton` controls so they're usable. `ComboBox` controls can be used to avoid both problems. A `ComboBox` stores a list of information from which users can choose. To display the list, they click the drop-down button (in the form of a downward pointing arrow) and once they've selected an option, it is displayed in a `TextBox`.

ListBox

An alternative to the `ComboBox` for presenting a lot of information is the `ListBox`. The `ListBox` can be sized and positioned so that users can see many lines of data at once.

In the default selection mode of the `ListBox`, clicking on a line will automatically deselect any other line. However, using the `SelectionMode` property, you can control this property to enable users to select multiple entries in the list.

HScrollBar and VScrollBar

Scrollbar controls can be used to enable users to pan the visible surface of your application or to control the value of a numeric variable. Visual Basic Express has two types of scrollbar — the `HScrollBar` for horizontal scrolling and the `VScrollBar` for vertical scrolling. Each control has `Minimum` and `Maximum` properties to set the bounds of the scrollable range, while the `Value` property returns the current value of the scrollbar's position.

Layout Controls

While it is possible to place all of the basic controls on the form individually (with the exception of multiple groups of `RadioButton` objects), Visual Basic Express provides additional components that make designing and maintaining the user interface more efficient. Layout controls do just that — control the layout of the form by grouping and positioning sets of other controls. Most of these controls are known as *container controls* because they can contain other controls. You can access the collection of controls within a container via its `Controls` property.

> *One other very important layout capability of Visual Basic Express is docking and anchoring, which is discussed later in this chapter.*

GroupBox

Around for almost as long as the `Button` and `TextBox` controls, the `GroupBox` component is a container object that has a frame and caption. Its only purpose is to help lay out groups of controls.

Panel

The `Panel` control is the workhorse of user interface layouts. `Panel` controls are borderless by default and inherit the color of the object on which they are being placed. This means you can lay out your form with many panels controlling the location of the different elements, and rather than having to move each individual control, you can simply move the panel instead.

There are two customized versions of the `Panel` that behave in a different manner — the `FlowLayoutPanel` and the `TableLayoutPanel`:

❑ The `FlowLayoutPanel` works much like a web page does — as each control is added to the panel's surface, it is tacked on the end. The layout moves from left to right, top to bottom, as shown in Figure 4-6.

❑ The `TableLayoutPanel` splits the panel's area into columns and rows, with each cell able to contain a single component. If you need more objects in one of the cells, simply use another `Panel` object in the cell and place the multiple controls you require in it.

Figure 4-6 shows examples of all three panel types, with the background colors set to make it easier to distinguish. The area in the top-left corner is a `FlowLayoutPanel` where the buttons wrapped around when the layout ran out of room. The area in the bottom-left corner is a `TableLayoutPanel` with a button object in each cell, while the area on the right side of the figure is a standard `Panel` on which the buttons can be placed where you need them (notice the button that is partially cut off because it extends beyond the visible area of the panel itself).

Figure 4-6

SplitContainer

The `SplitContainer` control is the odd man out in the layout controls group. Rather than being able to contain other controls, the `SplitContainer` divides an existing container component into two, either horizontally or vertically (controlled by the `Orientation` property). By default, when you drop the `SplitContainer` object onto a container component, it will automatically create two `Panel` objects and a splitter bar; the `Panels` will be placed on either side of the splitter bar.

With `SplitContainer` controls it is possible to create complex, powerful user interfaces that enable users to change the view to suit their needs. This is because not only does the control split a container into two discrete parts that can be managed separately, it also handles situations in which the user wants to resize the areas on either side. A real-world example is the interface of Microsoft Outlook 2003. The main area is divided into three views — the folder list, the list of e-mail, and the preview pane containing the currently selected item. Between each of these views is a splitter bar that enables the user to resize the areas.

Menu and Status Controls

Another handy group of components are those that provide information and quick links to common commands. Menu bars and toolbars have been around for as long as the Microsoft Windows operating system, and give the user a set of commands to interact with, usually grouped into categories. Status bars reside at the bottom of many application forms and provide instant feedback to users about the state of the program.

MenuStrip

Modern applications often allow users to change the location of a menu, and even allow the menu to be undocked from the side of the form, leaving it floating over the window. The `MenuStrip` control is the newest way of creating menus, and it incorporates these features as well as other recent developments in menu styles.

You'll take a look at how to create menu bars and toolbars in detail in Chapter 6.

ToolStrip

Toolbars are now created using the `ToolStrip` control. Each `ToolStrip` can be positioned independently and can have a number of controls added to it of various types, including text fields, combo box controls, and the standard button controls that are common to almost every toolbar.

One particularly cool item type is the `SplitButton` that works as both a button and a drop-down list. This enables you to emulate things such as the color selection command in Microsoft Word where clicking the main part of the button sets the color to the current setting, while clicking on the drop-down arrow shows a table of colors to choose from.

Like `MenuStrips`, `ToolStrips` can be repositioned by the user if you enable it and can have separators and customized buttons to help the user understand the layout.

StatusStrip

The `StatusStrip` component provides you with the capability to easily add informational panels to your form. By default, the `StatusStrip` will dock itself to the bottom of the form, but you can move it to another edge or even let it float over the rest of the form's components.

The `StatusStrip` has two types of area that can be added to it — *panels* and *progress bars*. The latter can be used to show the user the status of a particularly long operation. The panel areas can contain images, text, or both, and each `StatusStrip` can have multiple panel and progress bars to meet your design requirements.

ContextMenuStrip

When you right-click on an object in a form, often a small menu of commands will appear. This is known as a *context menu,* and you can easily create these menus using the `ContextMenuStrip` control.

Creating a context menu for a control is done in two steps:

1. Add a `ContextMenuStrip` control to the form by dragging it from the Toolbox. Then customize the contents of the menu as you would a normal menu control.

2. Once the control is set up, select the object that should have the context menu and locate its `ContextMenu` property in the Properties Window. Click the drop-down arrow to see a list of available menus and select the `ContextMenuStrip` control you added to the form.

ToolTip

The tool tip provides additional information about an element on a form when the user hovers the mouse over it. Visual Basic Express provides tool tip functionality with the `ToolTip` control. When the `ToolTip` control is added to the form, it extends all the other controls on the form with an additional property.

This automated extension of a control's properties enables all controls to have one central location for the tool tip settings, which is a departure from previous programming environments in which you were required to create the tool tip style for each component.

HelpProvider

Another control that does most of its work by extending other controls is the `HelpProvider`. To connect different parts of your program to a set of documentation, you first create a `HelpProvider` control and set its properties. Then, for each control that you want to connect to the documentation, use the four additional properties that are dynamically inserted by the `HelpProvider` to specify the parameters for how to display help.

NotifyIcon

Before Visual Basic Express, programmers were forced to create complicated objects and call obscure Windows system routines to add an icon to the notification area in the bottom right-hand corner of the main Windows interface. Now you can do it with the simple addition of the `NotifyIcon` control on the form. Used in conjunction with the `ContextMenuStrip` control, you can provide your users with quick access to common commands even if your application is not visible.

Dialog Controls

At times, you will need to provide users with functionality that is standard across most Windows applications. Enabling users to select a color or font, or open or save a file to a location of their choosing, are common enough that custom built dialog controls have been created and are included in Visual Basic Express for your use.

Each dialog control has a set of properties you set either at design time or in code. For example, the `ColorDialog` lets you set a default color and toggle the full color selection mode on and off. The dialog controls are not shown by default but instead are shown by writing program code to display them. In Chapter 12, you'll use the `OpenFileDialog` and `SaveFileDialog` controls to find and process files.

Graphic Controls

Sometimes you need to use graphics to convey information to the user either in the form of an icon on another control or standalone. Visual Basic Express comes with several controls to make these processes easy. The two most common are the `ImageList` and `PictureBox` controls.

ImageList

The `ImageList` control is another one of those non-user interface elements that doesn't display on its own. Instead, it is used to store a small library of similar images that can then be used by other controls. The advantage of using `ImageList` controls is that you can keep the icons for a series of buttons and other controls in one place; and instead of assigning the image itself to the control, you simply point to the location of the image.

The `Images` property provides access to the collection of images stored in the `ImageList` control. The images can be in most common image formats and can have different dimensions. The `ImageList` then converts them internally to a standard dimension and can even optionally identify transparent areas in the image.

Controls that can use the `ImageList` will have an `ImageList` property that enables you to select from all `ImageList` controls on the form, and an `ImageIndex` property that identifies the location within the collection of images.

PictureBox

The `PictureBox` control is used to display an image. With the introduction of `Panel` controls and the capability to independently set the background image of most standard controls, `PictureBox` controls are not as widely used as they once were.

Images in `PictureBox` controls can be stretched or zoomed and the control can contain several other images to show in the event of errors and long loading times. These additional images are handy when the main image is to be loaded from an external location such as the web.

Other Controls

Many other controls are available for your use, including nonstandard ones that you can purchase or download from third-party vendors. The following sections describe just some of the other components that are packaged with Visual Basic Express, ready for your use.

Data Controls

Several controls help bind database tables to the user interface elements. Collectively they're known as the *data controls* and can be found in the main Toolbox or in a separate group called `Data` for easy access.

You'll learn how to use these controls in Chapter 7.

Print Controls

One other set of components in the main Toolbox is related to printing. You'll use several of these in Chapter 11, but here's a quick summary of what they do:

- ❑ `PrintDocument` — The main workhorse for printing, the `PrintDocument` control sends information to the selected printer according to the formatting and settings you provide.

- ❑ `PrintPreviewControl` — As you might have guessed, adding the capability of previewing your printed documents is as easy as putting one of these controls on your form and passing it the information. Zooming and pagination is handled automatically.

- ❑ `PrintDialog` — This is the standard print dialog, which enables users to choose the printer to which they want to send documents, along with other options if you have provided the information, such as number of copies.

- ❑ `PrintPreviewDialog` — Used in conjunction with the `PrintPreviewControl`, the dialog provides access to various settings regarding how to preview the information to be printed.

- ❑ `PageSetupDialog` — This enables users of your application to customize the way the information should be printed.

Miscellaneous Controls

Still many more controls come with Visual Basic Express, along with hundreds of others that are available for purchase over the web. Some of the more interesting controls include the following:

❑ `DateTimePicker` — This control enables users to choose a date from a drop-down calendar. You'll use one of these in the Try It Out at the end of this chapter.

❑ `WebBrowser` — A complete web browser all wrapped up and ready to go, the `WebBrowser` control gives you the capability to put the Internet inside your application.

❑ `SoundPlayer` — As its name suggests, this control gives you quick access to playing audio files as part of your application's processing.

Anchoring and Docking

As mentioned earlier in this chapter, using layout controls is not the only way of automating the layout of your user interface design. All Visual Basic Express controls have two additional properties used specifically for layout — `Anchor` and `Dock`. Using these properties will automate the resizing of controls as the user resizes the form.

Anchoring

The `Anchor` property tells Visual Basic Express where the control should be situated on the form. By default, `Anchor` is set to `Top Left`, which means the control will always remain the same distance from the top and left edges of the form. Anchoring can be set to any side of the form and in any combination.

For example, changing the `Anchor` property of a button control to `Top Right` means that it will always stay the same distance from the right-hand border of the form as well as the top. You should be aware of a couple of tricks, however, that make anchoring even more powerful. If you anchor a control to opposing sides of a form, such as left and right, or top and bottom, the control will be resized so that it stays the same distance from both sides. The two side-by-side images in Figure 4-7 show the same form with a button that has an Anchor property set to `Left Right` — note how the button always remains the same distance from both edges of the form.
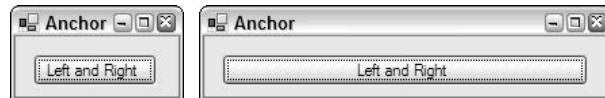

Figure 4-7

The second handy effect of the `Anchor` property is when you remove the anchor for two opposite sides. The result of this action is that the control will position itself proportionately on the form's surface. Suppose, for example, a button is placed on the form one-quarter of the way across the form and the `Anchor` property is set to `Top` only. As the form is resized, the control will always be located one-quarter of the width of the form, as you can see in Figure 4-8.

Figure 4-8

Docking

The `Dock` property aligns a control to one side of a form. When the Dock property is set, Visual Basic Express will automatically move the control to the specified side of the form and resize it so that it moves to the borders. For example, setting `Dock` to `Left` will move the control to the left-hand side of the form and then resize it so that the top edge of the control is aligned with the top edge of the form and the bottom edge is aligned to the bottom of the form. As the form is resized, the control will automatically resize itself so that the top and bottom edges are always aligned with the form's borders.

There is an additional `Dock` value — `Fill`. Rather than dock the control to one side of the form, the control is resized to fill the entire form. Usually when designing a user interface, you'll use a series of `Panel` objects to represent different areas of the form, and then dock them to different edges with one panel's `Dock` property set to `Fill` to take up the remainder of the space.

> *Remember that both of these properties are based on the container of the control, so if the control is anchored inside a `Panel` component, then it will move according to the `Panel`'s own size and position, rather than the form's.*

Like many features of Visual Basic Express, you could write code for this kind of situation; and in fact, previous versions of Visual Basic required many lines of code to resize and reposition controls when a form's dimensions were changed. These properties are another great example of how much Visual Basic Express helps in creating applications without the need for you to write code.

Building the User Interface for the Personal Organizer

Now that you're familiar with the kinds of controls that are available to you in Visual Basic Express, it's time to design the user interface for the personal organizer application. In the last chapter, you designed the basic structure of the database, so you know the kind of information you will need to add to the form to enable users to view and change the data.

In the following Try It Out, you will create the main form of your application, place and customize the basic elements that will be used to structure it, and create a custom control for information.

Try It Out   Creating the Main User Interface

1. Start Visual Basic Express and create a new Windows Application project. In this case, you're going to create everything from scratch, rather than use the wizards and starter kits you used in Chapter 2. Name the project Personal Organizer and click OK.

2. Once the form is displayed, change the following properties so that it is ready to contain the elements you will need to add:

❑ **Name** — `frmMainForm`

❑ **Text** — `Personal Organizer`

❑ **Size** — 460, 440

3. Rename the form to `MainForm.vb`. This will make it easier to determine what the file is when you're reviewing the project. To rename a file, find its entry in the Solution Explorer and right-click it. Select the Rename command and type the new name. The `vb` extension tells Visual Basic Express that this file is a Visual Basic code file, so make sure you retain it.

4. Add a `MenuStrip` to the form. Notice that the user interface element of the menu automatically docks to the top of the window. When the `MenuStrip` is selected in the Design view of the form, you'll notice a small arrow on the top right-hand side. This indicator is a *smart tag*, informing you that there are additional actions you can perform.

Click this arrow and change the Name to `mnuMainMenu`. While the Actions dialog window is still open, select the last command, Insert Standard Items. This will add default commands to the menu, such as File and Help menus (see Figure 4-9). You'll add additional menu items in later chapters as you need them.
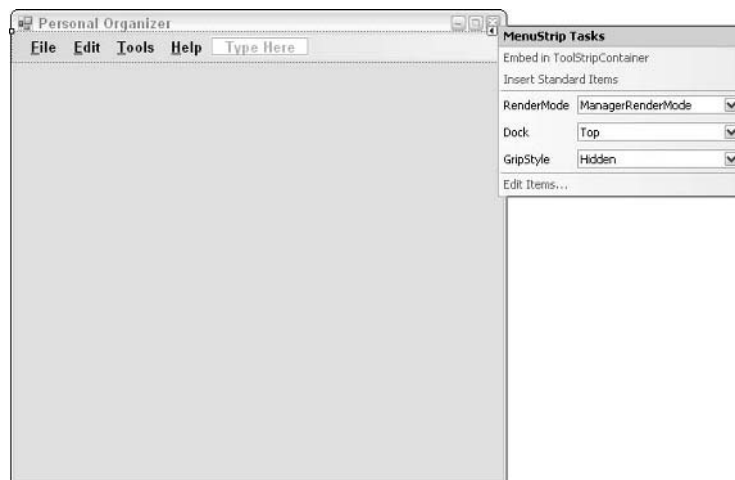
Figure 4-9

5. Add a `ToolStrip` control to the form. Again, click the smart tag indicator to bring up the Actions pane. Rename the control to `tbarMainTools` and run the Insert Standard Items command to add some default buttons to the strip. This will add commands such as New and Print, which you'll use in subsequent chapters.

6. Add a `StatusStrip` control to the form by double-clicking the entry in the Toolbox. Notice that this time it automatically docks to the bottom of the form. This is what you're after. The only property you need to change at this point is the `Name`, which should be set to `sbarStatus`. You can access the `Name` property in either the Properties window or the Actions dialog that is displayed with the smart tag.

7. Now you can add two `Panel` controls to the remaining area of the form. With the first, set the following properties:

❑ **Name** — `pnlNavigation`

❑ **BackColor** — `MenuBar`

❑ **Dock** — `Left`

This panel will contain the navigation buttons that users will use to access the various areas of your application. The `BackColor` property of `MenuBar` is found in the System color tab and will change if users change their systemwide color scheme.

The second panel will fill out the remainder of the form. To do this, set its `Dock` property to `Fill`. To finish the job, change its name to `pnlMain`. This name will be used when you tell the program to create and display controls in response to the user's requests. For example, when the user clicks the Show List button, this panel will be filled with a `PersonList` control (which you will create in a moment).

8. Add two buttons to `pnlNavigation` by dragging and dropping them over the panel control. Set their properties as follows:

❑ **Button #1 Name** — `btnShowList`

❑ **Button #1 Text** — `Show List`

❑ **Button #2 Name** — `btnAddPerson`

❑ **Button #2 Text** — `Add Person`

9. The main form design is done, so save the project and form by selecting File ➪ Save All.

10. Now create the first custom control. Custom controls are special Visual Basic Express files that combine a set of other controls for easier management. In Chapter 2 you saw custom controls in action when you created the starter kit (the `ListDetails` is a custom control, as is the `SearchOnline` component). To create the basic control, select Project ➪ Add User Control. In the dialog window, name the control `PersonalDetails.vb` and click OK to add it to the project.

11. You will see a blank window without borders. Add a `Label` and a `TextBox` to the design surface. Make sure you take advantage of the snap-to guidelines to ensure that the controls are positioned well. A special guideline is shown when moving a `Label` in proximity to a `TextBox` so that you can line up the actual text as opposed to the boundaries of the controls. Change the `Text` property of the label to `First Name`. Change the `Name` property of the `TextBox` to `txtFirstName`.

12. Add the controls for the other fields in your database table, excluding the `GiftCategories` field (that will be added in Chapter 7). Follow the same procedure for `Last Name`, `Home Phone Number`, and `Cell Phone Number`. Because a person's address may span multiple lines, change the `Multiline` property to `True`. Do the same for the `Notes` field.

The only field that requires a different control is the `BirthDate` — use a `DateTimePicker` for this control. Because the standard format includes the day of week, which is inappropriate for a birth date, set the following properties:

❑ **Format** — `Custom`

❑ **Name** — `dtpDateOfBirth`

❑ **Custom Format** — `MMMM dd yyyy`

Refer to Figure 4-10 for the final layout of the control.

Figure 4-10

The base user interface is now done for your Personal Organizer application. In only a few minutes you've created the shell of a program that you will be able to use to maintain information about your friends and family, and you'll extend it in each chapter to provide additional functionality.

Summary

Visual Basic 2005 Express does a lot of work for you, and user interface design is certainly one area that is made easier. The selection of controls that are available at your fingertips is large, and with the numerous customizations you can perform on each one, you can create almost any kind of interface without needing to resort to writing any code at all.

In this chapter, you learned to do the following:

❑   Think about your user interface rather than leave it to the last minute

❑   Use the controls that come with Visual Basic Express to create form designs quickly and easily

❑   Create the main form of your personal organizer application

Exercises

1.   **Anchor fields:** Set the `Anchor` properties on the Address and Notes `TextBox` controls so that they resize automatically when the form is resized.

2.   **Adding the PersonList user control:** In the next chapter you'll need the `PersonList` user control to show the list of people in the database. Create a new user control with a `ListBox` and two `Button` controls. Remember to set the `Anchor` properties so that the fields are resized and positioned when the form's dimensions are changed. Use Figure 4-11 as an example.
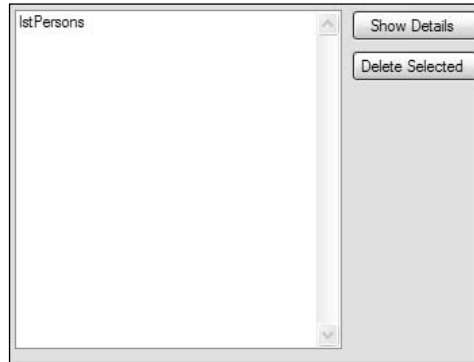

Figure 4-11

5

How *Do* You Make
That Happen?

In the last few chapters, you've learned about database design, along with how to create well laid out user interfaces. However, having a database and the user interface is pointless without the glue in the middle — the actual programming code.

Writing code in Visual Basic Express 2005 is much like writing it in any other programming language, with the extra benefits that the development environment gives you, such as code formatting, automatic completion of programming structures, and a whole lot of IntelliSense to help identify usable members and functions.

In this chapter, you learn about the following:

❑   Visual Basic Express code and how to write fundamental code structures

❑   The aids that Visual Basic Express provides to help you write code

❑   Hooking code to events

Writing Code

Before you get into the nuts and bolts of connecting the user interface to programming logic and database tables, the first thing to do is look at how to write Visual Basic Express code. Creating a program is made a lot easier using Visual Basic Express — the user interface can be designed using the mouse and entering a few values in the Properties window; the database design can be hooked into the program automatically using the Database Explorer and then hooked into some user interface components with `Data` controls (this is covered in Chapter 7); even the creation of code that hooks the user interface object's events can be generated by the environment by simply double-clicking the control.

However, despite all of this, you still need to understand, and be able to write, program code. Fortunately, the Basic language has always been one of the easiest to follow, and Visual Basic Express has combined it with solid object-oriented principles to make an incredibly robust but easy-to-use programming language.

The Basics of Basic

Variables are objects that store information. A variable in Visual Basic Express can contain something as simple as a number or as complex as a date or a fully constructed object, such as a user interface component. *Data types* identify the type of object a variable can be. The standard data types available in Visual Basic Express are as follows:

❑   **Boolean** — Variables of this type can store either `true` or `false`. You can use these as on-off flags to determine when program logic should be performed.

❑   **Byte** — This is a number in the range of 0 through 255. The `SByte` data type can contain signed integer values from -128 through 127.

❑   **Char** — Normally used for single characters, `Char` variables actually store numbers in the range of 0 through 65,535.

❑   **Date** — `Date` variables can store dates and, optionally, time values.

❑   **Decimal** — Variables declared as decimals are precise numbers that can include a value after the decimal point (unlike `Integer` and other whole number types). The range of values is $+/-1E^{-28}$ through to $+/-7.9E^{28}$.

❑   **Double** — The `Decimal` data types described above are relatively new to the Basic language. Before their introduction, `Double` variables were used to store values with fractional components. The precision of the `Double` data type is not as accurate as that found in `Decimal`; however, the range is much larger — $4.94E^{-324}$ through to $1.79E^{308}$.

❑   **Integer** — Integers have always been used in Basic, but in Visual Basic Express they are 32-bit numbers, which means they can store much greater values than previous versions of Basic. The range of number values that can be stored in an `Integer` variable is -2,147,483,648 through 2,147,483,647.

❑   **Long** — An abbreviation for *Long Integer*, variables of this data type can store 64-bit numbers — a range of $-9.2E^{18}$ through $9.2E^{18}$. Again, as the name implies, this data type can handle only whole numbers.

❑   **Short** — In previous versions of Basic, an integer was capable of storing only 16-bit values. The `Short` data type retains that data type and has a range of -32,768 through 32,767.

❑   **Single** — Similar to `Double`, `Single` data type variables are used to store numbers that have fractional components. The precision is not as great as either `Decimal` or `Double` and the range is $1.4E^{-45}$ through $3.4E^{38}$.

❑   **String** — `String` variables are used to store text of varying lengths. While previous versions of Basic may have had a limitation that could be broken, Visual Basic Express strings can store an amazing 2 billion characters — surely enough for the greediest person.

Besides these core data types, you can create your own structures that can then be used as variable types, as well as objects of any kind. In addition, for the integer data types — `Integer`, `Long`, and `Short` — there are unsigned versions that increase the range of positive number values possible by sacrificing the capability to store negative numbers.

Using Variables

To use a variable in your code, you must first tell Visual Basic Express that you want it. This is known as *declaring* the variable. You must tell Visual Basic the name of the variable and the data type you require, in the form of `Dim VariableName As VariableType`. For example, to create a variable that is to store integer values and has a name of `MyNumber1`, you would use the following line of code:

```
Dim MyNumber1 As Integer
```

Dim is a keyword that tells the Visual Basic Express compiler that you are declaring a variable. *As* is also a keyword and identifies the location of the data type the variable is to represent.

All variables start out with a default value. Numeric data types such as `Integer` are initialized to zero, the `Boolean` data type has a default value of `False`, and `String` variables contain an empty string. Use an *assignment* operation to change the value of a variable. Assignment tells the compiler that the variable on the left-hand side of the operation should store the value on the right-hand side. In Visual Basic Express, the assignment operator is the equals sign (=), so to tell the compiler that you want the `MyNumber1` integer variable to have a value of 3, you would write the following code:

```
Dim MyNumber1 As Integer
MyNumber1 = 3
```

While assigning values to variables can be performed at any point in the program logic, often you need to initialize the variable to something other than the default value for that data type. Visual Basic Express provides a neat shortcut for this process of initializing variables by first declaring the variable as a particular data type, and then assigning its initial value all on one line. Declaring a second `Integer` variable and initializing it to a value of 5 could be done like so:

```
Dim MyNumber1 As Integer
Dim MyNumber2 As Integer = 5
MyNumber1 = 3
```

Values such as 3 or 5 are known as *literal* because they represent a literal value that does not change. Using literal values might be good for initializing variables, but you'll often need to change the value to something else. To that end, variables can also be assigned the value of other variables. If, for example, you wanted to change the value stored in `MyNumber2` to the value in `MyNumber1`, you would use an assignment operation with `MyNumber2` on the left-hand side and `MyNumber1` on the right:

```
MyNumber2 = MyNumber1
```

Assignment operations can also assign the result of a function or other operation to the variable on the left-hand side. The standard mathematical operations such as addition (+), subtraction (–) and multiplication (*) are common, but other operations can be performed as well. Changing the preceding line of code to `MyNumber2 = MyNumber1 + 1` informs the compiler that `MyNumber2` should save the result of the operation on the right-hand side. In this case, the compiler would retrieve the value from `MyNumber1` (3) and add 1 to it to get a final result of 4, which would then be stored in `MyNumber2`.

71

Operations do not have to contain literal values at all. Using the `MyNumber1` and `MyNumber2` variables from the preceding example, you could store the product of the values in a third variable like so:

```
Dim MyNumber1 As Integer
Dim MyNumber2 As Integer = 5
Dim MyResult As Integer
MyNumber1 = 3
MyResult = MyNumber1 * MyNumber2
```

After this code were executed, `MyResult` would contain a value of 15 (3 multiplied by 5).

Creating Functions

You often will need to use the same code multiple times in different parts of your program. Rather than rewrite the code in multiple locations, you can encapsulate it as a subroutine or function and then call that from the different spots in your code. If the code is standalone and doesn't need to communicate back to the code that called it, then it can be created as a *subroutine*; otherwise, if it needs to return a value, then it should be defined as a *function*.

Declaring a subroutine is performed by using the `Sub` keyword followed by the name of the subroutine to be created. The name of the subroutine should be followed by a set of parentheses, but if you press the Enter key without them, Visual Basic Express will automatically add them for you. You also need to tell the compiler where the end of the subroutine is, which you do by adding a new line with the keywords `End Sub`. Visual Basic Express jumps to your aid with this, too, so creating a routine called `SayHello` could be done as easily as typing `Sub SayHello` and pressing Enter. This would generate the following code:

```
Sub SayHello()

End Sub
```

Any code that is enclosed between the `Sub` and `End Sub` lines is executed whenever the subroutine is called. Functions are declared in a similar fashion, but like variables, they must also be defined as a specific type. In addition, instead of `Sub` (short for Subroutine) the keyword `Function` is used, so a function that returns a number might look like this:

```
Function CalculateAge() As Integer

End Function
```

When using functions, you need to tell Visual Basic what value should be sent back to the code that called it. The `Return` keyword is used to do this, and as soon as it is executed, the code will set the value associated with the function and return back to the calling routine. This means you can have multiple places within the function that "return" based on different conditions.

While calling a function or subroutine in isolation may work in some cases, often you will need to tell it to use particular values. This is done by defining a parameter list between the parentheses in the routine's definition. Each parameter needs to be defined in a similar way to a normal variable, but because the compiler knows that the code between the parentheses is going to be parameter definitions, you do not need to use the `Dim` keyword.

When parameters are passed to a function or subroutine, Visual Basic needs to know whether it's the whole variable or just the value stored in the variable. The `ByVal` keyword tells the compiler that it

should use only the value of the variable being passed to the routine. As this is the usual way of using parameters, Visual Basic Express automatically inserts the keyword if you forget to do it. This looks like the following:

```
Function GetMeatStockCount(ByVal IncludeGoat As Boolean) As Integer

End Function
```

The `ByRef` keyword identifies objects that should be used in the routine. This means items being passed in with `ByRef` can be changed by the code within the routine. The best way to explain this is to illustrate it with an example:

```
010 Dim MyNumber1 As Integer = 5
020 Dim MyNumber2 As Integer = 6
030 Dim MyResult As Integer
040 MyResult = SomeFunction(MyNumber1, MyNumber2)
050
060 Function SomeFunction(ByVal FirstNum As Integer, ByRef SecondNum As Integer) _
   As Integer
070    FirstNum = FirstNum * 2
080    SecondNum = SecondNum + 5
090    Dim TheResult As Integer = SecondNum + FirstNum
100    Return TheResult
110 End Function
```

The preceding sample code is presented with code line numbers to better explain what's going on. By default, Visual Basic Express does not display line numbers alongside the code, but you can enable this option in the Options window. You'll find the option in the Editor options for the Basic language.

The function `SomeFunction` takes two integers as parameters and performs several calculations against them, returning the final result to the calling code. As the code is executed, the following table follows the values of the variables as the code is executed, showing how they change in the function and after it returns to the main program.

Line	MyNumber1	MyNumber2	MyResult	FirstNum	SecondNum	TheResult
010	5	N/A	N/A	N/A	N/A	N/A
020	5	6	N/A	N/A	N/A	N/A
030	5	6	0	N/A	N/A	N/A
040 ⇨ 060	5	6	0	5	6	N/A
070	5	6	0	10	6	N/A
080	5	11	0	10	11	N/A
090	5	11	0	10	11	21
100	5	11	0	10	11	21
100 ⇨ 040	5	11	0	10	11	21

Did you notice how the value of `MyNumber2` changed when `SecondNum` changed in line 080? This is because the function definition specified that the second parameter was being passed `ByRef`, meaning the whole variable is being worked with as opposed to only the value. As a side note, line 090 shows how the declaration and initialization of a variable can be combined with an operation.

Visual Basic Express uses two ways of passing parameters to a function. The default is `ByVal`, *which actually passes a copy of the variable, rather than the variable itself. This means it doesn't change the variable contents outside the function. The other option is* `ByRef`, *which tells Visual Basic to pass the actual variable.*

To confirm the concepts outlined so far in this chapter, use the following Try It Out to create a simple program that converts temperatures in Fahrenheit to Celsius.

Try It Out   Writing Code #1

1. Start Visual Basic 2005 Express and create a new Windows Application project. Once the Design view of the form is shown, add a button and a text box to the form with the following properties:

   ❏ **Button Name** — `btnConvert`

   ❏ **Button Text** — `Convert`

   ❏ **TextBox Name** — `txtDegrees`

2. Double-click the button. Visual Basic Express will automatically create a subroutine that will be executed whenever the user clicks the button. You'll look at how to do this later in this chapter.

3. In the subroutine, enter the following code:

```
Dim Result As Decimal
```

   Note that when you leave the line, Visual Basic Express will draw a colored wavy line underneath the word `Result` (as shown in Figure 5-1, without the color). This is one of the many visual cues Visual Basic Express provides you to make writing code easier. The color indicates that the variable is not currently being used.


```
Public Class Form1

    Private Sub btnConvert_Click(
        Dim Result As Decimal

    End Sub
End Class
```

Figure 5-1

4. Immediately after the line where you declared the `Result` variable, type these two lines:

```
Result = FahrenheitToCelsius(CType(txtDegrees.Text, Decimal))
txtDegrees.Text = Result.ToString
```

   As you press Enter after the first line, the gray line underneath `Result` on the declaration line is removed because the variable is now used. However, now a blue wavy line appears underneath the function name `FahrenheitToCelsius` (see Figure 5-2)! Don't worry — a blue line indicates that Visual Basic Express has found something that is not declared. You'll create this function in the next step, so it's fine as is.

```
Public Class Form1

    Private Sub btnConvert_Click(ByVal sender As System.Object, ByVal
        Dim Result As Decimal

        Result = FahrenheitToCelsius(CType(txtDegrees.Text, Decimal))
        txtDegrees.Text = Result.ToString

    End Sub
End Class
```

Figure 5-2

If you ever forget what a particular visual cue means, hover the mouse cursor over the displayed indicator. Visual Basic Express will show a tool tip explaining what is occurring.

> The other thing to note about this line is that the `FahrenheitToCelsius` function is being called with a parameter of `CType(txtDegrees.Text, Decimal)`. `CType` is a built-in function that enables you to convert one variable type to another. Because `txtDegrees.Text` returns a `String`, you need to convert it to a `Decimal` value before calling the function. Visual Basic will first call the `CType` function with the parameters of `txtDegrees.Text` and `Decimal`, and then call `FahrenheitToCelsius` with the result that is returned. You can call functions with other functions as many times as you want — it's called *nesting* — but usually one level, as shown in this example, is enough before it gets a bit confusing to read.

The requirement to convert the variable contents to `Decimal` *is true only if you have Option Strict On, as suggested in Chapter 2. I recommend this so that you always know what your variables are, but you can let Visual Basic Express automatically convert between data types it knows how to handle by using Option Strict Off instead.*

> The second line of code assigns the result of the call to the function to the `Text` property of the text box. Again, to explicitly indicate to the compiler that you know what you're doing, use the `ToString` method to convert the decimal value to a string. All objects have a default `ToString` method that returns a visual representation of their contents.

5. Create the `Fahrenheit` function by typing its definition directly after `End Sub`. As the previous code suggests, the function needs to be defined with a single parameter that is a `Decimal` data type, and it will return a value that is also a `Decimal` data type. Remember that you don't need to type the `ByVal` keyword because Visual Basic Express will do it for you:

```
Private Function FahrenheitToCelsius(ByVal FDegrees As Decimal) As Decimal

End Function
```

> The only word you may not recognize here is `Private`. This tells Visual Basic Express that this function is available only in the module in which it was created. This is known as an *access modifier*. You'll look at other access modifiers later in the book.

6. In between the `Function` and `End Function` lines, type the following code:

```
Dim CDegrees As Double

CDegrees = (5 / 9) * (FDegrees - 32)
Return CType(CDegrees, Decimal)
```

While this could also be achieved in a single line, `Return (5 / 9) * (FDegrees – 32)`, it's always good practice to do calculations separately from the `Return` command in case something goes wrong. First a variable of type `Decimal` is declared and then the standard algorithm to convert between Fahrenheit and Celsius is used. The result is then returned to the code that called the function.

7. Run the program by pressing F5 or selecting the Debug ⇨ Start command. Enter a value in the text box such as 98.6 and click the Convert button. The contents of the text box will change to the Celsius equivalent, as shown in Figure 5-3.
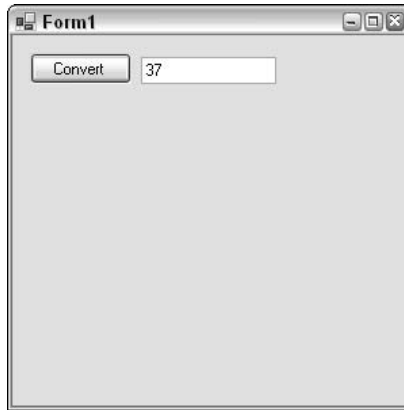
Figure 5-3

Once you've tested the application with a few different values, close it by clicking the red X and return to the Visual Basic Express IDE. From here, save the project with File ⇨ Save All, naming the solution something that you will remember, such as Temperature Converter. Leave it open because you'll use it again later in this chapter.

This application illustrates the use of function calls, variable declarations, and assignment operations, including mathematical functions.

Want Something More?

Creating variables and performing operations such as mathematical algorithms might be all you need, but it's pretty limiting without a few more concepts under your belt. It would be nice to be able to perform the code only under certain conditions, or to be able to repeat a block of code multiple times.

Conditional Logic

Conditional logic enables you to customize the way the program runs depending on the value of a variable, operation, or function. In the same way a nightclub might not let you in until you've reached the age of 21, using a *decision statement* in your code means the code inside the condition block will be performed only if the condition is met. The standard condition keyword in Visual Basic Express is `If`. This is followed by the code representing the decision that the program must make and is finished with the keyword `Then`. This results in the form `If TheConditionIsTrue Then`. Representing the nightclub analogy in code might look like this:

```
If CheckAge(Customer) > 21 Then
```

This is then followed by the code that is to be performed if the condition is `true`. If there is only one operation to be performed, it can be written on the same line as the `If`, like so:

```
If CheckAge(Customer) > 21 Then AllowEntry = True
```

However, if the condition requires a block of code to be performed, then you enclose the lines between the `If` line and an `End If` line, as shown here:

```
If CheckAge(Customer) > 21 Then
    AllowEntry = True
    OpenDoor()
End If
```

Under some circumstances, you might want to perform code if the condition is not true — a sort of "do this if it's true; otherwise, do that" formula. To identify code that should be executed only when the condition is not met, use the `Else` keyword. Any code between the `Else` and `End If` keywords will be performed only if the condition is not true:

```
If CheckAge(Customer) > 21 Then
    AllowEntry = True
    OpenDoor()
Else
    AllowEntry = False
    CloseDoor()
End If
```

To check out how this works, the following Try It Out customizes the Temperature Converter application so that you can convert temperatures in both directions.

Try It Out   Adding Conditional Code

1. Return to the Design View of the form by clicking on the Form1.vb [Design View] tab in the IDE. Add a `CheckBox` control to the form and set the following properties:

❑   **Name** — chkCelsiusToFahrenheit

❑   **Text** — Celsius to Fahrenheit?

2. Double-click the Convert button to return to the code view where the Click event is handled. In the subroutine, you'll need to add an `If` condition to determine the state of the `CheckBox` and perform the appropriate function depending on its state:

```
Private Sub btnConvert_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnConvert.Click
    Dim Result As Decimal
    If chkCelsiusToFahrenheit.CheckState = CheckState.Checked Then
        Result = CelsiusToFahrenheit(CType(txtDegrees.Text, Decimal))
    Else
        Result = FahrenheitToCelsius(CType(txtDegrees.Text, Decimal))
    End If
```

```
        txtDegrees.Text = Result.ToString

   End Sub
```

Now, if the `CheckBox` is checked, the code will store the results of a new function, `CelsiusToFahrenheit` in the `Result` variable; otherwise, it will perform the function you already created previously.

3. Create the new function so it reverses the calculation:

```
Private Function CelsiusToFahrenheit(ByVal CDegrees As Decimal) As Decimal
    Dim FDegrees As Double

    FDegrees = ((9 / 5) * CDegrees) + 32
    Return CType(FDegrees, Decimal)
End Function
```

4. Run the application, enter 98.6 in the text box, and click the Convert button to confirm that the original calculation is still performed. Now, check the checkbox and click the Convert button again. The value will be converted back to the original 98.6.

5. Close the application to return to the design environment. Remember to save it again so you don't lose your work.

You can perform almost any kind of conditional logic programming through the use of `If-Else-End If` blocks, but Visual Basic Express provides you with a couple of shortcuts to make your code more readable (and efficient to create).

First, if you have multiple sets of conditions that are mutually exclusive, the contraction `ElseIf` can be used in a chain of conditional blocks. As an example, consider a scenario in which cakes are allocated shelf space based on their type:

```
If Cake.Type = "Chocolate" Then
    AllocateShelf(Cake, TopShelf)
ElseIf Cake.Type = "Banana" Then
    AllocateShelf(Cake, BottomShelf)
Else
    AllocateShelf(Cake, MiddleShelf)
End If
```

The second abbreviation for `If` condition logic is the `IIf` command. The `IIf` command is like a `If-Else-End If` block all in one line, useful for assigning different values to a variable based on a condition. The form of the `IIf` statement is `IIf(Condition, TrueValue, FalseValue)`. If chocolate cakes were priced at \$2 but all other cakes were only \$1, the price of a particular cake could be calculated as follows:

```
Price = IIf(Cake.Type = "Chocolate", 1, 2)
```

There is another decision logic structure that enables multiple conditions to be checked against one variable. The `Select Case-End Select` statement block enables the code to check a variable's value with multiple cases, performing different operations in each instance. Each condition is prefixed with the keyword

Case and the value that should be matched. There is a special case of Else that is met if no other conditions are matched. The multiple conditions in the previous cake example could be rewritten using the Select Case statement like so (with carrot cake added to the trash):

```
Select Case Cake.Type
    Case "Chocolate"
        AllocateShelf(Cake, TopShelf)
    Case "Banana"
        AllocateShelf(Cake, BottomShelf)
    Case "Carrot"
        AllocateShelf(Cake, Trash)
    Case Else
        AllocateShelf(Cake, MiddleShelf)
End Select
```

Looping Logic

Another fundamental concept in programming languages is being able to repeat a set of commands a set number of times. This is commonly known as a *program loop*, and Visual Basic Express provides a variety of ways to implement this.

The first and most common program loop is the For-Next code block. A variable is defined as a counter and is used to control how many times the code block should be executed. The general structure is as follows:

```
For VariableName = Start To Finish
    ...code to be repeated
Next
```

The Start To Finish portion of the For line identifies the starting and ending points of the loop's counter, so if Start were a value of 1 and Finish were a value of 10, the code inside the For-Next block would be executed 10 times.

Start and Finish can be literal values or variables (or even the result of function calls!), while the current value of VariableName can be referenced within the code block if necessary. The For statement has an optional parameter named Step that specifies the number to increment by each time around in the loop. This is placed after the ending value with the keyword Step, followed by the number:

```
For VariableName = Start To Finish Step IncrementValue
```

Previously, the variable for the loop had to be declared in a separate location in the code. Visual Basic Express does away with needless coding though and enables you to define the variable within the For statement itself. The variable is declared in much the same way as if it were being declared on a separate line:

```
For VariableName As DataType = Start To Finish Step IncrementValue
```

The following table contains some sample `For-Next` loops.

```Dim Counter As Integer```
 	
 ```For Counter = 1 To 10```
 	
 ```Next```
	Defines an integer variable named `Counter`.
	
	 Uses the previously declared `Counter` to loop 10 times.

```Dim MaxValue As Long = 1000```
 	
 ```For Counter As Long = 50 To MaxValue```
 	
 ```Next```
	Defines a long integer named `MaxValue` and initializes it to 1,000.
	 Creates a long integer variable named `Counter` and uses it to count from 50 to the value stored in `MaxValue` (that is, 1,000).

```For Counter As Integer = 1 To 30 Step 5```
 	
 ```Next```
	Creates an integer variable named `Counter` and uses it to count from 1 to 30 in increments of 5. Note that this will loop 6 times, with the last iteration having a value of 26.

A variation of the standard `For-Next` block is for use with collections of objects. For example, you may need to perform an operation on every record in a database table. To do this, use the `For Each` statement that is of the form `For Each Object As ObjectType In CollectionName`. The database example would be implemented like this:

```
For Each CurrentRecord As DataRow In PersonTable
    ...code to be performed on each record
Next
```

When using the `For Each` construct, when the code within the loop is being performed, the `CurrentRecord` object is defined as the type specified in the `For Each` line and is populated with each object in the collection in turn.

Two other loop constructs can be used in Visual Basic Express. Both follow the same pattern — perform the code contained within the looping block while a certain condition exists. However, the condition checking for each construct occurs at a different point. The `While` statement will iterate through the code while the condition specified is true, and the `Do Until` statement performs the code until the condition specified is met.

The `Do Until` loop block can be specified in one of two ways, with the difference being when the condition is checked:

```
Do Until Condition
    ...code to be performed
Loop
```

or

```
Do
    ...code to be performed
Loop Until Condition
```

The following three example loops look like they do the same thing:

```
While VariableOne < VariableTwo
    VariableOne = VariableOne + 1
End While
```

```
Do Until VariableOne >= VariableTwo
    VariableOne = VariableOne + 1
Loop
```

```
Do
    VariableOne = VariableOne + 1
Loop Until VariableOne >= VariableTwo
```

There is a difference in the outcome for the third loop — can you see it? In the first one, the loop will only be performed if VariableOne is less than VariableTwo when it is first entered. Likewise, the second loop will be executed only if VariableOne contains a value less than VariableTwo. However, in the last loop, if VariableOne is already greater than VariableTwo, the loop will still be performed the first time because the condition is not tested until the end of the loop.

Using all of these fundamental constructs in your code, you can perform complex logic that incorporates conditional and looping statements that control how the code should be executed.

Events

An event in the real world is an occurrence with significance. It might be the sun rising, or a switch flicked on, or a car door closing. Regardless of what has occurred, anyone can take notice of it and react based on what has happened. In Visual Basic Express programming, every object has specific activities associated with it. These are called *events* and can be tracked by other objects in the program.

When an object raises an event, it sends a message to the system to notify it that something has occurred. In Visual Basic Express, code can be written to execute when such an event has been fired. This is called *handling* the event, and it is accomplished by creating a subroutine that has the same *signature* as the event structure.

Every event has a specific structure that accompanies it. For example, a FireAlarm event may include a parameter to indicate the building sector in which the fire is occurring. This structure is commonly known as the *event signature,* and any subroutine that wishes to handle a particular event must have the same signature of included parameters or the compiler will reject it.

A subroutine designed to handle an event is written in the same way as a regular subroutine:

```
Sub RoutineName(ParameterList)
End Sub
```

For example, handling a button object's Click event requires a subroutine with two parameters, a System.Object and an EventArgs object:

```
Sub MyButtonClickRoutine(ByVal sender As System.Object, _
    ByVal e As System.EventArgs)

    MessageBox.Show("My Button was clicked")
End Sub
```

81

Once the routine has been declared, it then needs to be connected to the event. To connect the two, an additional clause is added to the end of the subroutine definition. The `Handles` keyword tells the compiler that this subroutine is to intercept events that are named after it. To connect the preceding subroutine with the `Click` event of a button named `Button1`, the definition would be altered as follows:

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

    MessageBox.Show("My Button was clicked")
End Sub
```

This method of hooking object events with suboutines through a `Handles` clause enables a program to handle multiple events with one subroutine and have multiple subroutines all handling the same event. In the following Try It Out, you'll create a set of subroutines that handle different events to illustrate this capability of Visual Basic Express.

Try It Out   Writing Event Handlers

1. Create a new Windows Application project in Visual Basic Express. Once the form has been displayed in Design view, add two button objects to the form. Set their properties as follows:

    ❑   **First button Name** — `btnOne`

    ❑   **Second button Name** — `btnTwo`

2. Double-click on `btnOne` to have Visual Basic Express automatically add a subroutine that will handle the `Click` event of the button. When the code view is shown, enter the following line of code:

```
MessageBox.Show("First button clicked!")
```

3. Return to the Design view and this time double-click `btnTwo` to generate a default event handler for the `Click` event of that button. Enter the following line of code:

```
    MessageBox.Show("Second button clicked!")
```

4. Finally, create your own subroutine directly after the `btnTwo_Click` routine with this code:

```
Private Sub ButtonClicked(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnOne.Click, btnTwo.Click
        MessageBox.Show("A button was clicked!")
End Sub
```

5. When you run this application, clicking on either button will produce two messages. Clicking the first button will display `First button clicked!` as well as `A button was clicked!`, while clicking the second button will display `Second button clicked!` along with the `A button was clicked!` message. This example illustrates how multiple events can be handled by one routine (`ButtonClicked`), and how events can be handled by multiple routines.

When two or more routines are handling the same event, there is no guaranteed order in which they will receive it. When Visual Basic Express compiles the code, it assembles the event hooks, but it's up to the system to send the event notifications when they occur.

Objects: A Special Case

Objects, both those that are part of Visual Basic Express and the ones that you create yourself, are handled slightly differently when you declare them. This is because initially when an object is declared, it doesn't actually contain anything. Rather, it is a definition of the object waiting to be filled with something; put another way, it's a potential object waiting to happen.

An object is defined in the same way as a data type variable, such as the following:

```
Dim MyButton As Button
```

and

```
Dim APersonRecord As Person
```

However, before you use it in program logic, it must be initialized, either by assigning it the value of another object of the same type, or by creating a new instance of the class. This latter method is achieved using the `New` method, which needs a class type to identify what type of object should be created:

```
Dim APersonRecord As Person
APersonRecord = New Person
```

Like the initialization of data type variables, these two lines can be abbreviated into one:

```
Dim APersonRecord As Person = New Person
```

It can be abbreviated even further if the object types are the same:

```
Dim APersonRecord As New Person
```

Objects usually have a number of events, methods, and properties. *Methods* are called in a similar way to functions and subroutines. Method is the correct name for a function that belongs to an object; in reality, it is a function. If the `Person` class used as an example here had a `GoToSleep` method that accepted a parameter of `NumberOfHours` and returned a `Boolean` flag to indicate whether the Person did indeed go to sleep, it could be used like so:

```
DidItWork = APersonRecord.GoToSleep(7.5)
```

Because of this, object methods can be used anywhere functions can be—as part of conditional logic, or looping code blocks, assignments, and operational statements. Properties of objects are similar to variables. In fact, most properties are data type variables. The `Person` class might have a `FirstName` `String` variable and a `BirthDate` `Date` variable, but this is not always the case. A property of an object can be an entirely new object, which in turn can have properties that are objects, and so on.

Applying the Knowledge

The Personal Organizer application is designed to enable you to keep track of information about your friends and family members. It stores their names, addresses, phone numbers, and birth dates, along with a notes area so you can remind yourself about particularly important information. As you progress

through the chapters in this book, you will add more functionality to the application to allow for reminders and backing up the data.

So far you've created the database structure for this section of the application, along with a simple user interface. The two pieces are disconnected, however, so you'll need to create some code that will populate the user interface components with the contents of the database.

Before that can be done, the user interface itself needs to have some code written so that the appropriate parts of the application are displayed when the user clicks the various buttons. The following Try It Out creates the code to add instances of the user controls you created in Chapter 4 to the main form layout, and to indicate when the two buttons are clicked. It will demonstrate the use of subroutines that handle events as well as the creation of objects and conditional logic to remove user controls that are not being used.

Try It Out   Connecting User Interface Elements

1. Open the Personal Organizer project you created in Chapter 4 and go to the Design view of the `MainForm.vb` file by right-clicking its entry and selecting View Designer from the menu. Make sure you include the additions made in the Exercise section of that chapter so that you have both the `PersonList` control and the `PersonalDetails` control.

    If you haven't created the code from Chapter 4, you can find a starting point solution in the downloaded code for this book in `Chapter 5\Personal Organizer Start`.

2. You need to show the user controls when the user clicks the buttons, so double-click the Show List button to have Visual Basic Express automatically generate a subroutine that handles the `Click` event. When the user clicks on this button, you will need to show the `PersonList` control, but the `MainForm` object knows nothing about `PersonList` objects.

3. In the subroutine, declare an object of type `PersonList` and then initialize it with a new instance of `PersonList`:

```
Private Sub btnShowList_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnShowList.Click
        Dim objPersonList As PersonList
        objPersonList = New PersonList
End Sub
```

4. You now have a `PersonList` object in the main part of your program, but it needs to be added to the form's structure so the user can see it. The `MainForm` user interface has a `Panel` object named `pnlMain`. This was positioned and docked so that it takes up the central area of the form — perfect for the `PersonList` object.

    Because the `Panel` control is a container control, you can add controls to it in code by accessing the `Add` method of its `Controls` collection property, like this:

```
Private Sub btnShowList_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnShowList.Click
        Dim objPersonList As PersonList
        objPersonList = New PersonList

        pnlMain.Controls.Add(objPersonList)
End Sub
```

5.   Run the program by pressing the F5 key or using the Debug ⇨ Start menu command, and click the Show List button. Lo and behold, an instance of the `PersonList` control is created and then added to the main panel area.

6.   However, the problem with this implementation is that because the `PersonList` control was created in the `Click` event, it's inaccessible from anywhere else in the code. To change the scope of the `objPersonList` variable, stop the program and return to the code view of `MainForm.vb`. Remove the definition of the `objPersonList` from the `btnShowList_Click` routine, and, scrolling to the top of the code listing, insert the following definition before the start of the routine:

```
Private objPersonList As PersonList
```

When you define a variable or object outside the subroutine and function definitions, you need to specify its context. As you want it accessible only within the main form, the `Private` access modifier tells the compiler that all code within the `MainForm.vb` module can access the object but nothing outside the module can see it at all.

7.   Return to the Design view of `MainForm.vb`, and this time double-click the Add Person button. In the resulting routine that is generated, add the following code to create a new instance of the `PersonalDetails` user control and add it to the `pnlMain` component:

```
Private Sub btnAddPerson_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnAddPerson.Click
        objPersonalDetails = New PersonalDetails

        pnlMain.Controls.Add(objPersonalDetails)
End Sub
```

8.   Note that the instances of `objPersonalDetails` are marked with blue wavy lines, indicating that the variable has not been declared, so insert the definition of the object directly after the definition of `objPersonList`:

```
Private objPersonList As PersonList
Private objPersonalDetails As PersonalDetails
```

9.   Now run the application again and click the Add Person button to see the instance of the `PersonalDetails` user control displayed. There's a problem, however — if you click the Show List button, the `PersonList` control doesn't seem to be displayed. In fact, it is being displayed, but it's hidden by the `PersonalDetails` control already present in the panel. This is why you moved the definition of the objects so that they could be accessed from anywhere in the code.

10.   Stop the application and return to the code view of the form. If the user clicks the Show List button, then logically they don't want to see the Personal Details screen, so you'll need to write some code that determines whether this other object exists, and remove it if so.

Unfortunately, this is one condition that Visual Basic is not very good at — checking for the existence of an object. To determine whether an object does *not* exist, you can compare it to the special keyword `Nothing`, such as `If MyObject Is Nothing` Then. Previously, the way to check if it has been initialized is to reverse this condition with the `Not` operator:

```
If Not MyObject Is Nothing Then
```

In Visual Basic Express, it's a little better, but not much:

```
If MyObject IsNot Nothing Then
```

Once you have determined that the `PersonalDetails` object exists, first remove it from the `pnlMain's Controls` collection and then reset it so that it no longer contains an object by assigning `Nothing` to it. Your code should now look like this:

```
Private Sub btnShowList_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnShowList.Click
        objPersonList = New PersonList

        If objPersonalDetails IsNot Nothing Then
            pnlMain.Controls.Remove(objPersonalDetails)
            objPersonalDetails = Nothing
        End If
        pnlMain.Controls.Add(objPersonList)
End Sub
```

11. Run the application yet again and click the Add Person button followed by the Show List button. This time around, you'll find that the `PersonalDetails` control is removed from the form, and the `PersonList` control can be seen.

12. You'll need to repeat the process in the `btnAddPerson_Click` event handler routine. However, rather than use the clunky `IsNot Nothing` condition, you can use a method in the panel to determine whether the other user control currently exists. Because you know that if `objPersonList` is an actual object (as opposed to `Nothing`) it will be part of the `pnlMain's Controls` collection, use the `Contains` method to determine whether it exists, and if it does, remove it and then set it to `Nothing`:

```
Private Sub btnAddPerson_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnAddPerson.Click
        objPersonalDetails = New PersonalDetails

        If pnlMain.Controls.Contains(objPersonList) Then
            pnlMain.Controls.Remove(objPersonList)
            objPersonList = Nothing
        End If
        pnlMain.Controls.Add(objPersonalDetails)
End Sub
```

13. When you run the application now, the user controls will be created and destroyed correctly when either button is clicked.

14. The only thing left to do at this point is to tell the objects to fill the whole panel area allocated to them. This is done by writing code to set the `Dock` property of each object to `Fill`, like so:

```
objPersonList.Dock = DockStyle.Fill
```

Because the `objPersonList` and `objPersonalDetails` objects are reset when the button is clicked, you will need to set this property each time you create them, so insert the previous line at the bottom of the `btnShowList_Click` routine and a similar line setting the `objPersonalDetails.Dock` property at the bottom of the `btnAddPerson_Click` routine.

Run the application and compare it to Figure 5-4. As the user resizes the form, the components of the user control that is currently being displayed also resize to fill the available area.
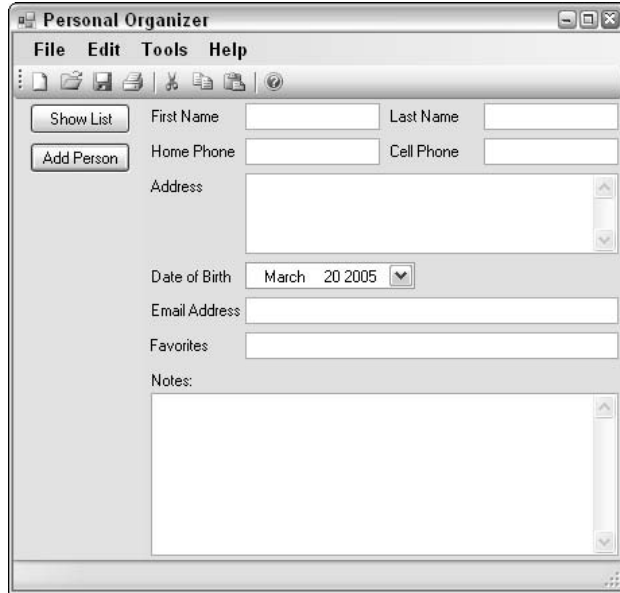
Figure 5-4

You've now created the code required to show the two main components of the Personal Organizer application. As you were creating the code, you undoubtedly noticed that the code editor in Visual Basic Express provided a number of helper features, including the following:

❑ **Gray wavy lines to indicate variables that were defined but not used.** These won't stop the program from compiling and running and will be removed when the object or variable is referenced in the code.

❑ **Blue wavy lines that show you where you have program errors.** Visual Basic Express will not compile while you have errors like these and will automatically add them to the Error List window so you can easily find them all.

❑ **Automatic completion of code blocks such as** `If-End If` **blocks.** Simply type `If` followed by the condition and Visual Basic Express will add the `Then` keyword and a new line containing the `End If` statement.

❑ **IntelliSense pop-ups for properties and methods.** This occurs when you type the name of a variable or object followed by the period (.) to indicate you want to use a member of that object. Visual Basic Express will display a list of commonly used methods and properties that you can scroll through to locate the right one. For example, when you typed `pnlMain`, you would have been presented with a list of properties and methods that are part of the `pnlMain` object. As you select a member in the list, a tool tip will be displayed giving you information about what it does (see Figure 5-5).
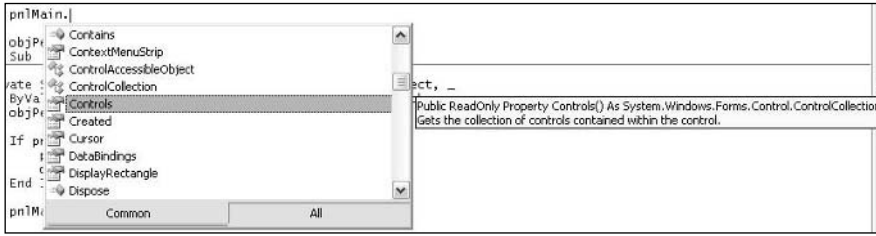
Figure 5-5

❑ **IntelliSense for enumerated types.** Enumerated types are variables that can contain only a set number of values. In the Try It Out, you set the `Dock` property for the two user control objects to `Fill`. When you typed the assignment operator (=), Visual Basic Express presented a list of valid values, which included `DockStyle.Fill`, as illustrated in Figure 5-6.
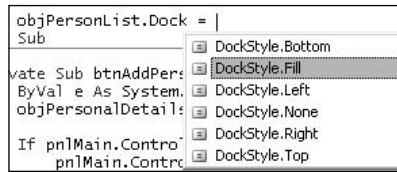
Figure 5-6

❑ **Extended tool tip information as you enter parameters for function and method calls.** For example, when you added the objects to the `pnlMain`'s `Controls` collection, the `Add` method followed by the open parenthesis produced a tool tip with what parameters are expected and what each one is used for (as shown in Figure 5-7).

Figure 5-7

In Chapter 7, you'll connect the user control components with the data source and finally have an application that displays information from a database in a user interface you have created.

Summary

Writing Visual Basic Express code is straightforward. It follows common programming language constructs; and with the various aids that the IDE provides you, you'll find yourself guided to the correct code at almost every step of the way. Armed with the knowledge found in these first five chapters, you can create applications with a good user interface design, coupled with code that responds to user-generated events.

In this chapter, you learned to:

❏ Write Visual Basic Express code, including defining and initializing variables and objects, as well as writing conditional and looping program logic

❏ Use the visual aids that the development environment displays

❏ Handle object events with your own subroutines

In the next chapter, you'll customize the menu and toolbar items, as well as build your own custom classes complete with properties, methods, and events.

Exercises

1. Create an application that changes the color of the text in a `TextBox` control if there are numbers present. To do this, you'll need to write a subroutine to handle the `TextChanged` event of a `TextBox` and set the `ForeColor` property if the condition is met.

2. Create an application that counts from 1 to 100 in increments specified by the user and displays the values in a `TextBox`.

3. Modify the application you created in Exercise 2 so that it ensures the increment is a number before it performs the loop.

Hint: Use the `IsNumeric` *built-in function to determine if a variable is numeric or not.*

Part II
Extending Yourself Is Good

6

Take Control of Your Program

Up until now, the appearance of the user interface was created at design time, with the underlying code being used to control the data that was displayed. In addition to this, you will often need to customize the appearance of the individual controls on your forms while the program is running. This runtime modification of the user interface is the subject of this chapter.

Along with learning about changing your controls at runtime, the following pages discuss the creation and extension of custom controls and classes, including building your own properties, methods, and events.

In this chapter, you learn about the following:

- ❑ Creating custom classes and controls
- ❑ Using events and properties to communicate within your programs
- ❑ Changing controls at runtime

Adding Some Class to Your Program

The programs you created in the first part of this book gave you a fundamental understanding of how to design an application using Visual Basic Express. By looking at the methods used to achieve good form design and getting familiar with creating functions with conditional logic and program loops, you learned how straightforward using this programming language can be.

Now comes the good stuff — classes. Visual Basic Express is an object-oriented language. As such, it enables you to segregate your application into discrete units that act as self-contained objects that interact with each other on defined boundaries and only through prescribed means, whether they are properties, methods, or events. Chapter 2 outlined the definitions of a number of the concepts required to understand class building and offered some tantalizing code snippets suggesting how you can reference classes in the code of your applications.

When you create an application's underlying structure you should think about how to break it down. Consider the following questions:

❑   What's the best way you can describe the various components of the program, the different types of data, and the way things interact with each other?

❑   Is there a discrete object or objects that you can distinguish in the overall design?

❑   Can you draw a diagram that shows individual pieces of data and how they communicate?

❑   Is there an object dependent on another, and if so, should it keep tabs on the other object or wait for the other to inform it of a change?

In Chapter 4 you created the initial user interface of a Personal Organizer application. It contained a main form with a basic menu and toolbar, a couple of buttons, and a big blank area along with a couple of custom controls. In the next chapter you added code to add these custom controls to the form and show them to the user, depending on what button was pressed.

What you may not have realized as you did this was that you were working with special classes in the form of custom-built Windows Form controls. These controls work just like any other class with properties, methods, and events, and the way you added them to the form — with the process of creating a new instance of the control and then adding it to a container — is exactly the same thing you need to do when you work with normal classes, too.

Consider the purpose of the application covered in Chapter 3 — a program that enables you to keep track of information about your friends and family members, including their birthday, e-mail addresses, and other contact information. From a data point of view, you need to keep track of each person; hence, the Person table you created.

That Person table is an obvious candidate for a custom class that you can use to store information. The advantage of defining it as a class is that you can use it in your code, disconnected from the database, and you can include methods and events that belong to that particular person. Conveniently, this class can be used together with the custom control to give you flexibility in processing each person.

The application itself could be defined as another class, with properties keeping track of various settings and program states such as who is using the program and what action was performed last. And because classes can contain other classes as attributes, the `PersonList` control you created in the exercise section of Chapter 4 and then hooked in to the main form in Chapter 5 could be defined as yet another class, complete with a collection of `Person` classes that detail each individual.

Creating Custom Classes

Creating a custom class involves just a few steps. You first create the class module in the Visual Basic Express IDE. Then you define its characteristics through the definition of the variables that will store the data about the class, the functions that will act on the data, and any events to tell the rest of the world something has occurred within the class.

It should be pointed out that the class module that you create can actually contain multiple class definitions, so when you get down to it, you could just add additional class definitions inside your main form's `.vb` file. However, you are encouraged to separate your classes into individual files. The reason is simple.

Using classes enables you to segregate information and actions into a self-contained unit. This unit can then be used by anything that has access to it. If you keep each class in a separate class module, you can then import just the ones you want into each project.

For example, if you wanted to build another application in the future that used the same `Person` class as the Personal Organizer application, it wouldn't be as easy to use if the `Person` class was defined in the main form's file because you would need to include the whole thing. It is usually acceptable to keep classes that work together in one physical file. This means you could keep the `Person` class and the `PersonList` class in the same file if that fits your own style of organization.

> *You can even define classes within classes if that makes sense to your application's design. Internal classes of this type are normally defined as Private so they can be accessed only within the main class' functionality.*

To add a class file to your project, use the Project ⇨ Add Class menu command or right-click the project in the Solution Explorer and choose Add ⇨ Class from the submenu. Either method will present you with the Add New Item dialog with the empty class template highlighted (see Figure 6-1). Name the class something appropriate to the kind of object you are defining and click Add to add it to the project.
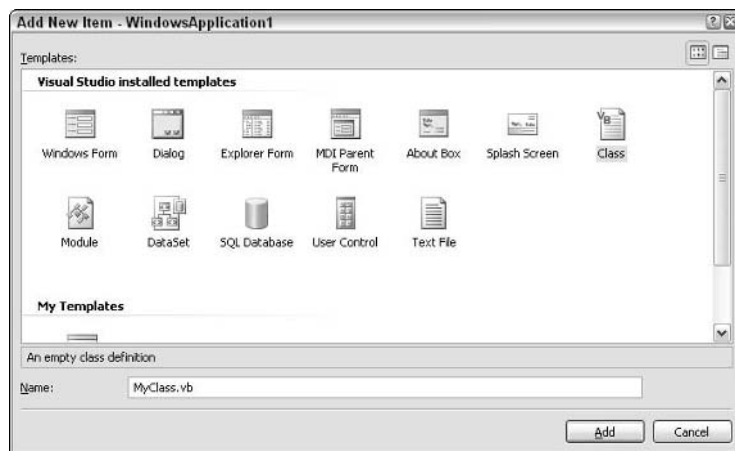
Figure 6-1

The new class file will be added to the Solution Explorer window, and you'll be able to access the code for it through the context menu or by clicking the View Code button at the top of the Solution Explorer. Selecting the class file will also change the context of the Properties window, where you can set a couple of properties that control how the class is built and the name of the file if you wanted to change it later.

When you add a class module, by default it adds an empty class with the same name and defines it as *Public*, which means any other part of the application can reference and use the class, as well as any external program that interfaces with your application. The code view of your class will look like this:

```
Public Class MyClassName

End Class
```

Properties

An empty class doesn't do much; you need to add code to specify its attributes. First up are the variables that store the information. These are usually placed at the top of the class structure and are defined in the same way as module-level variables are in your form code in a Windows Application project.

Variables within classes can be defined with a variety of access modifiers, including `Private` and `Public`. `Private` tells Visual Basic Express that the variable is accessible only within this class and will not be seen outside the class. `Public` is at the opposite end of the spectrum — a public variable can be accessed not only by the class and any other part of your application, but also by other programs as well (assuming they can access the class that the variable is part of).

Other access modifiers include `Friend`, which enables other parts of your program to access the variable (but nothing outside of it can see it), and `Protected`, which is an extension of `Private` that enables classes that inherit behavior from other classes to see their variables.

If you put this in action, the class definition would appear similar to the following:

```
Public Class MyClassName
    Private MyString As String
    Public MyInteger As Integer
End Class
```

In this case, the `MyString` variable would be accessible only from within the class, while other parts of the application could access and change the contents of `MyInteger`.

Classes often embody this kind of division of information, where some data is for internal use only, while other information is provided to the rest of the program. You may be tempted to implement the publicly available data using `Public` variables, but that allows other code to have access to the data in a way you may not want to allow.

These `Public` variables will be visible in the form of properties on the class, but unlike real properties, the code accessing the class can assign whatever data it wants to the variable, thus potentially corrupting your class contents. Real property definitions enable you to control access to the information.

To define a property, you use the `Property` structure, which has the following syntax:

```
Property propertyName() As String
    Get
        Return someValue
    End Get
    Set(ByVal newValue As String)
        Do something with newValue
    End Set
End Property
```

A property must have a name and a type, which specify how it can be accessed from outside the class. Within the property definition, the code needs to define what is returned if code tries to get the value from the property (the `Get` clause), and what action should be taken if another part of the program attempts to assign a value to the property (the `Set` clause).

The sample code can be altered to fit this preferred way of defining a public property, by changing the access modifier on `MyInteger` to `Private` and then returning it and assigning it through a `Property` definition:

```
Public Class MyClassName
    Private MyString As String
    Private MyIntegerValue As Integer
    Public Property MyInteger() As Integer
        Get
            Return MyIntegerValue
        End Get
        Set(ByVal newValue As Integer)
            MyIntegerValue = newValue
        End Set
    End Property
End Class
```

Notice in the preceding property definition that it was actually defined with a `Public` access modifier to explicitly tell the Visual Basic Express compiler that this property is to be accessible from outside the class.

This sample effectively does almost the same thing as giving external code direct access to the private variable. However, you can write whatever code you require in the `Get` and `Set` clauses to control that access. For example, if the value stored in `MyInteger` were allowed to be within a specified range of 1 through 10 only, the `Set` clause could be modified to ignore values outside that range:

```
Public Property MyInteger() As Integer
    Get
        Return MyIntegerValue
    End Get
    Set(ByVal newValue As Integer)
        If newValue >= 1 And newValue <= 10 Then
            MyIntegerValue = newValue
        End If
    End Set
End Property
```

The `Get` clause can be similarly modified if need be. In some cases, you may want to allow access to information to other areas of your program but not allow it to be modified. To disallow write access to a property, use the `ReadOnly` modifier on the `Property` definition:

```
Public ReadOnly Property MyInteger() As Integer
    Get
        Return MyIntegerValue
    End Get
End Property
```

Note that the `Set` clause is not even required (and in fact will cause a compilation error if it does exist) when the property is defined `ReadOnly`. Conversely, some information may be modified via external code, but cannot be retrieved. This may be for security reasons, or just because it's not needed. In either case, use the `WriteOnly` modifier in the place of the `ReadOnly` modifier and specify the `Set` clause instead of the `Get` clause.

Creating an instance of the custom class and accessing the properties defined within it is done using the same syntax as accessing the attributes of a control or form:

```
Dim MySample As New MyClassName
MySample.MyInteger = 6
```

Methods

If the only thing that the class were capable of was defining, storing, and controlling access to information through properties, it would be a powerful feature of programming in Visual Basic Express. But that's just the beginning, and like the methods on controls such as `Buttons` and `TextBoxes`, a class can have its own public functions.

Methods can be either subroutines or functions, and they have the same syntax as both of these structures (covered in Chapter 5). Because methods are part of the internal structure of the class, they can access the private variables defined within the class. Therefore, the sample class definition could be extended like so:

```
Public Class MyClassName
    Private MyString As String
    Private MyIntegerValue As Integer
    Public Property MyInteger() As Integer
        Get
            Return MyIntegerValue
        End Get
        Set(ByVal newValue As Integer)
            MyIntegerValue = newValue
        End Set
    End Property
    Public Sub MyFunctionName(ByVal ExtraParameter As Integer)
        MyIntegerValue += ExtraParameter
    End Sub
End Class
```

Class functions and subroutines are accessed by referencing the object name followed by a period (.) and then the name of the method. As you can see, this method of identifying members of objects is used throughout Visual Basic Express code, and this consistent approach of accessing information makes it easy to read programs. Using the sample property and method, this access is illustrated as follows:

```
Dim MySample As New MyClassName
MySample.MyInteger = 6
MySample.MyFunctionName(3)
MessageBox.Show(MySample.MyInteger.ToString)
```

The result of this code would be a message dialog containing a text representation of the value stored in the `MyInteger` property of `MySample`—9.

Your class structure can also have private functions that are used only internally within the class. These are usually helper functions that perform very specific tasks that do not serve much purpose outside the class.

Events

The cherry on top of the pie defining a class is the capability to define custom events. For any significant occurrence within the class, you can build a notifying action that other code can receive by writing an event handler routine.

Adding an event to a class is a two-step process. First, you need to define the event and identify what information will be included in the message when it occurs. Second, you need to tell the class to raise the event when a particular condition or situation is met.

Event definitions are placed outside any other property or method definition and consist of a single-line statement beginning with the keyword Event and naming the event followed by its parameter list enclosed in parentheses. The syntax is Event EventName(EventParameters) and is demonstrated here by adding an event named MyEvent at the top of the class definition:

```
Public Class MyClassName
    Event MyEvent(ByVal MyBigInteger As Integer)

    Private MyString As String
    Private MyIntegerValue As Integer
    Public Property MyInteger() As Integer
        Get
            Return MyIntegerValue
        End Get
        Set(ByVal newValue As Integer)
            MyIntegerValue = newValue
        End Set
    End Property
    Public Sub MyFunctionName(ByVal ExtraParameter As Integer)
        MyIntegerValue += ExtraParameter
    End Sub
End Class
```

Once the event has been defined, it then needs to be raised at an appropriate time. Events can be designed and raised for all sorts of reasons. Your class may need to raise an event if an error occurs, or it might need to inform the application every time a particular function is performed. You may also need to raise an event every time a particular interval of time has passed.

Telling Visual Basic Express that the event should be fired is done through the RaiseEvent command and has the syntax RaiseEvent EventName(EventParameters). The subroutine in the sample class could thus be modified like this:

```
    Public Sub MyFunctionName(ByVal ExtraParameter As Integer)
        MyIntegerValue += ExtraParameter
        If MyIntegerValue > 10 Then
            RaiseEvent MyEvent(MyIntegerValue)
        End If
    End Sub
```

The final class definition containing private variables, public properties and methods, and event definition appears as follows:

```
Public Class MyClassName
    Event MyEvent(ByVal MyBigInteger As Integer)

    Private MyString As String
    Private MyIntegerValue As Integer
    Public Property MyInteger() As Integer
        Get
            Return MyIntegerValue
        End Get
        Set(ByVal newValue As Integer)
            MyIntegerValue = newValue
        End Set
    End Property
    Public Sub MyFunctionName(ByVal ExtraParameter As Integer)
        MyIntegerValue += ExtraParameter
        If MyIntegerValue > 10 Then
            RaiseEvent MyEvent(MyIntegerValue)
        End If
    End Sub
End Class
```

You saw how the event handler routine side of things is implemented in Chapter 5, but to follow the example all the way through, here is a sample routine that handles the event that is defined and raised in the previous example:

```
Private WithEvents MySample As MyClassName
...
Private Function SomeFunction() As Boolean
    MySample = New MyClassName
    MySample.MyInteger = 6
    MySample.MyFunctionName(3)
    MySample.MyFunctionName(3)
End Function

Private Sub MyEventHandler(ByVal BigNumber As Integer) Handles MySample.MyEvent
    MessageBox.Show("Number getting big: " & BigNumber)
End Sub
```

This code creates an instance of the `MyClassName` class and assigns an initial value of 6 to the `MyInteger` property. It then performs the `MyFunctionName` method twice, each time effectively incrementing the `MyInteger` property by 3, with a result of 9 and then 12.

When the subroutine calculates the value of 12, it raises the event `MyEvent`, which is being handled by the `MyEventHandler` routine, and a message dialog is displayed warning the user that the number is getting big.

You may have noticed the extra keyword required as part of the definition of the class — `WithEvents`. For more information on how `WithEvents` works, see Chapter 9.

The With-End With Block

Sometimes you will want to work with a particular object or control extensively. Rather than type its name each time, you can use a special shortcut Visual Basic Express provides — the `With-End With` block.

The `With` statement identifies a particular variable name to be treated as a shortcut to the compiler. Wherever a property or method is preceded by a single period (.), the compiler will automatically insert the variable identified in the `With` statement. For example, the function definition in the previous example could be replaced with this `With` block:

```
With MySample
  .MyInteger = 6
  .MyFunctionName(3)
  .MyFunctionName(3)
End With
```

You can have only one shortcut variable at any one time, although you can embed `With` blocks inside other `With` blocks. This is particularly useful with very complex objects where you initially work with properties at one level but then need to deal with attributes further down the hierarchy:

```
With MyOtherSample
  .MyString = "Hello"
  With .MyOtherObject
    .MyStringTwo = "World"
  End With
End With
```

You'll find further examples of using `With` blocks throughout this book as a way of saving space. It can make your code more readable, so I encourage you to use `With` in your own applications.

Special Method Actions

As you were typing out code in Chapter 5 and taking notice of the cool IntelliSense features of Visual Basic Express, you may have noticed that some methods appeared to have multiple ways of being called, or multiple signatures. This is known as *method overloading* and is a relatively new addition to the Visual Basic language.

Method overloading enables you to define several functions with the same name but with different sets of parameters. Each function can do completely different things, although it's typical for functions of the same name to perform the same kind of action but in a different context. For example, you might have two methods that add an interval to a date variable, where one adds a number of days, while the other adds a number of days and months. These could be defined as follows:

```
Public Sub AddToDate(ByVal NumberOfDays As Double)
    MyDate.AddDays(NumberOfDays)
End Sub
```

```
        Public Sub AddToDate(ByVal NumberOfDays As Double, _
            ByVal NumberOfMonths As Integer)
            MyDate.AddDays(NumberOfDays)
            MyDate.AddMonths(NumberOfMonths)
        End Sub
```

You can also define a couple of special methods in your class that will automatically be called when the objects are first created and when they are being destroyed. Called *constructors* and *destructors*, these methods can be used to initialize variables when the class is being instantiated and to close system resources and files when the object is being terminated.

`Dispose` and `Finalize` are the two methods called during the destruction of the object, but the method called when a class is created is important enough to be discussed now. The `New` method is called whenever an object is instantiated. The standard `New` method syntax accepts no parameters and exists by default in a class until an explicitly defined `New` method is created; that is, the following two class definitions function in the same way:

```
  Public Class MyClass1
  End Class
  Public Class MyClass2
      Public Sub New()
      End Sub
  End Class
```

Both of the preceding class definitions enable you to define and instantiate an object with the `New` keyword:

```
  Dim MyObject As New MyClass1
```

However, if the explicitly defined `New` method accepts parameters, then you must instantiate the object with the required parameters or the program will not compile:

```
  Public Class MyClass2
      Private MyInteger As Integer
      Public Sub New(ByVal MyIntegerValue As Integer)
          MyInteger = MyIntegerValue
      End Sub
  End Class
  Dim MyObject As New MyClass2(3)
```

Bringing the capability of method overloading into the equation, however, enables you to define multiple versions of the `New` method in your class. The following definition would enable you to define objects and instantiate them without any parameter or with a single integer value:

```
  Public Class MyClass2
      Private MyInteger As Integer
      Public Sub New()
          MyInteger = 0
      End Sub
      Public Sub New(ByVal MyIntegerValue As Integer)
          MyInteger = MyIntegerValue
      End Sub
  End Class
```

One last important point: If your class is based on another class, to create the underlying class you must call the special method `MyBase.New` as the first thing in your `New` method definition so that everything is instantiated correctly.

The following Try It Out brings all of this information about how to create a class together by creating the `Person` class for the Personal Organizer application, complete with multiple `New` methods to demonstrate overloading.

Try It Out   Creating a Class

1. Start Visual Basic Express and create a new Windows Application project. Add a new class module to the project by selecting Project ⇨ Add Class. Name the class `Person.vb` and click Add to add it to the project.

2. In the class definition, start by defining private variables to store the `Person` information:

```
Private mFirstName As String
Private mLastName As String
Private mHomePhone As String
Private mCellPhone As String
Private mAddress As String
Private mBirthDate As Date
Private mEmailAddress As String
Private mFavorites As String
Private mGiftCategories As Integer
Private mNotes As String
```

3. For each of these variables, create a full property block with `Get` and `Set` clauses. For now, simply translate the property to the private variable. For example:

```
Public Property FirstName() As String
    Get
        Return mFirstName
    End Get
    Set(ByVal value As String)
        mFirstName = value
    End Set
End Property
```

4. Revise the code for setting the birth date so that it does not allow dates in the future. You can do this by comparing the date value passed in against the special date keyword `Now`, which returns the current date and time:

```
Public Property BirthDate() As Date
    Get
        Return mBirthDate
    End Get
    Set(ByVal value As Date)
        If value < Now Then
            mBirthDate = value
        End If
    End Set
End Property
```

5. Create a read-only property called `DisplayName` that concatenates the first names and last names:

```
Public ReadOnly Property DisplayName() As String
    Get
        Return mFirstName + " " + mLastName
    End Get
End Property
```

6. Create two `New` methods to enable the creation of a new `Person` class with or without the first and last names:

```
Public Sub New()
End Sub
Public Sub New(ByVal sFirstName As String, ByVal sLastName As String)
    mFirstName = sFirstName
    mLastName = sLastName
End Sub
```

7. Return to `Form1.vb` in Design view and add a button to the form. Double-click the button to create and edit the `Click` event handler and add the code to create a `Person` object and populate it:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    Dim pPerson As Person = New Person("Brett", "Stott")
    With pPerson
        .HomePhone = "(555) 9876 1234"
        .CellPhone = "(555) 1234 9876"
        .BirthDate = CType("1965-10-13", Date)
        .Address = "101 Somerset Avenue, North Ridge, VA"
    End With
    MessageBox.Show(pPerson.DisplayName)
End Sub
```

8. Run the application and click the button. After a moment you should be presented with a message dialog with the text `Brett Stott`. You've created your first class, complete with overloaded methods and read-only properties.

Control Freaks Are Cool

In a moment, you're going to take a look at how to interact with controls by changing their properties and intercepting their methods, so it's worth reviewing what can be done at design time to initialize the attributes of your controls even before you begin to run.

The Properties window (see Figure 6-2) enables you to customize the appearance of each element on the form, including the form itself. It also can be used to control behavior through attributes related to data and other nonvisible aspects of the control.
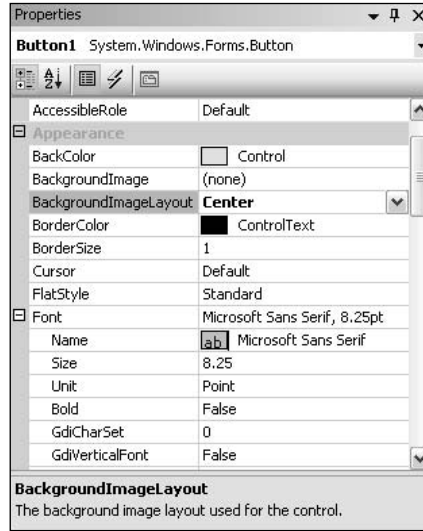
Figure 6-2

Design-time Properties

Rather than set the properties through simple text fields, the variety of methods to set the attributes is rich with drop-down lists, visual cues, and a hierarchy that groups related properties together. In Figure 6-2, the Appearance group for a Button control is shown. Out of all the properties in the visible area, only BorderSize and Size are simple text edits.

The remainder of the properties use a number of different editor types. For example, BackColor and BorderColor drop down three lists of colors to choose from and provide a sample of the color option right in the Properties window. The lists include system colors, which give you the capability to set your controls' color schemes to match the rest of the Windows system — if users change their system settings, your application can stay in synch with the rest of the environment (see Figure 6-3).
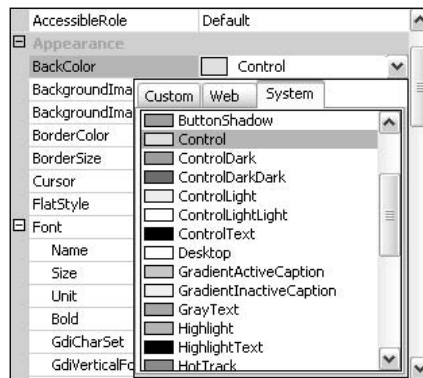
Figure 6-3

The Font property is actually an object and can be expanded as shown in Figure 6-3 to show the individual fields, such as Name and Size. While these properties can be set individually, the Font property can be used to show a general Font dialog window that enables you to set several of these attributes at once. In addition, the Font Name property offers a visual preview of the font option selected right in the Properties window so you can verify it's the correct choice.

The Properties window can be organized to show the properties in either alphabetical order or in categories, which is the default view. To switch between the two, click the Categorized and Alphabetic buttons at the top of the pane.

An interesting addition with Visual Basic Express is the capability to access the events that the selected control has. Click the Events button, which is a little yellow lightning strike icon, and the properties will be replaced by a list of events. Any event that has an event handler routine explicitly intercepting it will have the name of the routine listed here, and you can easily change the routines handling the different events by clicking the drop-down arrow and choosing them from the list. You're safe in that only the subroutines that have the correct signature will be listed.

If you're not sure what a particular property or event does, the Properties window will give you a brief description at the bottom of the pane. This tray area also serves another purpose for complex objects such as data-bound controls and visual components such as menus and toolbars.

Setting the Tab Order

Speaking of navigating through a form by pressing the Tab key raises a valuable point. By default, as you add controls to the design surface of a form or user control, they have a TabIndex value automatically assigned to them. This TabIndex controls the order in which the components are traversed when the user presses Tab.

In most applications, you'll find that this order is, well, orderly and logical, usually flowing from left to right and top to bottom, much like you would read this page. If you add your controls in an order that differs from this, or if you realize when running the application that it doesn't quite make sense for the navigation flow to work the way you've set it up, you'll need to change the TabIndex property.

These values can be set directly in the Properties window like any other property, or you can use the Tab Order Wizard, which makes setting them easy. To change to Tab Order mode, use the View ⇨ Tab Order command.

Tab Order view will place the current TabIndex properties over each control on the form (see Figure 6-4). To change the order, select each control in the order you want the navigation to occur by clicking them. As each control is selected, its TabIndex number will be set to the next available number (starting with 0), and the TabIndex marker will change color to indicate it has been set.

Once every element has been set, the TabIndex markers will reset to the original gray color to indicate you have done them all. At this point you can start again, or exit Tab Order mode through the View menu.

You may notice that this tab order is also followed at design time. This enables you to verify that the tab order is what you intended, and it gives you a logical way of proceeding through your controls as you edit their properties and events.
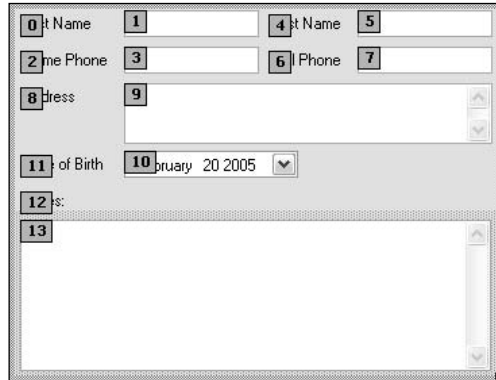
Figure 6-4

Editing Collections of Objects

If an object such as a `MenuStrip` is selected in Design view, among the properties you'll find objects that contain a collection of subordinate items. In the case of `MenuStrip`, the `Items` property identifies an array of `MenuItems` that belong to the control. Each `MenuItem` is a control in its own right and can be accessed by clicking it in the form's Design view, but a more natural way of editing the properties of the collection is through a Collection Editor.

It may appear that there are several of these Collection Editors, each specifically targeted at a particular type of object. In reality, there is one Collection Editor that Visual Basic Express customizes dynamically to suit the control you are editing.

The Collection Editor for the `Items` collection of a `MenuStrip` is shown in Figure 6-5. Each of the objects belonging to the `MenuStrip`'s `Items` collection is shown in the left-hand list, while the right-hand properties view provides direct access to its properties.

The beauty of this Collection Editor paradigm is that it is recursive. In Figure 6-5, the Edit menu item is selected and the properties list has been scrolled down to the `DropDownItems` property. This is another collection object that in turn can be edited through the Collection Editor (and if an item in that collection had a collection of sub-items, they could also be edited through this process, and so on).

Items within a collection can be repositioned or removed using the command buttons situated between the two lists. The Collection Editor is smart enough to know which types of items are valid for inclusion in the current list type. In the case of a `MenuStrip`, four kinds of items can be added to the collection — a standard menu item, a `ComboBox`, a `TextBox`, and a separator. To add the required item, choose its type from the drop-down list and click Add.

In the following Try It Out, you'll modify the menu and toolbar of the Personal Organizer application so that it contains only the items that you'll need. This will demonstrate the advanced aspects of the Properties window, including the Collections Editor and in-place property editing.
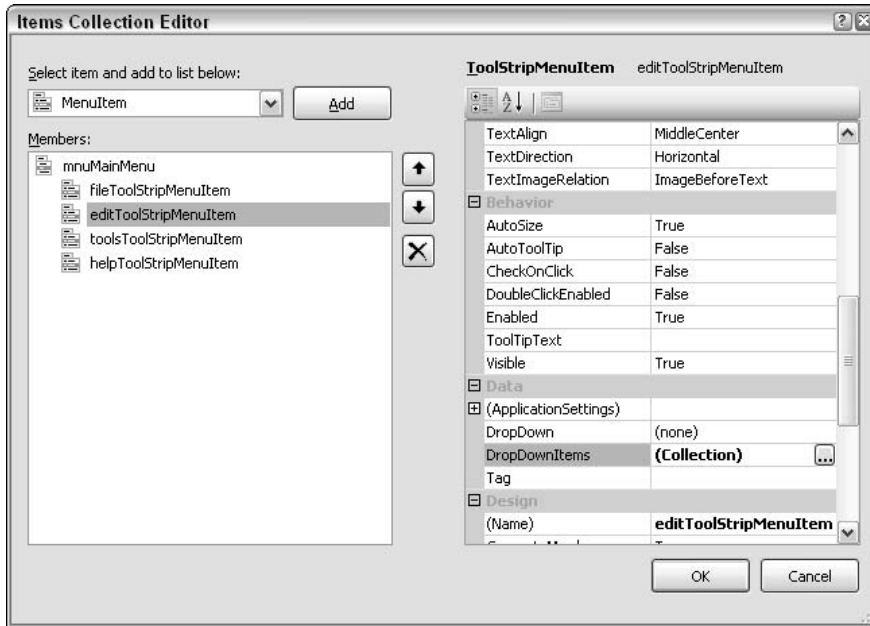
Figure 6-5

Try It Out   Modifying the Menu and Toolbar

1. Start Visual Basic Express and open the Personal Organizer solution you worked with in Chapter 5. If you don't have this project handy, you can find a copy of it in the `Chapter 06\PersonalOrganizer Start` folder of the downloaded code that you can get from the Wrox website (at `www.wrox.com`), complete with `MenuStrip` and `ToolStrip` with standard items.

2. Some of the items that were added through the Insert Standard Items command are unnecessary, and some commands you're going to need later. Therefore, you need to customize the menus and toolbars. First, change the toolbar so that it contains only what you need.

3. The `ToolStrip` has seven default items — New, Open, Save, Print, Cut, Copy, and Paste — with two separator lines dividing the buttons into logical groups. It also has a gripper so that it can be dragged around. Because it doesn't apply in this application with only the one `ToolStrip`, turn the gripper off by changing the `GripStyle` property to `Hidden`.

You can use all of these buttons except for Open, so go ahead and right-click the picture of the open folder and choose Delete to remove it from the `ToolStrip`. Do the same for the Help button as you won't implement help in this application.

4. Two useful commands that you'll build the code for later in this book are not present. They are shortcuts to delete the currently selected person from the list and to enable the user to log off. Click on the `ToolStrip` to make it active and show the in-place editor. In the Type Here area, enter `Delete` and press Enter to save the new button.

Repeat this action and add a new button with a text label of `Logoff`.

5. The buttons look a bit bulky at the moment and out of place because they are text buttons (the others are icons). Select `Delete` and then bring up the Smart Tag dialog window by clicking the small arrow on the right.

Change the `DisplayStyle` to `Image` and click on the ellipsis button next to the `Image` property to import a new image. In the Select Resource dialog, click the Import button to select an image file on your computer. If you don't have one handy, browse to the `Chapter 06\Images` folder included in the downloaded code for this book and select `delete.gif`.

When you've found the image you want to use, click Open to return to the Select Resource dialog, make sure it looks right in the Preview pane, and click OK.

6. Repeat the process in step 5 with the Logoff button. The image used in the example shown in Figure 6-6 can be found in the same folder of the downloaded code and is named `user.gif`.
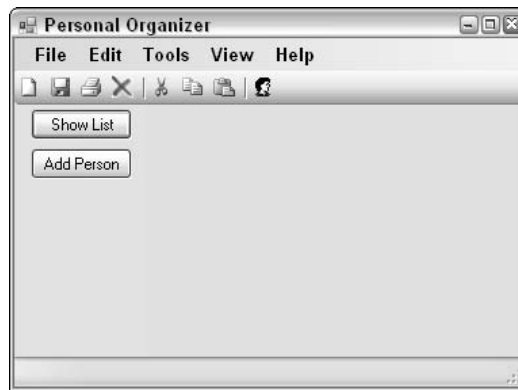
Figure 6-6

7. It would be nice to have the Delete command grouped with the New, Save, and Print buttons, so click and drag the icon to where you want it to be positioned and release it. You can rearrange the items on any toolstrip or menu like this.

8. Now it's the `MenuStrip`'s turn. Rather than use the in-place editor, this time you'll use the Collection Editor for the `Items` property. Select the `MenuStrip` and locate `Items` in the `Properties` window. Click the ellipsis button to bring up the Collection Editor.

9. Select the File menu's object—it has a name of `fileToolStripMenuItem`—and find its `DropDownItems` collection and again click the ellipsis button to dive down an extra level in the Collection Editor.

10. You won't need the Open or Save As commands, so select each one in the list and click the Delete button to remove them. The menu looks a bit awkward with a separator between New and Save now that they're by themselves, so remove that separator as well.

11. Select `ToolStripMenuItem` from the item drop-down and click Add. By default, it will be added to the bottom of the list, so click the Move Up button to move the new item above the Exit command. Change the new item's properties as follows:

❑ **Name** — logoffToolStripMenuItem

❑ **Text** — &Logoff

❑ **Image** — The same image you used for the Logoff button in step 6.

When you're finished, the list of items should look like Figure 6-7. To save all these changes, click OK to return to the main MenuStrip's Item Collection Editor.

```
ToolStripDropDownMenu
   newToolStripMenuItem
   saveToolStripMenuItem
   toolStripSeparator1
   printToolStripMenuItem
   printPreviewToolStripMenuItem
   toolStripSeparator2
   logoffToolStripMenuItem
   exitToolStripMenuItem
```
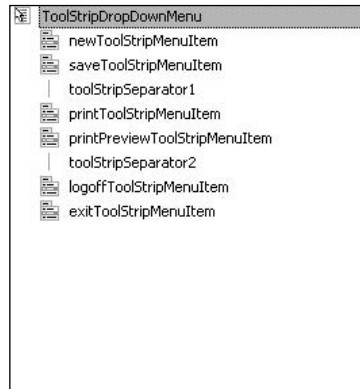
Figure 6-7

12. Leave all the menu items in place for the Edit menu, but add a new separator and an extra menu item at the bottom of the list. A ToolStripSeparator item will draw a line in between other commands to group them for your users. The extra ToolStripMenuItem should have the following properties set:

❑ **Name** — deletePersonToolStripMenuItem

❑ **Text** — &Delete Person

❑ **Image** — The same image you used for the Delete button in step 5.

13. Save the Edit menu by clicking OK, and then edit the Tools menu by selecting toolsToolStrip MenuItem in the list and clicking the ellipsis button next to its DropDownItems property. Remove all of the items that are currently there by selecting each one and clicking the Delete button. In their place, add two new menu items for Export Data and Import Data. Their settings, respectively, are as follows:

❑ **Name** — exportListToolStripMenuItem

❑ **Text** — &Export Data

and

❑ **Name** — importListToolStripMenuItem

❑ **Text** — &Import Data

14. Add a new View menu item with a name of `viewToolStripMenuItem`. Just like when you added the Logoff button to the Edit menu, adding the View menu will result in its being added to the end of the menu list, so move it "up" one position so it will appear before the Help menu.

15. In the View menu's `DropDownItems` collection, add two `ToolStripMenuItems` to give users easy access to the Person List and the Web Browser (this will be added in Chapter 9). Set the properties as follows:

❏ **Name** — `personListToolStripMenuItem`

❏ **Text** — `&Person List`

and

❏ **Name** — `webBrowserToolStripMenuItem`

❏ **Text** — `&Web Browser`

❏ **ShortcutKeys** — `Ctrl + W`

Again, save the changes you've made and return to the main menu list. The final menu is Help — remove all the items except for `aboutToolStripMenuItem`.

When you're done, click OK in the main `MenuStrip`'s Collection Editor to save all of the changes made. Run the application and compare it to Figure 6-8.
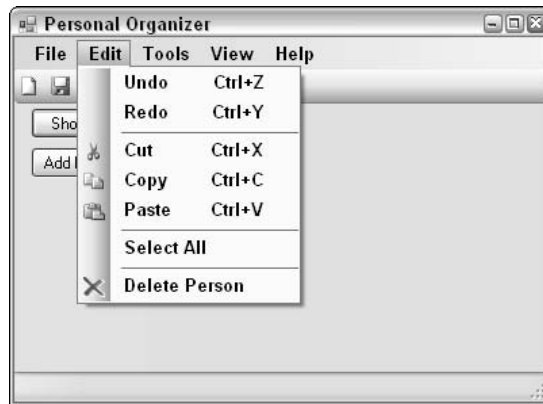
Figure 6-8

Custom Controls — Empower Yourself

When you create a user control, you're just defining a specialized class that includes visual components. That means all the information outlined in this chapter also applies to custom-built controls, including the creation of properties and methods, and the definition and invocation of events.

In Chapter 4, you created two basic user controls, which you then dynamically added to the main form area when the user clicks the appropriate button. However, you didn't add any properties, methods, or events to the user control's definition, so you may assume that you cannot access any information within the control.

That's actually not the case. If you take a look at the drop-down list that Visual Basic Express provides through IntelliSense for the controls, you'll find a list of properties and methods that are exposed by the user control itself, along with `Friend` scope variables for each of the components you added to the control.

Because they are defined with a scope of `Friend`, these controls can be accessed from the form that owns the user control, but using these properties is similar to creating public variables within a class — it allows the external code full access to the component — possibly more access than you would want.

On top of this, any events that are fired by the individual components within a user control are not passed on to the owner of the user control. The only events that the owner can access — assuming the control has been defined `WithEvents` so that the events can be intercepted — are those belonging to the user control itself.

Because of both of these reasons, it's best to explicitly define the members of the user control and in so doing regain control of what can and cannot be done to the internal elements of the control. Visual Basic Express keeps the code underneath the user control clean by keeping the objects and properties that you add to the design surface in a separate module.

By default, this module is hidden from view in the Solution Explorer, but you can access it by clicking the Show All Files button at the top of the Solution Explorer pane. The module will be named the same as the form or control with an extra extension of `Designer`, so the code behind a control named `MyControl` would be contained in a file named `MyControl.Designer.vb`.

What you'll find in this module is that Visual Basic Express uses the same class constructs you need to use when creating your own classes. Each control is defined using the `WithEvents` keyword so their events can be trapped, and there is a `New` method defined that initializes the properties of each component with the values you've set in Design view.

The result of this separation of visual design code and the underlying program logic is that when you modify the user control's code, you start with an empty class:

```
Public Class MyUserControl

End Class
```

All your own events, methods, and properties are defined within this class in exactly the same way as any other class you might create. However, because this class is connected to the hidden designer class, you have access to the components and their members. Each component will be accessible in the Class drop-down list at the top of the code editor, too, so you can easily find the events that you can intercept for each control.

To illustrate how you might define your own members for a user control, the following Try It Out walks through adding properties and methods to the `PersonalDetails` custom control that is part of the Personal Organizer application.

Try It Out   **Adding Properties to Persons**

1.   Return to the Personal Organizer project that you've been working on in this chapter. Add a new class module by selecting Project ⇨ Add Class and name it `Person.vb`. Follow the steps in the Try It Out entitled "Creating a Class" earlier in this chapter to define the basic `Person` class containing properties, the read-only property `DisplayName`, and the overloaded `New` methods.

2. Open the code view for `PersonalDetails.vb` by right-clicking its entry in the Solution Explorer and choosing View Code. Add a private module-level variable to store the `Person` class associated with the control and add a public property to allow other parts of your application to access it:

```
Private mPerson As Person

Public Property Person() As Person
    Get
        Return mPerson
    End Get
    Set(ByVal value As Person)
        mPerson = value
    End Set
End Property
```

3. Add code to the `Set` clause to automatically update the component controls on the user control whenever the `Person` property is updated:

```
Private mPerson As Person

Public Property Person() As Person
    Get
        Return mPerson
    End Get
    Set(ByVal value As Person)
        mPerson = value

        txtFirstName.Text = mPerson.FirstName
        txtLastName.Text = mPerson.LastName
        txtHomePhone.Text = mPerson.HomePhone
        txtCellPhone.Text = mPerson.CellPhone
        txtAddress.Text = mPerson.Address
        txtEmailAddress.Text = mPerson.EmailAddress
        txtFavorites.Text = mPerson.Favorites
        txtNotes.Text = mPerson.Notes
        dtpBirthdate.Value = mPerson.BirthDate

    End Set
End Property
```

4. When the user clicks the `New` button on the menu or toolbar or the Add Person button, it would be handy for the form to tell the control to revert to its default values in case it is currently being displayed. Add a subroutine called `ResetFields`, resetting all the controls and the `Person` object to empty values:

```
Public Sub ResetFields()
    txtFirstName.Text = vbNullString
    txtLastName.Text = vbNullString
    txtHomePhone.Text = vbNullString
    txtCellPhone.Text = vbNullString
    txtAddress.Text = vbNullString
    txtEmailAddress.Text = vbNullString
    txtFavorites.Text = vbNullString
    txtNotes.Text = vbNullString
```

```
            dtpBirthdate.Value = Now
            mPerson = New Person
        End Sub
```

5.   To test that these properties are accessible, you can change the code in the Add Person button on
`frmMainForm` to create a `Person` object, fill it with information, and then pass it over to the
`PersonalDetails` control. Ultimately, this will be the process you'll use in the Show Details
functionality when viewing the Person List, but you haven't connected the database yet, so this
serves as a test.

First, create a new `Person` object with the second `New` method definition to include the first
names and last names. Set some other properties of the object and then assign it to the `Person`
property exposed by the `PersonalDetails` control:

```
        Dim objPerson As New Person("Glenda", "Brown")
        With objPerson
            .BirthDate = "1965-12-02"
            .Address = "333 Green Valley Way, Los Angeles, CA"
            .HomePhone = "(811) 8888 7777"
        End With
        objPersonalDetails.Person = objPerson
```

To illustrate the use of the `ReadOnly` property, add an additional line to change the title text of
the form to include the `DisplayName` of the `PersonalDetails` control. The `Me` keyword is a
reserved word in Visual Basic Express identifying the form or control containing the code:

```
        Me.Text = "Personal Organizer - Viewing " + _
        objPersonalDetails.Person.DisplayName
```

The final `Add Person Click` event handler should look like this:

```
Private Sub btnAddPerson_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnAddPerson.Click
    objPersonalDetails = New PersonalDetails
```

```
        Dim objPerson As New Person("Glenda", "Brown")
        With objPerson
            .BirthDate = "1965-12-02"
            .Address = "333 Green Valley Way, Los Angeles, CA"
            .HomePhone = "(811) 888 7777"
        End With
        objPersonalDetails.Person = objPerson

        Me.Text = "Personal Organizer - Viewing " + _
        objPersonalDetails.Person.DisplayName
```

```
    If pnlMain.Controls.Contains(objPersonList) Then
        pnlMain.Controls.Remove(objPersonList)
        objPersonList = Nothing
    End If

    pnlMain.Controls.Add(objPersonalDetails)

    objPersonalDetails.Dock = DockStyle.Fill
End Sub
```

6. Double-click the New button on the toolbar and add code to its `Click` event handler to reset the fields if the `PersonalDetails` control is being displayed:

```
Private Sub newToolStripButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles newToolStripButton.Click
    If objPersonalDetails IsNot Nothing Then
        objPersonalDetails.ResetFields()
        Me.Text = "Personal Organizer"
    End If
End Sub
```

7. Run the application and click the Add Person button. The `Personal Details` control will be loaded, assigned the `Person` class, and then displayed in the main panel area of the form. Note that the properties that you set in the `Person` class were transferred over to their respective fields in the control, and that the title bar of the form has changed to include the `DisplayName` value (see Figure 6-9). Click the New button on the toolbar to reset the fields.
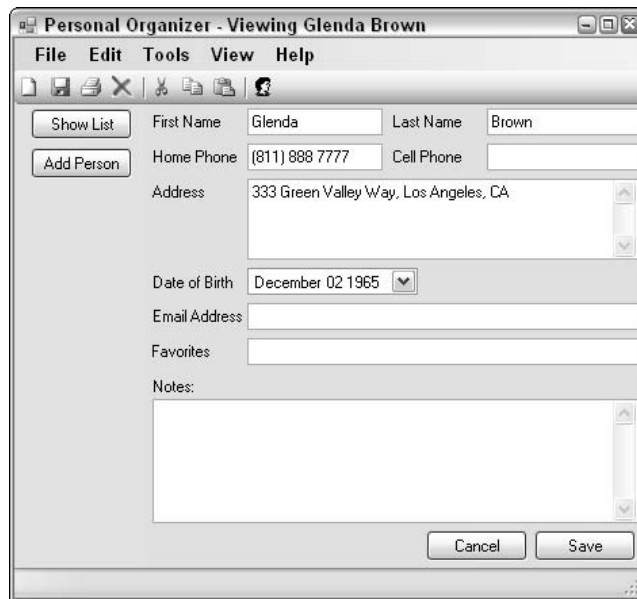
Figure 6-9

Go That Extra Mile

While controls can be created at design time, there's nothing stopping you from adding additional components while the program is running. In Chapter 5, this is effectively what you were doing when you created new instances of the `PersonList` and `PersonalDetails` controls and then added them to the Controls collection of `pnlMain`.

When you dynamically create a control in code, you must set its container context so that Visual Basic Express knows how to show it to the user. If you do not add it to another component that can contain it, it won't be visible (this may be what you want if you're keeping track of a control's properties). The

normal process of adding a control is to first define it as a module-level variable, including the `WithEvents` clause if your program needs to handle any events it may raise. Then you create a new instance of the particular control you're after and add it to the container control. If the control is to be placed directly on the form or user control, you can use the `Me.Controls` collection. Set the properties of the control — for example, a button might require Name, Text, and positional properties such as Left and Top as a minimum. If the control has events that are to be intercepted, you can use the Visual Basic Express code editor to automatically create the event handler routine and hook it to the `Click` event.

Visual Basic Express enables you to go an extra step in dynamically creating controls. Rather than define the control as a module-level variable and then instantiate it when you need it during the execution of the application, you can wait until the control is required.

The process of creating the control, adding it to its container, setting the properties, and so on is the same as previously discussed, but you'll find that you cannot connect the event handler routines to the control's events. This is because the control isn't defined at the module level but is created on the fly in a local subroutine or function and then added to an existing part of the form or user control.

Fortunately, Visual Basic Express enables you to connect event handler routines dynamically with the `AddHandler` statement. `AddHandler` specifies that a particular object's event should be hooked to a subroutine. The subroutine must have the same signature as the object's event. Rather than connect it up directly, because this is done while the program is running (and therefore the program is already compiled), you need to specify that the event should be hooked to the address of the function. Thus, the syntax of the `AddHandler` statement is as follows:

```
AddHandler ObjectName.EventName, AddressOf EventHandlerName
```

When the program runs, it creates the object and then connects the specified event to the location in the compiled code where the event handler routine resides. As a result, whenever the event is fired, Visual Basic knows what subroutine should be executed.

You may have noticed that the `PersonalDetails` user control in the Personal Organizer application doesn't have any Save or Cancel buttons in its design even though it is currently accessed through the Add Person button. This is because the control is to be used for multiple purposes. In the next Try It Out, you'll dynamically create Save and Cancel buttons, position them at the bottom of the control, and resize the Notes area to accommodate the new components. Finally, you'll use the `AddHandler` method to intercept the `Click` events of the buttons and pass the events on to the owner of the user control with your own events.

Try It Out   Creating Dynamic Buttons

1. Return to the Personal Organizer project you've been working with and open the `PersonalDetails` user control in code view. The first step is to create a new variable to keep track of the view state in which the `PersonalDetails` control is meant to be shown:

```
Private mAddMode As Boolean
Public Property AddMode() As Boolean
    Get
        Return mAddMode
    End Get
    Set(ByVal value As Boolean)
        mAddMode = value
    End Set
End Property
```

2. Add code to set up the buttons if the control is in Add mode and to remove them if not:

```
Public Property AddMode() As Boolean
    Get
        Return mAddMode
    End Get
    Set(ByVal value As Boolean)
        mAddMode = value
        If mAddMode = True Then
            SetUpButtons()
        Else
            RemoveButtons()
        End If
    End Set
End Property
```

3. Each button is created in the `SetupButtons` subroutine and has the basic properties of `Name` and `Text` set. They are both anchored at the bottom right so they move as the control is resized, and then their `Top` and `Left` properties are calculated so they are positioned at a reasonable distance from the edge of the control. Note how the Cancel button's `Left` property needs to take the Save button into account for its position. In addition, the Cancel button's `Top` property does not need to be calculated — it can use the same value as the Save button.

Because these buttons are taking up space that was previously consumed by the Notes text box, you'll need to resize `txtNotes` so that it finishes above the Save and Cancel button positions.

Finally, use `AddHandler` to connect the Click events of the two buttons to an event handler you'll create in a moment. The final `SetupButtons` routine looks like this:

```
Private Sub SetupButtons()
    Dim mSaveButton As New Button
    Me.Controls.Add(mSaveButton)

    With mSaveButton
        .Name = "btnSave"
        .Text = "Save"
        .Anchor = AnchorStyles.Bottom + AnchorStyles.Right
        .Top = Me.Height - (.Height + 5)
        .Left = Me.Width - (.Width + 5)
    End With

    Dim mCancelButton As New Button
    Me.Controls.Add(mCancelButton)

    With mCancelButton
        .Name = "btnCancel"
        .Text = "Cancel"
        .Anchor = AnchorStyles.Bottom + AnchorStyles.Right
        .Top = mSaveButton.Top
        .Left = mSaveButton.Left - (.Width + 5)
    End With

    With txtNotes
        .Height = mSaveButton.Top - (.Top + 5)
    End With
```

117

```
        AddHandler mSaveButton.Click, AddressOf ButtonClickedHandler
        AddHandler mCancelButton.Click, AddressOf ButtonClickedHandler

    End Sub
```

4. The `RemoveButtons` function needs to cycle through the `Controls` collection of the user control looking for each of the buttons. When it finds it (which it can do by checking the `Name` property of each member of the collection), it then uses the `Remove` method to delete it from the user control.

To finish the job, you'll need to reset the height of the Notes text box so that the space previously taken by the Save and Cancel buttons is used up again:

```
Public Sub RemoveButtons()
    With Me.Controls
        For iCounter As Integer = 0 To .Count - 1
            If .Item(iCounter).Name = "btnSave" Then
                .Remove(.Item(iCounter))
                Exit For
            End If
        Next
        For iCounter As Integer = 0 To .Count - 1
            If .Item(iCounter).Name = "btnCancel" Then
                .Remove(.Item(iCounter))
                Exit For
            End If
        Next
    End With
    With txtNotes
        .Height = Me.Height - (.Top + 5)
    End With
End Sub
```

5. The last piece of code you'll need to create is the `ButtonClickedHandler`. This routine is used for both `Click` events, so you'll need to determine which button was clicked. One of the parameters of control events is a sender object. By default, it is defined in the parameter list as a `System.Object` and as such you cannot access the properties you need.

6. To get at the button properties, you first convert `sender` to a `Button` object using `CType`. Once you've got the object converted to a button, you can interrogate the `Name` property to determine which button was clicked:

```
Private Sub ButtonClickedHandler(ByVal sender As System.Object, _
    ByVal e As System.EventArgs)

    Dim btnSender As Button = CType(sender, Button)
    If btnSender.Name = "btnSave" Then
        MessageBox.Show("Save was clicked")
    ElseIf btnSender.Name = "btnCancel" Then
        MessageBox.Show("Cancel was clicked")
    End If
End Sub
```

7. Edit the Add Person Click event handler in the main form code to set the `AddMode` property to `True` and run the application. Now whenever the Add Person button is clicked, the `PersonalDetails` control will dynamically change to include Save and Cancel buttons, with an event handler connected to display dialog windows when they themselves are clicked.

Summary

Visual Basic Express provides countless ways of creating and organizing your application. Creating classes and custom controls is one of the best ways of segregating the information and activities in a larger program.

In this chapter, you learned to do the following:

❑ Create custom classes and controls to organize your application.

❑ Pass information between sections of your program through events and properties.

❑ Customize controls dynamically while the program is running.

In the next chapter, you'll learn about the data controls and how to use them to connect databases to user interface components with minimal coding.

Exercises

1. Create an event handler for the New Person menu item that replicates the code you created for the New button on the `ToolStrip`.

2. Create an event in the `PersonalDetails` control that you can raise when the Save and Cancel buttons are clicked.

7

Who Do You Call?

In the first section of this book you learned the fundamentals necessary to start creating applications with Visual Basic Express. With those skills, you can design well-constructed user interfaces, write Visual Basic Express code, and use many of the aids and helper utilities that Visual Basic Express provides to make your experience more enjoyable and much easier. Chapter 6 enhanced those skills by discussing runtime customization of controls and controlling information as events occur.

This chapter continues your journey into the world of programming by extending your knowledge of writing code. Defining, creating, and using functions to access the database and using code to access database files to populate user interface elements are both discussed, and you will learn some more powerful components for the user interface design itself that automatically bind to a data source.

In this chapter, you learn about the following:

❏   Using the `Data` controls to access databases within your program

❏   Methods that are used to find individual rows and fields of information within a database

❏   Controls that can be data-bound to a data source so you don't have to write your own code

Using the Database Connection

By the end of Chapter 3 you were able to create a database using the Database Explorer and then add it to your project. Once you have that connection, you're able to access the information in the database in all sorts of ways, the easiest being to use the Visual Basic Express Integrated Development Environment (IDE) to automatically do most of the work for you.

Visual Basic Express comes with a number of controls that are used exclusively for database access. Some of these controls are actually invisible components that enable other parts of your program to get to the database, such as the `BindingSource` and `DataSet` controls, but the `DataGridView` is a powerhouse when it comes to accessing data.

The `DataGridView` is a table-like control, with an appearance similar to the data view you get in the Database Explorer. Each row of information in the database table is represented by a row in the `DataGridView`, with each field displayed in a separate column (see Figure 7-1).

Figure 7-1

The default properties of the `DataGridView` enable users to view the information as well as update it by adding new rows, deleting existing ones, or updating the information they can see. You might not want to allow some of these actions, however, so fortunately you have properties that can be used to control the user's level of access to the information.

Adding a `DataGridView` to your project is done in the same way as any other visual control. Locate it in the Toolbox — it is in the Data category — and either double-click its entry or click and drag it to the desired location. Either way, Visual Basic Express adds the control to the form and presents you with the DataGridView Tasks dialog so you can select some of the more common property settings (see Figure 7-2).
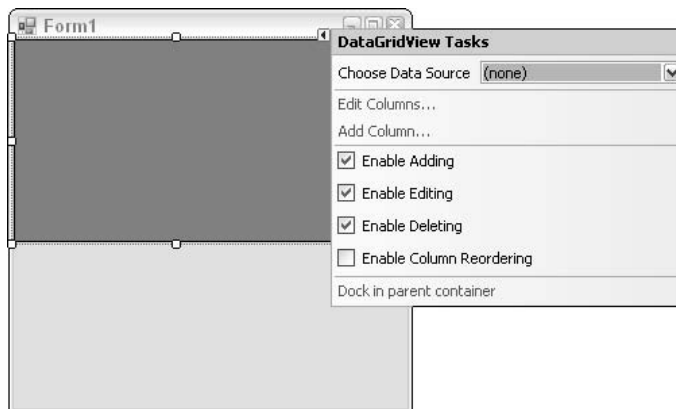
Figure 7-2

The obvious first setting you need to update is the Data Source. You could leave this set to (none) and assign a data source in code. If you follow that method, note that as long as the data source is the correct object type, the `DataGridView` will accept it. This means you can use the `DataGridView` even for information that is not stored in a database — just create a temporary data source object and populate it with the information you want to show and then assign it to the `DataGridView` control.

Clicking the drop-down button in the Choose Data Source property will display a list of all data sources currently available for your use (see Figure 7-3). If a data source is already defined in the form, it will be shown first, and then you will have a list of other places to get the information, including databases added to the project but not referenced in the current form. Select the table of information you want to connect to, and Visual Basic Express does the rest.
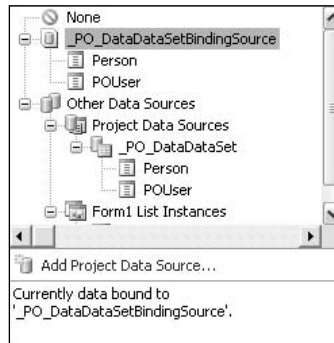
Figure 7-3

Behind the scenes, Visual Basic Express adds a `DataSet` object along with the other components it needs to connect the database to the `DataSet`, and then the `DataSet` to the `DataGridView`. This includes a `DataAdapter`, which you'll see later in this chapter, and a `BindingSource` component that automates the binding of data to visual components.

This Tasks window is also where you can disable the database updating functionality — each of the three update types is represented individually for maximum flexibility. Imagine a scenario in which you wanted the user to be able to add a new record of information but not be able to edit any of the existing information. Alternatively, you might want users to be able to edit only existing rows but not add new information or delete records. These scenarios are possible by selecting the appropriate combination of checkboxes.

The Edit Columns command gives you the capability to change the settings about each individual field in the database. You can remove columns completely, reorder them, and change their visual cues in the Bound Column properties window (see Figure 7-4). For example, you might not want to display the key field that is automatically maintained by the database. You can either remove it completely or set its `Visible` property to `False`.

Another field might be better served by using a `ComboBox` instead of a normal text edit so the user selects only valid values. The `ColumnType` property can be used to set this. One common task is to change the heading captions to be more user friendly — you don't want your users seeing `MyDataUserBankDetailsAccountNumber` when it would be much easier for them to see `Account No.`.

Once you have chosen the fields you want to display and have configured the various properties regarding each, you're done. You could change additional options about the `DataGridView`, but with only these few actions you've done enough to prepare your program to use the `DataGridView` with your own data source.
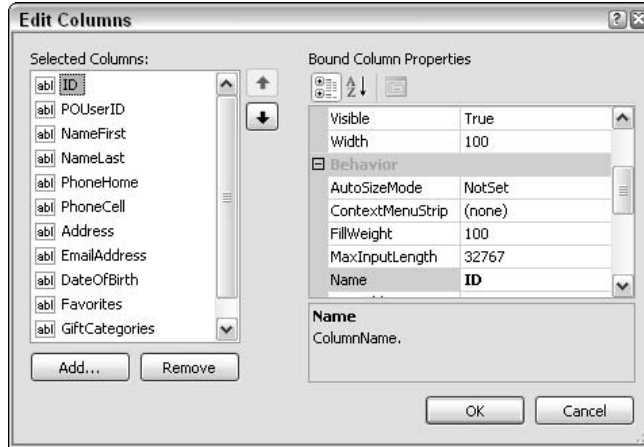
Figure 7-4

An Alternate Method

While adding a `DataGridView` to your application really is that straightforward, you might want to give users some additional visual cues to let them know what actions they can perform on the data. Visual Basic Express comes to the rescue by automating the process.

To add a `DataGridView` with associated navigation controls, open the Data Sources tab in the IDE (located in the same space as the Solution Explorer). Navigate through the list of data sources you can use and drag the table you want to display onto the form. A `BindingNavigator` control is added to the form in addition to the `DataGridView` itself. The `BindingNavigator` is automatically docked to the top of the form and contains buttons for navigating through the rows presented in the `DataGridView`, as well as buttons for adding new rows, deleting the selected row, and saving the changes to the database (see Figure 7-5).

At first glance, you might think that the navigation controls are overkill. Navigating through the `DataGridView` is just as easy using the cursor keys, adding a new row to the database is as simple as typing in the new row area of the control, and deleting is similarly easy. But the `BindingNavigator` opens the door to different kinds of data representation. Rather than use a `DataGridView` that presents all the information contained in the table in a gridlike fashion, you can present a single row of data at a time to the user. As users click the navigation buttons, they can show different rows of information from the database without you having to write a single line of code.

Figure 7-5

If you would rather present the data in this way, click on the drop-down button next to the table's name in the Data Sources window and select Details. This tells Visual Basic Express that when you add the table to the form, it should use individual fields rather than the `DataGridView`. It uses the same `BindingNavigator` control to allow users to access the different actions, but enables you to customize the appearance to suit your data more appropriately.

By default, Visual Basic Express tries to guess the best control for each data field — so date fields, for example, will be represented by a `DateTimePicker` control — and each editor control is paired up with a `Label` control to describe it so that it is presented to the user with as much supporting information as Visual Basic Express can determine automatically (see Figure 7-6).

Figure 7-6

You can add the data to the form in an even more detailed fashion by selecting individual fields in the Data Sources window and dragging them to the form design. This enables you to select different editor types for specific fields. Click the drop-down button next to the field's name and select the type of editor that should be used. When you're happy with the type, just drag the field to the form and let Visual Basic Express do the rest. If you already have the connection to the database set up from a previous field or table being added to the form, Visual Basic Express uses the same connection for the new fields.

What about Existing Controls?

Of course, there's also the situation in which you have defined your user interface and now want to connect it to a database so it is automatically populated. The creators of Visual Basic Express didn't overlook this common scenario, and you'll be pleasantly surprised with how easy it is to do.

Open your form's design and locate the control you want to connect to the database. Then, go to the Data Sources window and select the database field that should be used to populate the control. Click and drag it over to the form. When you position the mouse cursor over a valid control, the cursor changes to indicate that it can be bound. Release the mouse button. Visual Basic Express kicks into gear by connecting the field to the control. Again, if you have no existing data components on the form, it will do all that for you, too.

The great thing about this method is that you have total control over your form design, and you can use almost any control you want to represent information from your database, including `ListBox`, `ComboBox`, and `Labels`.

125

In the next Try It Out, you'll see this capability in action when you connect the database to the Personal Organizer application you've been building, and bind the `ListBox` control in the `PersonList` user control to the `Person` table.

Adding a Database to Personal Organizer

1.   Open the Personal Organizer project you've been building in previous chapters. Up to this point, you have defined the database in one project. In addition, you also created a user interface including several user controls that display a list of `Person` records and give the end user the capability to add a person. (Well, it's a simulated add process because you haven't actually done the database code yet — that's what the Try It Outs in this chapter will achieve.) Now it's time to combine the two.

If you haven't followed along up to this point, the `Chapter 07\Personal Organizer Start` folder in the downloaded code from `www.wrox.com` contains a starting point for this Try It Out.

2.   Add the database file you created in Chapter 3 by selecting Data ⇨ Add Data Source. The Data Source Configuration Wizard is displayed. Select Database and click Next to continue.

3.   Click the New Connection button and browse to the database file. When you find it, click the OK button to return to the wizard and click Next to proceed to the next step of the wizard.

If you're prompted to add the database to your project, you can choose to do so at this point. If you add it to the project, then each time you build the application, Visual Basic Express copies the database file to the build directory from the original database file. This means each time you run it, you get a clean set of data.

If you answer No to this prompt, it leaves the database file where it is and the `Connection String` value points directly to the file. This option is good if you want to continue to work on the same set of data between builds of the application.

4.   Save the `Connection String` as `PO_DataConnectionString` and click Next to display the database information. Select both the `Person` table and the `POUser` table and set the `DataSet` name to `_PO_DataDataSet` (this should be the default if you've been following along through this book) and click the Finish button to add the data source to your project.

5.   Open the `PersonList` user control in Design view. When it is displayed, click on the Data Sources tab (it shares space with the Solution Explorer) to bring it to the fore. Click and drag the `Person` table over to the Design view of `PersonList`. When the mouse cursor is positioned over the `lstPersons ListBox` control, release the mouse button.

Visual Basic Express automatically changes the `ListBox` properties so that it is bound to the `Person` table. By default, it sets the `Value Member` property to `ID` because it detects that as the unique identifier for the `Person` table. It also sets the `DisplayMember` property to the first field that is not a key of some kind. In this case it's the `NameFirst` column (see Figure 7-7).

6.   Run the application and click the Show List button to create a new instance of the `PersonList` control. Visual Basic Express automatically populates the underlying `Data` controls with the information in the database and then populates the content of the `ListBox` from those controls.
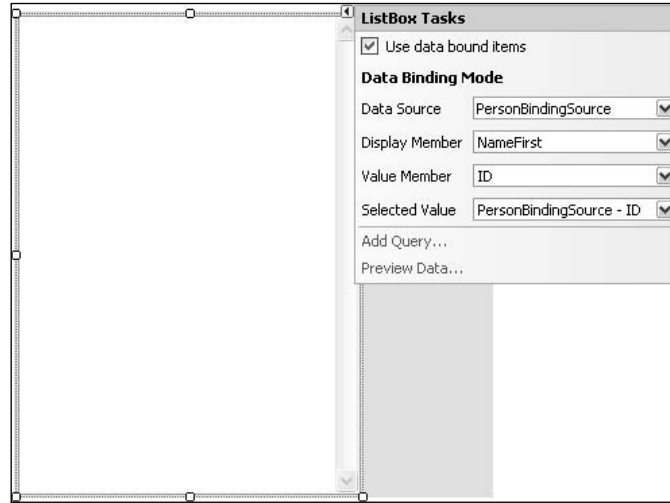
Figure 7-7

Database Programming

While adding data-bound controls is really that easy, usually you'll want a bit more control over what is displayed. The data-bound elements can be customized a little further than this click-and-drag methodology, but it's far more powerful creating your database access in code.

You will use three main objects in all your database programming — `DataTable`, `DataAdapter`, and `DataView`. With these objects, you can do just about anything with your database. Each serves a different purpose:

❑ **DataTable** — The main object containing the information about a database table and all of the rows contained within it. The `DataTable` is used to store the actual data and doesn't connect directly to a database itself, relying instead on other objects to populate and extract its information.

❑ **DataAdapter** — This is the component used to connect to the database. It contains the connection settings necessary to find the specific database file and has methods to retrieve information from the database and to post data back.

❑ **DataView** — This object is used as a "do it all" component for accessing the data within a `DataTable` object. While a lot of the actions that a `DataView` exposes can be performed directly from the `DataTable` class, it's better to separate the actions from the data to help identify what you're doing (in terms of functionality).

To connect to the database using these objects, you first create a connection to the database. This can be done by adding the connection through the Data Sources window or selecting the Data ➪ Add New Data Source command.

This adds a new entry to the Solution Explorer representing the database and customized versions of the `DataAdapter` and `DataTable` objects for use in your code (the same ones Visual Basic Express uses for the automated user interface components discussed earlier in this chapter). As an example, consider the case of adding a database called `MyDB.mdf` that contains a table called `MyTable`. An entry of `MyDB.mdf` with a database icon would be added to the Solution Explorer.

More importantly, however, when writing code, you could create an instance of the `DataAdapter` class that was generated by Visual Basic Express, like so:

```
Dim MyAdapter As New MyDBDataAdapters.MyTableAdapter
```

Similarly, creating a `DataTable` object for the table can be achieved using another customized object:

```
Dim MyTable As New MyDBDataSet.MyTableTable
```

Note that you do not have to add the database to your project at all. Instead, you can use the default `DataAdapter` and `DataTable` classes — for SQL they are called `Data.SqlClient.SqlDataAdapter` and `Data.SqlClient.SqlDataTable`, respectively — but you need to establish the connection in code as well.

To connect to a SQL database, you would use the `Data.SqlClient.SqlClient` class and specify the full connection string, including the database type, the physical location of the data, and any additional parameters needed to log on, such as user name and password.

The problem with using these generic objects is that you don't get all the customized properties and methods to manipulate your data, and you have to access individual fields through a general `Items` collection. It is far easier to let Visual Basic generate the appropriate objects, as you'll see in the Try It Out at the end of this discussion.

Once you have created the `DataTable` object, it starts out empty. To populate it with the data from the database, you call the `Fill` method of the `DataAdapter`, passing in the `DataTable` as a parameter:

```
MyAdapter.Fill(MyTable)
```

If you perform updates on the data in the table and need to send them back to the database, you must use the `DataAdapter` to pass the new data back to the database via the `Update` method:

```
MyAdapter.Update(MyTable)
```

Actions You Can Perform

Once you have the data in your `DataTable`, you can perform four types of action — select, insert, update, and delete. These should be fairly self-explanatory but here's a brief summary:

❑   **Select** — This is an informational action only. It is the mainstay of the `DataTable` and enables you to filter out only the rows of information you need. The default action for `Select` is to return all of the information in the `DataTable` in an array of `DataRow` objects, one for each record in the database, but you can include parameters to select only records that have fields that match certain criteria, to sort the data in different ways, and even to select records that are

in a particular database state, such as newly added, deleted (records that are deleted are marked for deletion but are not actually removed from the database until you call the `Update` method and pass the information back to the database), and modified.

❑ **Insert** — To insert an additional row into the database, you would normally call the `Insert` command in SQL. The Visual Basic Express way is to call `AddRow` and pass the new row information into the newly added row. It goes one step better than that, however, when you connect to the database as recommended in this chapter. Keeping with the previous example, to add a new row to the `MyTable` table, you'll find two versions of `AddMyTableRow` available. One accepts a `MyTableRow` object and the other enables you to specify the information about each field right there in the function call without having to create a temporary object.

❑ **Update** — Editing existing information is done with two commands — `BeginEdit` and `EndEdit`. These methods are available on the individual `DataRow` object and signify the beginning and end of the edit process, respectively. Once you have called `BeginEdit`, you can update the contents of the fields in the same way you would any property in a normal class — for example, `MyTable.FirstName = "NewName"`. If you begin the update but need to cancel it, use the `CancelEdit` method to discard the changes that have been made.

❑ **Delete** — Removing rows from the table is performed on the row itself. First find the row you need to remove and then call the `Delete` method, like so: `MyRow.Delete`.

Remember that updates to database tables are not saved to the database until you call the `Update` method in the `DataAdapter`. If you don't do this, all of the changes you've made in the `DataTable` will be lost.

The following Try It Out walks through the creation of three database-related functions to access and update `Person` details in the Personal Organizer application you've been building throughout the book. The code in the application is then modified to call these functions when needed. In addition, the `PersonList` control is updated to automatically populate the `ListBox` control and delete `Person` rows from the database table to show how database access code can be written anywhere in your application.

Try It Out   Accessing the Database through Code

1. Return to the Personal Organizer project you've been working on in this chapter. Because you're going to do everything in code, open the `PersonList.vb` user control in Design view and select the `lstPersons` component. Locate the `DataSource` property and change it to `None` so that it no longer uses the `BindingSource` object to retrieve information from the database.

2. To allow the `ListBox` to be populated from different places in the code, you'll create a single subroutine that can be called by different functions. The subroutine should define a `PersonTableAdapter` to connect to the database and a `PersonDataTable` to process the data stored in the database:

```
Private Sub LoadListBox()
    Dim PersonListAdapter As New _PO_DataDataSetTableAdapters.PersonTableAdapter
    Dim PersonListTable As New _PO_DataDataSet.PersonDataTable
End Sub
```

This is the fundamental system outlined earlier — create a `DataAdapter` to connect the program to the database using a connection string. Then use a `DataTable` object to store the information. This is populated via the `DataAdapter`'s `Fill` method:

```
Private Sub LoadListBox()
    Dim PersonListAdapter As New _PO_DataDataSetTableAdapters.PersonTableAdapter
    Dim PersonListTable As New _PO_DataDataSet.PersonDataTable
    PersonListAdapter.Fill(PersonListTable)
End Sub
```

Note that because you added the tables to the project via the Add Data Source command, you can use special classes that expose properties specific to the particular table to which they belong. Instead of declaring `PersonListAdapter` as a `DataAdapter`, you can reference the `PersonTableAdapter` instead and access the fields that belong to the table directly.

3. Clear the `Items` collection of the `ListBox` to prepare it for the database information:

```
With lstPersons
    .Items.Clear()
    ... database population code goes here
End With
```

4. The `ListBox` control enables you to add almost any kind of object to its `Items` collection as long as you also specify an accessible property that can be used for the `DisplayMember` property. This means you can use the `Person` class you created in previous chapters. However, the `Person` class doesn't contain a fundamental property that uniquely identifies a corresponding `Person` row in the database. To remedy this, edit the `Person.vb` class, adding a module-level variable and a public property for the ID field:

```
Private mID As Integer
Public Property ID() As Integer
    Get
        Return mID
    End Get
    Set(ByVal value As Integer)
        mID = value
    End Set
End Property
```

5. Return to the `LoadListBox` subroutine and loop through the `PersonRow` objects that were returned by the Fill command in the `PersonListTable`. For each row found, create a new `Person` class containing the information you need—first names and last names as well as the ID—and add it to the `Items` collection:

```
With lstPersons
    .Items.Clear()
    .DisplayMember = "DisplayName"
    For Each CurrentRow As _PO_DataDataSet.PersonRow In PersonListTable.Rows
        Dim CurrentPerson As New Person(CurrentRow.NameFirst, CurrentRow.NameLast)
        CurrentPerson.ID = CurrentRow.ID
        .Items.Add(CurrentPerson)
    Next
End With
```

6. Add an event handler routine for the user control's `Load` event and call the `LoadListBox` sub-routine you just created:

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

7. Run the application and click the Show List button to display the `PersonList` control. This time, the `ListBox` is being populated through code, and you can customize its appearance by using the `DisplayName` property of your `Person` class to show information that might be a bit more meaningful to the user (see Figure 7-8).
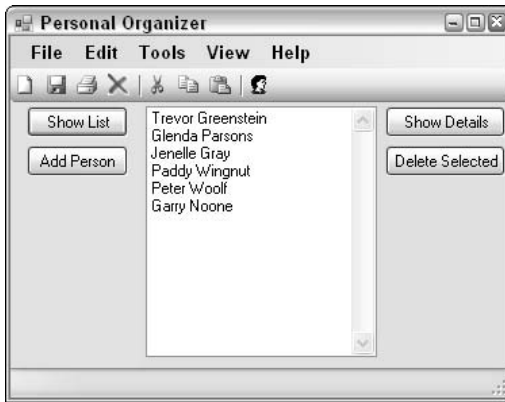
Figure 7-8

8. You'll create code for the two buttons on the `PersonList` form next. The Delete Selected button needs to ensure that at least one row is selected in the `ListBox`. Add an event handler routine for its `Click` event:

```
Private Sub btnDeleteSelected_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnDeleteSelected.Click
    If lstPersons.SelectedItems.Count > 0 Then
        ... code to delete selected people goes here
    Else
        MessageBox.Show("You have not selected any people to remove")
    End If
End Sub
```

9. Just as you did in step 2, first define a `PersonTableAdapter` and a `PersonDataTable` and populate the contents of the `PersonDataTable` object:

```
Dim DeletePersonAdapter As New _PO_DataDataSetTableAdapters.PersonTableAdapter
Dim DeletePersonTable As New _PO_DataDataSet.PersonDataTable
DeletePersonAdapter.Fill(DeletePersonTable)
```

10.   You now have two collections of data — the contents of the `SelectedItems` property of the `ListBox` and the `PersonRow` collection from the database table. Create a loop to iterate through the database rows. For each one, check its ID value against the ID values in the `SelectedItems` collection. If there is a match, remove the record using the `Delete` command:

```
For Each CurrentPersonRow As _PO_DataDataSet.PersonRow In DeletePersonTable.Rows
    For Each objPerson As Person In lstPersons.SelectedItems
        If CurrentPersonRow.ID = objPerson.ID Then
            CurrentPersonRow.Delete()
            Exit For
        End If
    Next
Next
```

11.   While this process has deleted the rows in the `DataTable`, you still need to transfer those changes back to the database itself. Use the `Update` method of the `DataAdapter`, passing in the table that contains the deleted rows. When you're done, you can call the `LoadListBox` method to reload the contents of the `ListBox`:

```
DeletePersonAdapter.Update(DeletePersonTable)
LoadListBox()
```

12.   Run the application and show the Person List again. This time, select a couple of person entries and then click the Delete Selected button. The code will first delete them from the database and then repopulate the list without the selected people.

13.   The other button in the `PersonList` control is Show Details. This is intended to swap the view over to the individual person details by showing the `PersonalDetails` control and then populating it with the `Person` information from the database. To do this, you need a new function that retrieves a specific `Person` row from the database.

14.   Add a new module to the project via the Project ⇨ New Module menu command and name it `GeneralFunctions.vb`. Visual Basic Express enables you to short-cut object definitions through the use of the `Imports` statement. You place this statement at the top of the module file and specify a namespace that you're going to use. Because this module ultimately contains many database-related functions, it would be nice to be able to refer to the object types without having to continually type the whole path, so add an `Imports` statement for the `System.Data` namespace:

```
Imports System.Data
Module GeneralFunctions
End Module
```

15.   Create a new function called `GetPerson` that accepts an `Integer` parameter containing the ID of the `Person` row to retrieve and returns a `Person` object. Add a `PersonTableAdapter` and a `PersonDataTable` and populate the table with the adapter's `Fill` method (starting to see a pattern?):

```
Imports System.Data
Module GeneralFunctions
    Public Function GetPerson(ByVal PersonID As Integer) As Person
        Dim GetPersonAdapter As New _PO_DataDataSetTableAdapters.PersonTableAdapter
        Dim GetPersonTable As New _PO_DataDataSet.PersonDataTable
        GetPersonAdapter.Fill(GetPersonTable)
    End Function
End Module
```

16. To demonstrate how strongly typed datasets can work in conjunction with the more generic `Data` objects, you now create a `DataView` that filters the `GetPersonTable` object so that it contains only the row that matches the ID:

```
Public Function GetPerson(ByVal PersonID As Integer) As Person
    Dim GetPersonAdapter As New _PO_DataDataSetTableAdapters.PersonTableAdapter
    Dim GetPersonTable As New _PO_DataDataSet.PersonDataTable
    GetPersonAdapter.Fill(GetPersonTable)
    Dim PersonDataView As DataView = GetPersonTable.DefaultView
    PersonDataView.RowFilter = "ID = " + PersonID.ToString
End Function
```

You can now check the `DataView` to determine whether there are any matches. If there are more than zero, then you know that there is only one (because the ID field is unique). Create a new `Person` class and populate the properties with the corresponding fields from the database. You should also return `Nothing` if a matching record was not found:

```
Public Function GetPerson(ByVal PersonID As Integer) As Person
    Dim GetPersonAdapter As New _PO_DataDataSetTableAdapters.PersonTableAdapter
    Dim GetPersonTable As New _PO_DataDataSet.PersonDataTable
    GetPersonAdapter.Fill(GetPersonTable)
    Dim PersonDataView As DataView = GetPersonTable.DefaultView
    PersonDataView.RowFilter = "ID = " + PersonID.ToString
    With PersonDataView
        If .Count > 0 Then
            Dim objPerson As New Person
            With .Item(0)
                objPerson.ID = CType(.Item("ID"), Integer)
                objPerson.FirstName = .Item("NameFirst").ToString.Trim
                objPerson.LastName = .Item("NameLast").ToString.Trim
                objPerson.HomePhone = .Item("PhoneHome").ToString.Trim
                objPerson.CellPhone = .Item("PhoneCell").ToString.Trim
                objPerson.Address = .Item("Address").ToString.Trim
                objPerson.BirthDate = CType(.Item("DateOfBirth"), Date)
                objPerson.EmailAddress = .Item("EmailAddress").ToString.Trim
                objPerson.Favorites = .Item("Favorites").ToString.Trim
                objPerson.GiftCategories = CType(.Item("GiftCategories"), Integer)
                objPerson.Notes = .Item("Notes").ToString.Trim
            End With
            Return objPerson
        Else
            Return Nothing
        End If
    End With
End Function
```

17. Now that you have the database function prepared, return to the `PersonList` control in code view. Add an event at the top of the class to tell the owner of the user control that a request was made to show a person's details:

```
    Public Event ShowPersonDetails(ByVal PersonID As Integer)
```

133

18. Create an event handler subroutine for the `Click` event of the Show Details button. First, check whether the `SelectedItems` count is 1. If the user has selected one entry in the list, then retrieve the `Person` class from the `SelectedItems` object and raise the event with the corresponding ID value. If the count is not 1, then you should display a message informing users that they can show the details of only one person at a time:

```
Private Sub btnShowDetails_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnShowDetails.Click
    If lstPersons.SelectedItems.Count = 1 Then
        Dim SelectedPerson As Person = CType(lstPersons.SelectedItems.Item(0), _
            Person)
        RaiseEvent ShowPersonDetails(SelectedPerson.ID)
    Else
        If lstPersons.SelectedItems.Count = 0 Then
            MessageBox.Show("You must select an entry to display the details")
        Else
            MessageBox.Show("You have too many people selected. Select one only")
        End If
    End If
End Sub
```

19. To intercept the event from the `PersonList`, you need to modify the module-level variable `objPersonList` so that it includes the `WithEvents` keyword. Then you can add an event handler routine for the `ShowPersonDetails` event. It contains code similar to the `Click` event handler for the Add Person button, but in this case you need to retrieve the information from the database first and pass it over as a `Person` object:

```
Private Sub objPersonList_ShowPersonDetails(ByVal PersonID As Integer) _
    Handles objPersonList.ShowPersonDetails
    objPersonalDetails = New PersonalDetails

    Dim objPerson As Person = GetPerson(PersonID)
    objPersonalDetails.Person = objPerson
    objPersonalDetails.AddMode = False

    Me.Text = "Personal Organizer - Viewing " & _
        objPersonalDetails.Person.DisplayName

    If pnlMain.Controls.Contains(objPersonList) Then
        pnlMain.Controls.Remove(objPersonList)
        objPersonList = Nothing
    End If
    pnlMain.Controls.Add(objPersonalDetails)
    objPersonalDetails.Dock = DockStyle.Fill
End Sub
```

20. Run the application again. This time, when you show the list and select a person, you can click the Show Details button, and the information is retrieved from the database and passed to the `PersonalDetails` control, as shown in Figure 7-9.

Figure 7-9

21. At this point, you can modify the `Click` event handler routine for the Add Person button. Remove the initialization code you used in Chapter 6 to populate the fields in the `PersonalDetails` control. This will allow the user control to be initialized with default values when the user clicks the Add Person button, and paves the way for writing code to handle the Save and Cancel buttons that are dynamically created on the user control.

At the end of Chapter 6, you added an event handler in the `MainForm.vb` code to intercept the Save and Cancel buttons' `Click` events. First remove the `MessageBox` line of code. If the user clicks the Cancel button, you should close the `PersonalDetails` user control without doing anything:

```
Private Sub objPersonalDetails_ButtonClicked(ByVal iButtonType As Integer) _
    Handles objPersonalDetails.ButtonClicked
    Select Case iButtonType
        Case 2
            If objPersonalDetails IsNot Nothing Then
                pnlMain.Controls.Remove(objPersonalDetails)
                objPersonalDetails = Nothing
            End If
    End Select
End Sub
```

22. If the Save button is clicked, it's a whole different scenario. You need to add the person to the database and, if successful, return to the `PersonList` where it will be populated with the new information. You'll write an `AddPerson` function in a moment, so add the code to call it and then create the `PersonList` object in a similar way to how the Show List button's `Click` event handler does (change the 1 in the `AddPerson` call to an ID value that is present in your `POUser` table):

```
Private Sub objPersonalDetails_ButtonClicked(ByVal iButtonType As Integer) _
    Handles objPersonalDetails.ButtonClicked
    Select Case iButtonType
        Case 1
            If AddPerson(1, objPersonalDetails.Person) Then
                objPersonList = New PersonList

                If objPersonalDetails IsNot Nothing Then
                    pnlMain.Controls.Remove(objPersonalDetails)
                    objPersonalDetails = Nothing
                End If

                pnlMain.Controls.Add(objPersonList)
                objPersonList.Dock = DockStyle.Fill
            Else
                MessageBox.Show("Person was not added successfully")
            End If
        Case 2
            If objPersonalDetails IsNot Nothing Then
                pnlMain.Controls.Remove(objPersonalDetails)
                objPersonalDetails = Nothing
            End If
    End Select
End Sub
```

23. To add the new information to the database, you use the `AddPersonRow` method of the `PersonDataTable` object. This is inherited from the `AddRow` method of the generic `DataTable` object by Visual Basic Express and includes functions to accept Visual Basic Express data types as parameters. This is handy for fields such as dates that SQL stores in a different way than Visual Basic Express.

The only other thing to be aware of is that because the `Person` table has a foreign key into the `POUser` table, you need to assign a `POUserID` to each `Person` row you add. In the next chapter, you'll modify the call to `AddPerson` so that it includes the currently logged on user's ID, but for now, you'll just use an ID of any record that exists in the database. Define the `AddPerson` function in the `GeneralFunctions.vb` module and create the standard initialization code to create the `DataAdapter` and `DataTable`:

```
Public Function AddPerson(ByVal UserID As Integer, ByVal NewPerson As Person) As _
    Boolean
    Dim AddPersonAdapter As New _PO_DataDataSetTableAdapters.PersonTableAdapter
    Dim AddPersonTable As New _PO_DataDataSet.PersonDataTable
    AddPersonAdapter.Fill(AddPersonTable)
    ... adding code goes here.
    Return True
End Function
```

24. Create another set of data objects, this time for the `POUser` table. These are used to retrieve the `POUserRow` that matches the `UserID` passed into the function:

```
Dim GetUserAdapter As New _PO_DataDataSetTableAdapters.POUserTableAdapter
Dim GetUserTable As New _PO_DataDataSet.POUserDataTable
GetUserAdapter.Fill(GetUserTable)
```

25. The `POUserDataTable` class exposes the `Select` method, which accepts filter criteria. Create an array of `POUserRows` and assign it as the return value for the `Select` method, like so:

```
Dim MyRows() As _PO_DataDataSet.POUserRow = CType(GetUserTable.Select("ID = " & _
    UserID.ToString), _PO_DataDataSet.POUserRow())
```

26. If the array contains data, then you can use the first element in `MyRows` to reference the `POUser` row. Call the `AddPersonRow` method of the `DataTable` object mentioned earlier to add a new row to the table. To save it to the database, use the data adapter's `Update` method:

```
If MyRows.Length > 0 Then
    With NewPerson
        AddPersonTable.AddPersonRow(MyRows(0), .FirstName, .LastName, .HomePhone, _
            .CellPhone, .Address, .EmailAddress, .BirthDate, .Favorites, _
            .GiftCategories, .Notes)
    End With
    AddPersonAdapter.Update(AddPersonTable)
Else
    Return False
End If
```

27. If you run the application as is, you'll get a database failure. This is because the `Person` object in the `PersonalDetails` control is not populated with the information from the user interface components, so before you run the project, add the following code to the `Get` clause of the `Person` object in that control just before you return the `mPerson` object:

```
        With mPerson
            .FirstName = txtFirstName.Text
            .LastName = txtLastName.Text
            .HomePhone = txtHomePhone.Text
            .CellPhone = txtCellPhone.Text
            .Address = txtAddress.Text
            .EmailAddress = txtEmailAddress.Text
            .Favorites = txtFavorites.Text
            .Notes = txtNotes.Text
            .BirthDate = dtpDateOfBirth.Value
        End With
```

28. You now have `AddPerson` and `GetPerson` functions defined in the project — the only additional function you need at this point is the `UpdatePerson` function for when the user is modifying an existing `Person` and clicks the Save button on the toolbar.

29. In the case of an update, you first have to find the row that needs updating. When you find it, you call `BeginEdit` to tell the `DataTable` you're going to change values, change all of the values, and then use `EndEdit` to mark the changes complete. Remember to use the `Update` method of the `DataAdapter` to return the changes to the database itself. Everything else in this function has been discussed in either `GetPerson` or `AddPerson`:

```
Public Function UpdatePerson(ByVal UserID As Integer, ByVal UpdatedPerson As _
    Person) As Boolean
    Dim UpdatePersonAdapter As New _PO_DataDataSetTableAdapters.PersonTableAdapter
    Dim UpdatePersonTable As New _PO_DataDataSet.PersonDataTable
    UpdatePersonAdapter.Fill(UpdatePersonTable)
```

```
        Dim MyRows() As _PO_DataDataSet.PersonRow = _
            CType(UpdatePersonTable.Select("ID = " + UpdatedPerson.ID.ToString), _
            _PO_DataDataSet.PersonRow())
        If MyRows.Length > 0 Then
            With MyRows(0)
                .BeginEdit()
                .NameFirst = UpdatedPerson.FirstName
                .NameLast = UpdatedPerson.LastName
                .PhoneHome = UpdatedPerson.HomePhone
                .PhoneCell = UpdatedPerson.CellPhone
                .Address = UpdatedPerson.Address
                .EmailAddress = UpdatedPerson.EmailAddress
                .DateOfBirth = UpdatedPerson.BirthDate
                .Favorites = UpdatedPerson.Favorites
                .GiftCategories = UpdatedPerson.GiftCategories
                .Notes = UpdatedPerson.Notes
                .EndEdit()
            End With
            UpdatePersonAdapter.Update(UpdatePersonTable)
        End If

        Return True
    End Function
```

30. When the user clicks the Save button on the main form, you should first determine whether the
PersonalDetails control is showing. If it is and the AddMode property is True, then you
should call the AddPerson function to add the new information to the database. If the
PersonalDetails control is visible but the AddMode property is set to False, then the user
must be updating an existing record, so you should call the UpdatePerson function:

```
Private Sub saveToolStripButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles saveToolStripButton.Click
    If objPersonalDetails IsNot Nothing Then
        If objPersonalDetails.AddMode = True Then
            If AddPerson(1, objPersonalDetails.Person) Then
                MessageBox.Show("Person was added successfully")
                objPersonList = New PersonList

                If objPersonalDetails IsNot Nothing Then
                    pnlMain.Controls.Remove(objPersonalDetails)
                    objPersonalDetails = Nothing
                End If

                pnlMain.Controls.Add(objPersonList)
                objPersonList.Dock = DockStyle.Fill
            Else
                MessageBox.Show("Person was not added successfully")
            End If
        Else
            If UpdatePerson(1, objPersonalDetails.Person) Then
                MessageBox.Show("Person WAS updated successfully")
            Else
                MessageBox.Show("Person was not updated successfully")
            End If
```

```
        End If
      End If
  End Sub
```

31. As an additional feature, you should also place the Gift Categories onto the `PersonalDetails` user control. Add six `CheckBox` controls to the user control design surface. You need to move the Notes area down to make room. Set their `Text` properties so that they match the ones shown in Figure 7-10 and name them accordingly — that is, the `Books CheckBox` should be named `chkBooks`, and so on.

Figure 7-10

32. Open the `PersonalDetails` control in code view and add the following code to the `ResetFields` routine so that the `CheckBoxes` are returned to their default state:

```
chkApparel.Checked = False
chkBooks.Checked = False
chkToys.Checked = False
chkVideos.Checked = False
chkVideoGames.Checked = False
chkMusic.Checked = False
```

33. Because all of the gift category flags are stored in a single integer in the database, you need some way to translate between them. Visual Basic Express enables you to do what's known as *bitwise comparisons,* comparing individual bits of a number. This is possible because all numbers are represented in binary form. For example, the number 2 is represented by the binary number 10, while the number 9 is represented by the binary value 1001, where the first 1 represents 8, the last 1 represents 1, and the middle two zeros represent 4 and 2.

When you compare two numbers using `Or` and `And`, Visual Basic Express automatically translates this for you, so if you define each of your categories using a different position in the binary stream, you can uniquely identify whether they are set.

34. Create a private `Enum` at the top of the `PersonalDetails` code to define the numbers that identify each gift category:

```
Private Enum CategoryValues
    Books = 1
    Videos = 2
    Music = 4
    Toys = 8
    VideoGames = 16
    Apparel = 32
End Enum
```

None of these numbers overlap, so if a `Person` had the `Music` and `Toys` categories set, then the `GiftCategories` value would be 4 + 8 = 12.

35. Modify the `Set` clause of the `Person` property to set the `CheckBox` values if the corresponding bit in the `GiftCategories` property is set. The following code compares the `GiftCategories` value against each `Enum` value and, if there is a match, sets the corresponding `CheckBox`:

```
chkBooks.Checked = (mPerson.GiftCategories And CategoryValues.Books) <> 0
chkVideos.Checked = (mPerson.GiftCategories And CategoryValues.Videos) <> 0
chkMusic.Checked = (mPerson.GiftCategories And CategoryValues.Music) <> 0
chkToys.Checked = (mPerson.GiftCategories And CategoryValues.Toys) <> 0
chkVideoGames.Checked = (mPerson.GiftCategories And CategoryValues.VideoGames) <> 0
chkApparel.Checked = (mPerson.GiftCategories And CategoryValues.Apparel) <> 0
```

36. Now modify the `Get` clause to calculate a new `GiftCategories` value based on the states of the `CheckBoxes`. This is done by effectively reversing the preceding code:

```
Dim GiftCategorySetting As Integer = 0
If chkBooks.Checked Then GiftCategorySetting = GiftCategorySetting Or _
    CategoryValues.Books
If chkVideos.Checked Then GiftCategorySetting = GiftCategorySetting Or _
    CategoryValues.Videos
If chkMusic.Checked Then GiftCategorySetting = GiftCategorySetting Or _
    CategoryValues.Music
If chkToys.Checked Then GiftCategorySetting = GiftCategorySetting Or _
    CategoryValues.Toys
If chkVideoGames.Checked Then GiftCategorySetting = GiftCategorySetting Or _
    CategoryValues.VideoGames
If chkApparel.Checked Then GiftCategorySetting = GiftCategorySetting Or _
    CategoryValues.Apparel
.GiftCategories = GiftCategorySetting
```

Now you can go ahead and run your application. When you edit or create a person, you'll see the six `CheckBox` controls in the `PersonalDetails` control. When you select different values and then save them to the database, the code combines the values to form a single integer that can be stored in the database. When it reads them back out, your code converts them back to individual flags for the `CheckBox` controls.

Summary

Accessing the database once was a difficult process, but Visual Basic Express provides you with several methods for doing so that without exception are easy to implement. Whether you build your data access with the `DataGridView` or by binding simple components such as `TextBox` and `ComboBox` controls to a data source, you can present information to the user without writing a single line of code.

In addition, even when you need to build code, the functions to do so are simplified by Visual Basic Express's capability to create customized versions of the `DataTable` and `DataAdapter` classes that expose just the right number of properties and functions that you need to get the job done.

In this chapter, you learned to do the following:

❏   Create simple database access functionality through the `Data` controls

❏   Use controls that have the ability to be bound to data sources so you don't have to write your own code

❏   Build program functions that can be used to select information programmatically from within a database

In Chapter 8, you'll return to the coding side of Visual Basic Express, where you'll learn about the special `My` namespace Microsoft has built just for Visual Basic programmers, along with how collections can be used to store data that is alike.

Exercise

1.   Add four more routines to the `GeneralFunctions.vb` module to perform the following functions:

    a.   Determine whether a specified user exists.

    b.   Determine whether a user's password matches a given string.

    c.   Create a new user record.

    d.   Update a user record's Last Logged In value.

These functions are needed for the next chapter, so make sure you do them all!

8

It's *My* World — Isn't It?

Visual Basic Express is one of those programming environments that just keeps on giving. If the visual aids, constant feedback cues, ease of design, and simple programming model aren't enough for you, this chapter will reveal even more features that make Visual Basic Express the language of choice for developers, from beginners to professionals.

The My namespace is a new section of .NET designed specifically for Visual Basic programmers. It serves to simplify many complex areas of Windows into a series of basic objects and methods. This collection of classes and other more advanced features of Visual Basic Express, such as generics and partial classes, are the subject of the next few pages.

In this chapter, you learn about the following:

❑   Using My classes to simplify complex tasks

❑   Creating classes in pieces and building collections generically

❑   Extending the Personal Organizer for multiple users

They're *My* Classes

The .NET Framework is a robust and rich collection of classes organized into a hierarchy of categories known as *namespaces*. These classes are automatically exposed to Visual Basic Express, which means you can use any of them in your applications. In fact, the various objects you've been working with are actually part of that Framework. While the .NET Framework is not the subject of this book, knowing how it works can be handy. Appendix B runs through the fundamentals of the Framework, with a focus on some of the more interesting sets of classes.

The new My namespace is a special case. Most of the operating system is accessible through the main .NET Framework classes. Sounds, graphics, files, and hardware settings can be retrieved and used by manipulating information through the classes exposed by the .NET Framework. The challenge lies in the complexity of retrieving the bits and pieces required to do any one action.

For example, in previous versions of the Visual Basic language, sending data to the default printer in the system required a couple of lines of code—one to send the information to a printer queue and another to tell the printer to print.

In .NET all of that changed, which required the creation and monitoring of a printer object. It was up to the printer object to raise an event when it was ready to print, and then you would pass in the next page of information for printing. Then this process would be repeated until you finished.

As another, simpler example, reading a file using standard .NET classes would require at least three lines of code, and that's using a concatenated definition and instantiation. The main class involved was also sometimes hard to remember:

```
Dim MyFileReader As New IO.StreamReader("C:\PersonalDetails.txt")
Dim sPersonalDetails As String = MyFileReader.ReadToEnd
MyFileReader.Close()
```

Because programmers using previous versions of Visual Basic with .NET experienced this increased level of difficulty in accessing fairly commonplace functionality, Microsoft introduced a whole new namespace called My, and Visual Basic Express users are the first to be able to take advantage of it.

Think of the members you find in My classes as shortcuts to other parts of the .NET Framework. They give Visual Basic Express programmers the edge in accessing tasks that are performed often in Windows applications, such as printing and file processing, while also simplifying other system-related tasks that were difficult in all previous versions of Visual Basic.

Of the two examples mentioned, the printing classes found in My return to the simplicity of referencing the printer and sending the information directly to it. All the complexity of the internal printer object raising events when it has completed printing each page and waiting for the next chunk of data is hidden away, and all you need to do is tell it to print. And the file example—don't you think the following line of code is easier to understand?

```
Dim sPersonalDetails As String = _
    My.Computer.FileSystem.ReadAllText("C:\PersonalDetails.txt")
```

It's All about the Computer

The majority of objects within the My namespace deal with the computer system. From the simple but still extremely useful methods giving you access to the system clock and clipboard to the much more complex structures that enable you to access and manipulate files and hardware devices such as any printers connected to the computer, My.Computer makes it a straightforward process.

The main My.Computer object serves as a launching pad for the subordinate classes that divide the system into a number of discrete components (those children classes are the subject of the next sections in this chapter). The only other property of note is the name of the computer, aptly called Name. You can use this property as you would any other class property:

```
txtComputerName.Text = My.Computer.Name
```

Figure 8-1

Using custom data is almost as easy. The only difference is that you need to pass the name of the data format you are using. Unless you're trying to use the same data format as another application, you can make up any name you desire:

```
My.Computer.Clipboard.SetData("MyFormat",MyObject)
If My.Computer.Clipboard.ContainsData("MyFormat") Then
    MyObject = My.Computer.Clipboard.GetData("MyFormat")
End If
```

My.Computer.Clock

The `My.Computer.Clock` object contains properties for getting the current system time and date in both local time and GMT (Greenwich Mean Time), also known as UTC. The `Clock` object cannot be used to change the system time, but as that is rarely a need for a Visual Basic application, you shouldn't find yourself too disappointed by that.

Both `LocalTime` and `GmtTime` return full `Date` variables, which can then be used in any kind of date manipulation that you use for other dates. `LocalTime` is equivalent to the special Visual Basic keyword `Now`.

My.Computer.Info

If your application needs to know anything about the state of the computer, or wants to report this information back to the user, the `My.Computer.Info` class will be immensely useful. With `Info`, you have access to the computer's name, the operating system name and version, current memory usage, and the selected culture of the system.

With this information, you can make decisions about what to do with your application. For example, if you determine that the system culture is not U.S. English, you might want to display a message to the user in multiple other languages. The `InstalledUICulture` object contains many properties that return this system-specific information, including the kind of calendar the user is using and formats of date, time, and money.

The next Try It Out uses `My.Computer.Info` to display information about the computer on which the application is running, including memory and the calendar and date and time formats that are set.

Try It Out   **Accessing System Information**

1.   Start a new Windows Application project and place a button and three labels on the form. Create an event handler routine for the button's `Click`.

2.   In the `Click` event handler, set the label's `Text` property to the amount of memory currently available:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    With My.Computer.Info
        Label1.Text = "Memory (Available/Total): " & _
            .AvailablePhysicalMemory.ToString & "/" & _
            .TotalPhysicalMemory
    End With
End Sub
```

3.   The other labels are to contain information about the culture — the calendar being used and the short date format:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    With My.Computer.Info
        Label1.Text = "Memory (Available/Total): " & _
            .AvailablePhysicalMemory.ToString & "/" &  _
            .TotalPhysicalMemory
        Label2.Text = .InstalledUICulture.Calendar.ToString
        Label3.Text = .InstalledUICulture.DateTimeFormat.ShortDatePattern.ToString
    End With
End Sub
```

4.   If you run the application the way it is, you'll find that the memory values are displayed in bytes and are not formatted for easy reading. Change that part of the `Click` event routine so that the memory is displayed in megabytes by dividing the values by 1,024 to get kilobytes, and again to get megabytes.

Modify the `ToString` methods to include a format string. This will display the final numbers with thousand separators:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    With My.Computer.Info
        Dim AvailableMemory As Double = .AvailablePhysicalMemory / (1024 * 1024)
        Dim TotalMemory As Double = .TotalPhysicalMemory / (1024 * 1024)
        Label1.Text = "Memory (Available/Total): " & _
            AvailableMemory.ToString("#,###") & "/" & _
            TotalMemory.ToString("#,###") & "Mb"
    End With
End Sub
```

Run the application to observe the results. Your date format may differ from the one shown in Figure 8-2 — this is a system setting that a lot of people change to suit their own styles.
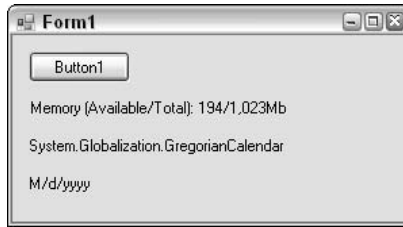
Figure 8-2

My.Computer.Screen

The `My.Computer.Screen` object provides a shortcut to the `PrimaryScreen` property in the `System.Windows.Forms.Screens` namespace. It returns information about the user's current monitor settings, including screen resolution and bit depth.

One particularly useful property is the `WorkingArea` object, which is returned as a `Rectangle`. `Rectangle` objects contain a number of values defining the area that is enclosed, and they are used extensively by Windows to define windows, forms, and control sizes and positions. In this case, the `Rectangle` variables of interest are `Height` and `Width`. The values stored in these properties specify the total working area of the screen — that is, the part of the screen not taken up by the Windows system tray, taskbars, and any other system-controlled component that takes away screen real estate from your application.

My.Computer.Audio

`My.Computer.Audio` provides a series of methods to play audio in your application. The main `Play` method is overloaded to enable your application to play audio wave files from different sources — a normal file, an IO stream, and a `Byte` array.

In addition, the `Play` method enables you to control how the sound should be played. The application can either wait for the sound to finish playing before it continues or continue executing while the sound plays in the background. If the background option is chosen, it can be set to continuously loop until the application explicitly stops it with the `Stop` method.

A simple example for this kind of use would be playing music while a particularly long process was taking place:

```
My.Computer.Audio.Play(MyWaveFileName, AudioPlayMode.BackgroundLoop)
MyReturnValue = SomeVeryLongFunction()
My.Computer.Audio.Stop
```

My.Computer.Mouse

The `My.Computer.Mouse` object enables your application to get the status of several mouse properties. Whether the computer detected a mouse at all and, if so, whether the mouse has a scroll wheel are returned in the Boolean properties `Exists` and `WheelExists`. `WheelScrollLines` returns the user setting specifying how many lines are supposed to be scrolled for every notch the mouse wheel is turned. Finally, `ButtonsSwapped` helps determine whether the user is left-handed and has swapped the right and left button functionality.

My.Computer.Keyboard

Getting information about the state of the keyboard is even more useful than information about the mouse. `My.Computer.Keyboard` is an object that provides such status information, such as whether the Caps Lock key is on, or whether the user is currently pressing the Shift or Alt keys. In addition to the status monitors, `Keyboard` has a `SendKeys` method to programmatically emulate the pressing of keys on the keyboard.

The following Try It Out uses the `My.Computer.Keyboard` object along with `My.Computer.Mouse` and `My.Computer.Clipboard` to programmatically copy text from one `TextBox` and paste it into another. It illustrates how the various `My.Computer` objects can work together to easily perform functions that might otherwise take many lines of code to perform.

Try It Out   **Sending Keystrokes with SendKeys**

1. Create a new Windows Application project and add a `Button` and two `TextBox` controls to the form. Make sure you resize `TextBox2` so that it will have room for multiple lines of text.

2. Add an event handler for the `Button`'s `Click` event and first set the `Text` property of `TextBox1` depending on whether the Alt key is being held down or not:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    If My.Computer.Keyboard.AltKeyDown = True Then
        TextBox1.Text = "The cat slept."
    Else
        TextBox1.Text = "The dog jumped."
    End If
End Sub
```

3. If the Caps Lock is on, the program should copy the animal name from `TextBox1` to the system clipboard as text so you can paste in the other `TextBox`:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    If My.Computer.Keyboard.AltKeyDown = True Then
        TextBox1.Text = "The cat slept."
    Else
        TextBox1.Text = "The dog jumped."
    End If
    If My.Computer.Keyboard.CapsLock = False Then
        With TextBox1
            .SelectionStart = 4
            .SelectionLength = 3
            My.Computer.Clipboard.SetText(.SelectedText)
        End With
    End If
End Sub
```

4. You're going to use `SendKeys` to programmatically emulate keystrokes in `TextBox2`, so first put the cursor on that control using its `Focus` method. Then create a loop that will run for a number of times equal to the `WheelScrollLines` property of the `Mouse` object. In the loop, you'll paste the clipboard text by emulating Ctrl+V followed by the Enter key to force a new line between each paste operation.

The code will then Shift+Tab back to `TextBox1`, delete the selected text, and replace it with the word `mouse`. The final subroutine appears as follows:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    If My.Computer.Keyboard.AltKeyDown = True Then
        TextBox1.Text = "The cat slept."
    Else
        TextBox1.Text = "The dog jumped."
    End If
    If My.Computer.Keyboard.CapsLock = False Then
        With TextBox1
            .SelectionStart = 4
            .SelectionLength = 3
            My.Computer.Clipboard.SetText(.SelectedText)
        End With
        With TextBox2
            .Focus()
            For iCounter As Integer = 1 To My.Computer.Mouse.WheelScrollLines
                My.Computer.Keyboard.SendKeys("^V~")
            Next
        End With
        My.Computer.Keyboard.SendKeys("+{TAB}{DEL}mouse")
    End If
End Sub
```

The strings in the `SendKeys` methods may be a little unusual but once you're familiar with the various control symbols it should be straightforward. The control keys are signified with the following replacements:

Key	Symbol to Use
Shift	+
Control	^
Alt	%
Enter	~ (or can be specified as {ENTER}

After running the application and clicking the button, the result should look like Figure 8-3.
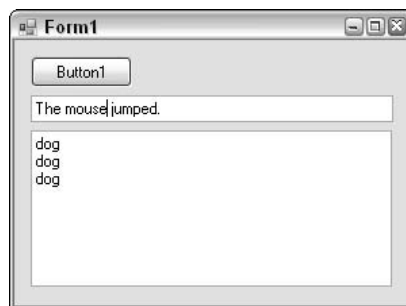
Figure 8-3

My.Computer.Registry

Traditionally, while the Windows Registry is used to store immense amounts of information about applications and Windows settings, it has been hard to work with by Visual Basic programmers. .NET made it a little easier, but accessing Registry settings was still an awkward task that should have been easy.

`My.Computer.Registry` revolutionized all that and Visual Basic Express can take full advantage of this new object. The main properties of this object offer direct access to the root folders within the Registry such as `Local Machine`. Each of these properties is returned as a `RegistryKey` object, which is rich with methods to manipulate the data found within each node of the Registry.

For example, retrieving the version number of the installed copy of Internet Explorer could be done in two lines:

```
Dim RK As RegistryKey = My.Computer.Registry.LocalMachine.OpenSubKey( _
    "SOFTWARE\Microsoft\Internet Explorer")
Dim IEVers As String = RK.GetValue("Version", "Internet Explorer not installed")
```

The important thing to remember when working with the Registry is that you should treat it the same way as you would a file. That is, if you open a part of the Registry for processing, then you should also close it when you're done. If you do not follow this kind of procedure, you could end up corrupting the Registry data, which in turn can cause problems in the system, possibly as severe as preventing Windows from starting.

The methods you will most likely use to retrieve information are as follows:

- ❏ **OpenSubKey** — Opens the location within the specified root node in the Registry and returns a `RegistryKey` object.

- ❏ **GetValue** — Returns the value found for the specified name and optionally includes a default value if the name is not found.

- ❏ **Close** — Cleanly closes the location within the Registry.

In addition to these three methods, you can also create folders of information, set individual values, and delete both:

- ❏ **CreateSubKey** — Creates a folder within the Registry within the current `RegistryKey` context. For example, to create a folder called `MySettings` within the `CurrentUser` root node, you would use `My.Computer.Registry.CurrentUser.CreateSubKey("MySettings")`.

- ❏ **SetValue** — Assigns a new value to the specified name within the current `RegistryKey`.

- ❏ **DeleteValue** — Removes the specified name from within the current location.

- ❏ **DeleteSubKey** — Deletes an entire folder from the `RegistryKey` specified. If the folder contains other folders, you must either delete those first or use the `DeleteSubKeyTree` method instead.

My.Computer.Network

When the computer is connected to a network, the `My.Computer.Network` object can be used to transfer files between the local machine and a remote computer. The cool thing about this is that a "network" includes being connected to the Internet, so downloading a file from a remote location (assuming you have permission) can be implemented with a single line of code, as shown here:

151

```
My.Computer.Network.DownloadFile("http://www.myurl.com/thefile.txt", _
    "C:\Downloaded Files")
```

Uploading a file is similarly straightforward, and the only thing you should make sure you do before performing either action is check whether the network is available. This is also made easy for you — the `My.Computer.Network.IsAvailable` variable returns `True` or `False` depending on the network's status.

My.User

If you need to write code based on the current user, the `My.User` object will be a tool of choice. With this object, you can interrogate the system to determine whether a valid user is currently logged on, and if so, what his or her official system name is. You can also use this object to determine whether the user belongs to a particular user group and potentially change the permissions on your own application based on their membership.

Consider the following snippet of code that could be included as part of the startup of a program. It checks the current user, changes the title bar text of the form to include the name, and then hides the View Options button if the user is not defined as an Administrator:

```
With My.User
    If .IsAuthenticated Then
        Me.Text = "Personal Organizer - logged in as " & .Name
        If .IsInRole("BUILTIN\Administrators") Then
            btnViewOptions.Visible = True
        Else
            btnViewOptions.Visible = False
        End If
    End If
End With
```

My.Computer.Printers and My.Computer.FileSystem

With all of those objects out of the way, you are left with the two big guns — `FileSystem` and `Printers`. The `Printers` object enables you to access each printer defined in the system and send data to them to be printed. The objects also return information about the capabilities of each printer, including the printable area and printer resolution. In Chapter 11, you'll learn how to send information to a printer using the printer classes Visual Basic Express provides.

`My.Computer.FileSystem` represents a number of functions that can be performed on the files and folders of the computer system. While reading a file into a string such as the one shown in the example at the beginning of this chapter might be one function you need to perform in your application, the `FileSystem` object enables you to manipulate the file system, performing such actions as copy, rename, and delete on files and folders. The following list presents some commonplace tasks that you can perform using the FileSystem methods:

❑ **Copy a file** — Use the `CopyFile` method, specifying the source and destination filenames. Optionally, you can indicate whether any existing file should be replaced, whether the Windows-defined animation should be displayed while the file operation is being performed, and even a new name if it differs from the original.

❑ **Copy an entire folder** — The `CopyDirectory` method can be used to copy an entire directory structure to another location on the computer. It has the same set of options as `CopyFile`.

❑ **Rename a file** — Use `RenameFile` to change the name of an existing file. `RenameDirectory` performs the same action but on a folder.

❑ **Delete a file** — Call the `DeleteFile` method to delete a file. Optionally, you can display the Windows defined animation while the file is being deleted and send the file to the Recycle Bin instead of permanently deleting it. Unsurprisingly, there is a `DeleteDirectory` method to remove a directory of information with similar options. It has an additional option to recursively delete subdirectories as well.

❑ **Get a list of subfolders** — The `GetDirectories` method returns an array of string values, each containing the name of a subdirectory belonging to the folder specified.

❑ **Determine whether a file or folder exists** — `DirectoryExists` and `FileExists` return Boolean values to indicate whether the specified element exists or not.

❑ **Get the current directory** — Use `CurrentDirectory` to retrieve the name of the current location in the file system.

In addition to these basic methods, the `FileSystem` object also enables you to retrieve the current locations of the Windows Special Folders, such as My Music and Temp. These folders are represented by the `SpecialDirectories` collection and all return a string containing the *absolute path* to the particular folder.

The `GetRelativePath` and `CombinePath` methods can be used to work with *relative paths* within the file system. A relative path is one that indicates the location of a folder or file based on the current location, whereas an absolute path includes the information to get to the file regardless of the current location. Consider the following scenarios to access a file named `temp.txt` in the Windows System folder.

Current Location	Absolute Path	Relative Path
C:\Windows	C:\Windows\System\temp.txt	System\temp.txt
C:\Windows\Drivers	As above	..\System\temp.txt
C:\Program Files	As above	..\Windows\System\temp.txt

Getting to the App

The other group of classes within the `My` namespace deal with different parts of your application. These objects are used to process the settings and components that define the program. `My.Application` returns such information as the command-line arguments with which the program was started. It also has properties that return application-specific versions of `My.Computer` values, such as `Culture`.

`My.Application` also has an `OpenForms` collection that you can use to iterate through the current forms belonging to your application that are open. This can be useful if you want to reuse a form for different purposes and need to know whether it exists already. The `ApplicationContext.MainForm` and `SplashScreen` properties identify the forms that the application uses to start with and refer to the same values that are accessible in the My Project page in the Solution Explorer (refer to the sidebar "My Project" for more information).

My Project

Used alongside the application-related `My` classes, the My Project page in the Solution Explorer provides design-time access to a myriad of settings related to the project as a whole, including the form that should be used as the startup object and the splash screen.

Six categories of options can be accessed in My Project:

Application — Here you can specify the namespace that identifies your program, which form to use at startup, and the icon associated with the application. In addition to these values, the Application category enables you to specify a form that will act as a splash screen for your program. This form will be shown while the application starts and initializes any settings. Once the program is ready to show the main form, it will automatically hide the splash screen form for you.

References — A list of all references made in the project. These can be system references to parts of the .NET Framework so you can use the associated classes; COM references to external objects; and web services (covered in Chapter 9).

Debug — The Debug category enables you to include command-line arguments with which you can test the application, as well as specify a starting working directory. This latter option is useful if you have commonly used files that you would like to use in place, rather than having to navigate from the default debug folder in the solution.

Compile — You can control the way the program is built by Visual Basic Express with this page of options. You won't normally need to change any of these settings at all.

Resources — Briefly mentioned in Chapter 2, the Resources page contains lists of all resources associated with your application. This is where you can manage any images and audio files you've imported into the project. You can also add strings of text and even whole files in the Resources page, which can be useful for storing commonly used pieces of information.

Settings — Used to store custom information, the individual settings can be defined for an individual user or for the application as a whole.

The next Try It Out creates an application with two forms, marks one of them as the splash screen, and then displays several settings from the `My.Application` and `My.Computer` objects in a `TextBox`.

Try It Out   **Using My Project and My.Application**

1. Create a new Windows Application project. With the form that is added by default, set the following properties:

    ❑   **FormBorderStyle** — `None`

    ❑   **BackColor** — `255, 255, 192` (a pale yellow)

    ❑   **Cursor** — `AppStarting`

    ❑   **StartPosition** — `CenterScreen`

2. Add a new `Form` with the Project ➪ Add Windows Form menu command. In this second form, add a `TextBox` and resize it so that it fills most of the form.

3. Double-click a part of the form not taken up by the `TextBox` to automatically create an event handler for the `Load` event of the form. Add the following code:

```
Private Sub Form2_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    With My.Application
        TextBox1.Text = "Splash screen = " + .SplashScreen.Name
        TextBox1.Text += vbCrLf + "Start up form = " + _
            .ApplicationContext.MainForm.Name
        TextBox1.Text += vbCrLf + "Current Directory = " + _
            My.Computer.FileSystem.CurrentDirectory
    End With
End Sub
```

4. Double-click the My Project entry in the Solution Explorer and change the options to set the Startup object to `Form2` and the Splash screen to `Form1`. Also change the Working directory in the Debug section to `C:\Temp`. If the directory you specify doesn't exist, you get an error when you try to run the application.

5. Run the application. Initially, the form you set to pale yellow will be displayed for a moment while the rest of the application is created and prepared for execution. Then this form is removed and the second form is displayed, with text identifying values that are part of `My.Application` (see Figure 8-4).
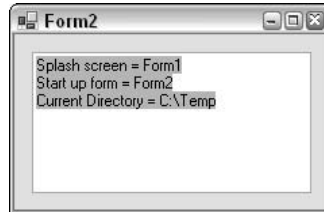
Figure 8-4

The remaining `My` classes provide access to the other parts of your application not covered by `My.Application`. The `My.Forms` collection enables you to programmatically process each form that you have defined in the project. This includes the capability to navigate right to the controls housed on the form, either directly through their names or through the form's `Controls` collection.

`My.Resources`, `My.Settings`, and `My.WebServices` all reference different parts of the My Project page. Each resource or setting is dynamically exposed as a property of its respective type, while `My.WebServices` enables you to call the web service methods referenced in the application.

The `My` objects that come with Visual Basic Express definitely make system-related tasks easier to accomplish. Whether it's file manipulation or printing, retrieving the date and time, or determining user roles, `My` helps you write code that is easy to manage and track.

You Can Use It Again and Again . . . and Again

Sometimes you'll need some code that you already know you're going to reuse repeatedly. Or you might be in a bit of a dilemma about the best way to achieve something. Enter another Visual Basic Express feature designed to make your life easier — code snippets.

Code snippets are small (well, sometimes not so small) pieces of code stored in a library and organized into categories. Whenever you're in the code view and need to write code for a common task, you can often use the code snippet library provided by Microsoft to write it for you.

The way it works is simple: You browse the category hierarchy until you find the task you're trying to perform and select it from the list. Visual Basic Express inserts the code at the cursor location and marks any parts that need to be replaced with your own code, including variable names and literal values. Change the marked areas and you're done — a complete section of code ready to use in your program.

The code snippet library is *contextual*. This means that you'll get a different set of snippets if you open the library with the cursor inside a subroutine or function than if the cursor were in the class itself and outside of any routines. It makes sense to divide the snippets in this way — you're unlikely to want to create a whole new routine inside another, and if you insert code without enclosing it in a function definition it won't even compile.

To bring up the code snippet library, place the cursor where you want to insert the code and right-click. From the context menu, select Insert Snippet…. A smart IntelliSense-like dialog will appear, displaying the main list of categories from which you can choose. As you choose each category of items, it will be inserted onto the form as a hyperlink, and the next list will be displayed. If you realize you have made a mistake, you can double-click these hyperlinks to return to a previous category of items.

When you find the particular snippet that meets your needs, click it, and the IntelliSense and hyperlinks will be replaced with the actual snippet. Code that should be replaced by your own values is highlighted in yellow (this color can be changed in the Options page for fonts). In some cases, the code will compile as is and can provide functions or subroutines that can be called without modification. Mostly, however, you will want to change either the literal values or the controls to which the code refers.

In the next Try It Out, you'll create an application with vertically drawn text. Rather than create the routine that will draw text vertically from scratch, you'll use the code snippet library to automatically create a base definition that you can then modify.

Try It Out   **Using Code Snippets**

1. Create a new Windows Application project in Visual Basic Express, add a `Button` and a `TextBox` to the form, and create a `Click` event handler routine for the button.

2. Above the `Click` event handler, right-click and select Insert Snippet…. Because the code snippet is actually an entire subroutine, you'll add it outside any other routines. From the Insert Snippet list, select Creating Windows Forms Applications ⇨ Drawing ⇨ Draw Vertical Text on a Windows Form.

Visual Basic Express will automatically insert the following code:

```
Public Sub DrawVerticalString()
    Dim drawString As String = "hello"
    Dim x As Single = 150.0
    Dim y As Single = 50.0
    Dim drawFormat As New StringFormat()

    Using formGraphics As Graphics = Me.CreateGraphics(), _
        drawFont As New System.Drawing.Font("Arial", 16), _
        drawBrush As New SolidBrush(Color.Red)

        drawFormat.FormatFlags = StringFormatFlags.DirectionVertical
        formGraphics.DrawString(drawString, drawFont, drawBrush, _
            x, y, drawFormat)
    End Using
End Sub
```

The routine gets the graphics object it needs to draw on, creates the system objects needed to draw text — `Font` and `Brush` — and draws the specified string using a format flag to indicate vertical direction. It then cleans up after itself.

3. Because you want to pass in your own text, add a parameter in the definition of the `Sub` and change the `drawString` definition to use this parameter instead of the literal:

```
Public Sub DrawVerticalString(ByVal StringToDraw As String)
    Dim drawString As String = StringToDraw
...
```

4. Return to the `Click` event handler you created and add a call to the `DrawVerticalString` method, passing in the `Text` property of the `TextBox`:

```
DrawVerticalString(TextBox1.Text)
```

5. Run the application, enter some text in the `TextBox`, and click the button. You've created an application that draws text directly on the form in a vertical orientation, without having to remember the objects you needed (see Figure 8-5).

Figure 8-5

Reusing Code Properly

Two relatively recent advances in programming languages are *generics* and *partial classes*. While code snippets enable you to reuse common blocks of code, it's far more likely that you'll need to use code repeatedly but in slightly different contexts. Both partial classes and generics enable you to do that easily.

Partial Classes

Partial classes are a new way of creating classes from multiple files. This feature of Visual Basic Express enables you to build a single class in your application from multiple definitions, effectively building them all together into one cohesive class.

As a simple example, consider the following two class definitions:

```
Partial Public Class MyClass
    Private mMyString As String
End Class
```

```
Partial Public Class MyClass
    Public Property MyString() As String
        Get
            Return mMyString
        End Get
        Set(ByVal value As String)
            mMyString = value
        End Set
    End Property
End Class
```

Both classes have the same name and would normally cause a compilation error just because of that alone. The other problem is that the top class has a private variable, while the bottom class has a public property that references a variable that doesn't exist within that class definition.

However, because they are both marked as `Partial`, Visual Basic Express will bring them together, treating them as a single class:

```
Partial Public Class MyClass
    Private mMyString As String

    Public Property MyString() As String
        Get
            Return mMyString
        End Get
        Set(ByVal value As String)
            mMyString = value
        End Set
    End Property
End Class
```

Visual Basic Express also understands partial classes at design time, so you won't get any of those visual indicators that something is wrong with your code because of a missing definition.

While partial classes may not seem very useful, they can be extremely valuable under certain conditions. What they enable you to do is create one part of a class that is used by multiple applications, and then enhance that partial class with application-specific classes to tailor the code to fit the requirements.

To illustrate this process, consider the following main class, saved as `MyClassHeader.vb`. It contains all the code necessary to store and maintain three properties related to an employee:

```
Partial Public Class MyClass
    Private sFirstName As String
    Private sLastName As String
    Private dSalary As Decimal

    Public Property FirstName() As String
        Get
            Return sFirstName
        End Get
        Set(ByVal value As String)
            sFirstName = value
        End Set
    End Property
    Public Property LastName() As String
        Get
            Return sLastName
        End Get
        Set(ByVal value As String)
            sLastName = value
        End Set
    End Property
    Public Property Salary() As Decimal
        Get
            Return dSalary
        End Get
        Set(ByVal value As Decimal)
            dSalary = value
        End Set
    End Property
End Class
```

Two applications are created, both adding the `MyClassHeader.vb` file to their projects. The first one also adds `MyClassFinancial.vb`, while the second application adds `MyClassPersonal.vb`. These partial classes are shown in the following table.

159

MyClassFinancial.vb	MyClassPersonal.vb
```Partial Public Class MyClass Public Sub AddSalary(ByVal IncAmount _ As Decimal) dSalary += IncAmount End Sub End Class Public ReadOnly Property _ DisplayName() As String Get Return sFirstName + sLastName End Get End Property End Class```	```Partial Public Class MyClass```

When the code is created for the first application, it will have access to the three properties, along with the `AddSalary` method, but not the `DisplayName` read-only property. The second application, however, can use the `DisplayName` property along with the three main properties, but it doesn't know a thing about the `AddSalary` method.

Generics

Generics are a way of defining a single class that can be defined for multiple types. You create a class definition that can contain any normal class members — private variables, properties, methods, and so on. When you define an object of that class, you pass over what type it should be, and the internal class structure will work as if it were originally defined as that type.

While it may sound a little complicated, it's one of those concepts that becomes clear once you've seen it in action. Consider the following class:

```
Public Class MyClass
    Private mItemValue As String
    Public Property ItemValue() As String
        Get
            Return mItemValue
        End Get
        Set(ByVal value As String)
            mItemValue = value
        End Set
    End Property
End Class
```

Now, suppose you needed a similar class but the `ItemValue` property were an `Integer`. You could create a whole new class, repeating all the code within the `MyClass` and replacing all instances of `String` with `Integer`. Or you could use generics.

To create a generic class, you define it exactly the same way, but add an extra clause at the top of the class definition, `Of`. The `Of` keyword is followed by the name you will use to identify the type that the class will become whenever it's instantiated. Taking the previous class example, you could convert it to a generic like so:

```
Public Class MyClass(Of MyType)
    Private mItemValue As MyType
    Public Property ItemValue() As MyType
        Get
            Return mItemValue
        End Get
        Set(ByVal value As MyType)
            mItemValue = value
        End Set
    End Property
End Class
```

Whenever you define an object of type `MyClass`, you need to specify the type that it should be treated as, using the same `Of` clause:

```
Dim MyStringClass As MyClass(Of String)
Dim MyIntegerClass As MyClass(Of Integer)
```

While the preceding example is simple, generic classes can be used to create complex reusable objects, and they are perfect for building custom array and collection type objects. The main requirement you need to remember is that every statement within the class definition must work with every type that is used with it. It wouldn't be possible, for example, to include mathematical functions in the `Set` clause in the preceding example because one of the objects defines it with a `String` type.

Visual Basic Express comes with several built-in generic classes, all in the generic namespace. These classes emulate existing object types, such as collections and other list types, but because the definition of an object with a generic class requires you to define the particular type that will be used, you can constrain the items that will exist in the class without having to write code to do it. The following two `Collection` classes look similar, but while the first one will run without a problem, the second will throw an exception when the string is passed to the `Add` method:

```
Dim MyCollection2 As New Collection
MyCollection2.Add(5)
MyCollection2.Add("test")
```

```
Dim MyCollection As New Generic.Collection(Of Integer)
MyCollection.Add(5)
MyCollection.Add("test")
```

To finish the chapter and consolidate the things you've learned, the next Try It Out adds a splash screen and login form to the Personal Organizer application.

Adding the Login Form

1. Start Visual Basic Express and open the Personal Organizer application you've been working on. If you don't have the project up to date to this point, you'll find a version of the files in the `Chapter 08\Personal Organizer Start` folder of the code downloaded from www.wrox.com for this book.

2. Add a new form by selecting Project ⇨ Add Windows Form. Name the form `SplashScreen.vb`. Set the following properties:

- ❑ **Size**—`300, 300`
- ❑ **Start Position**—`CenterScreen`
- ❑ **FormBorderStyle**—`None`

3. In the `Chapter 08` folder of the downloaded code, you'll find an image named `splash.bmp`. Set the `BackgroundImage` property of the form to this image by clicking the ellipsis button and then importing the image as described previously.

4. Add a label to the form and position it in the bottom right-hand corner of the form. This will contain the version number of your application so users know what version they're using. Change the `Anchor` property to `Bottom, Right` so that it will automatically align to the right when the text is changed programmatically. In addition, set its `BackgroundColor` property to `Transparent` (which you'll find in the list of web colors).

5. Double-click the form to create a `Form Load` event handler and set the text of the label to the version number of the application:

```
Label1.Text = My.Application.Info.Version.ToString
```

6. Open the My Project page by double-clicking the entry in the Solution Explorer and select the SplashScreen form for the Splash Screen setting. You're done with the splash screen now, so you can move to the login form.

7. Add another form to the project by selecting Project ⇨ Add Windows Form. Name this form `Login.vb` and click OK to add it.

8. Add three `Labels`, two `TextBoxes`, and two `Buttons` to the form and position them as shown in Figure 8-6. Name the buttons `btnOK` and `btnCancel` and set their `Text` properties appropriately.

Figure 8-6

9. Set the following properties for the form so that it will emulate a proper login screen. The `AcceptButton` property enables users to press Enter, which will emulate the `Click` event for the selected button. `CancelButton` does the same but for the Escape key.

❑ **StartPosition** — `CenterScreen`

❑ **FormBorderStyle** — `FixedSingle`

❑ **AcceptButton** — `btnOK`

❑ **CancelButton** — `btnCancel`

10. Set the following properties on the first `TextBox`:

❑ **Name** — `txtUser`

❑ **ReadOnly** — `True`

Set the following properties on the second `TextBox`:

❑ **Name** — `txtPassword`

❑ **UseSystemPasswordChar** — `True`

11. Add an event handler for the form's `Load` event. Add the following code:

```
Private Sub Login_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    If My.User.IsAuthenticated Then
        txtUser.Text = My.User.Name
        If UserExists(My.User.Name) Then
            txtPassword.Focus()
        Else
            mNewUser = True
            MessageBox.Show("You are new to the system. Please enter your " + _
                "password and it will be saved in the database for future use.")
        End If
    Else
        MessageBox.Show("Sorry, you are not authenticated and cannot use " + _
                "this program.")
        End
    End If
End Sub
```

This ensures that the user is authenticated in Windows and, if so, sets the `Text` property to the user's name. It then calls the `UserExists` function you created in Chapter 7 to determine whether the user exists in the database. You'll need to define the module-level variable `mNewUser` as a Boolean at the top of the form class. This will be used in the `Click` event for the OK button to determine whether it should create the user and password entry or check the password.

12. Because you need to keep track of the ID value for the user who is currently logged in, create a new function in `GeneralFunctions.vb` that returns the ID for a given Name. This uses the same logic as the database functions discussed in Chapter 7, so review the process there if you're not sure of what's going on here:

```
Public Function GetUserID(ByVal UserName As String) As Integer
    Dim CheckUserAdapter As New _PO_DataDataSetTableAdapters.POUserTableAdapter
    Dim CheckUserTable As New _PO_DataDataSet.POUserDataTable

    CheckUserAdapter.Fill(CheckUserTable)
    Dim CheckUserDataView As DataView = CheckUserTable.DefaultView
    CheckUserDataView.RowFilter = "Name = '" + UserName + "'"
```

163

```
      With CheckUserDataView
          If .Count > 0 Then
              Return CType(.Item(0).Item("ID"), Integer)
          Else
              Return -1
          End If
      End With

  End Function
```

13. Add an event handler for the OK button's `Click` event and add this code:

```
Private Sub btnOK_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnOK.Click
    Dim bOKToContinue As Boolean = False
    If mNewUser Then
        CreateUser(txtUser.Text, txtPassword.Text)
        bOKToContinue = True
    Else
        If UserPasswordMatches(txtUser.Text, txtPassword.Text) Then
            bOKToContinue = True
        Else
            MessageBox.Show("Password doesn't match. Re-enter the password")
            txtPassword.Text = vbNullString
            txtPassword.Focus()
        End If
    End If
    If bOKToContinue = True Then
        Dim MainFormObject As New frmMainForm
        MainFormObject.CurrentUserID = GetUserID(txtUser.Text)
        MainFormObject.Show()
        Me.Dispose()
    End If
End Sub
```

At this point, you'll get an error indicator line underneath the `MainFormObject.CurrentUserID` *property. You'll add this property in step 16.*

If the code determines that this is a new user, it calls the `CreateUser` function (created in Chapter 7) and sets the OK flag. If the user already exists in the database, it checks the password against the one stored in the database. If they do not match, it rejects the login attempt and returns the focus to the Password `TextBox`. If they match, then it sets the same OK flag.

Finally, it checks the OK flag; if it is set to true, it shows the main form and closes the login form.

14. The Cancel button's `Click` event handler should halt the program by using the `End` statement. This statement will end the program immediately and should be used only if no resources are open, as is the case here.

15. Return to the My Project page and change the Startup object to `Login`, and the Shutdown mode to `When last form closes`. This will ensure that the application stays active until the user closes the main form.

16. Open `MainForm.vb` in code view to add the variable and property definition to keep track of the user ID. At the top of the class module, add the following code:

```
Private mCurrentUserID As Integer
Public Property CurrentUserID() As Integer
    Get
        Return mCurrentUserID
    End Get
    Set(ByVal value As Integer)
        mCurrentUserID = value
    End Set
End Property
```

17. In Chapter 7, you created routines to handle the saving and updating of Person data. In those routines you used a hardcoded `UserID` value of 1. Now that the `MainForm` keeps track of the current user, you can change those routines to include the ID. The `saveToolStripButton_Click` routine is shown here, but you should also make a similar change in `objPersonalDetails_ButtonClicked`:

```
Private Sub saveToolStripButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles saveToolStripButton.Click
    If objPersonalDetails IsNot Nothing Then
        If objPersonalDetails.AddMode = True Then
            If AddPerson(mCurrentUserID, objPersonalDetails.Person) Then
                MessageBox.Show("Person was added successfully")
                objPersonList = New PersonList

                If objPersonalDetails IsNot Nothing Then
                    pnlMain.Controls.Remove(objPersonalDetails)
                    objPersonalDetails = Nothing
                End If

                pnlMain.Controls.Add(objPersonList)
                objPersonList.Dock = DockStyle.Fill
            Else
                MessageBox.Show("Person was not added successfully")
            End If
        Else
            If UpdatePerson(mCurrentUserID, objPersonalDetails.Person) Then
                MessageBox.Show("Person WAS updated successfully")
            Else
                MessageBox.Show("Person was not updated successfully")
            End If
        End If
    End If
End Sub
```

18. The last thing to do — now that you can determine which user is using the Personal Organizer application — is to control which Person records are shown in the `PersonList` control. Open that control in code view and add a property at the top of the class:

```
Private mUserID As Integer
Public Property UserID() As Integer
    Get
```

```
            Return mUserID
        End Get
        Set(ByVal value As Integer)
            mUserID = value
        End Set
    End Property
```

Locate the `LoadListBox` routine at the bottom of the class code and add an additional check for the `UserID` value before adding each set of Person information to the `ListBox`:

```
 Private Sub LoadListBox()
     Dim PersonListAdapter As New _PO_DataDataSetTableAdapters.PersonTableAdapter
     Dim PersonListTable As New _PO_DataDataSet.PersonDataTable

     PersonListAdapter.Fill(PersonListTable)

     With lstPersons
         .Items.Clear()
         For Each CurrentRow As _PO_DataDataSet.PersonRow In PersonListTable.Rows
             If CurrentRow.POUserID = mUserID Then
                 Dim CurrentPerson As New Person(CurrentRow.NameFirst, _
                     CurrentRow.NameLast)
                 CurrentPerson.ID = CurrentRow.ID
                 .Items.Add(CurrentPerson)
                 .DisplayMember = "DisplayName"
             End If
         Next
     End With
 End Sub
```

19. Return to `MainForm.vb` and locate the `btnShowList_Click` routine that controls when the `PersonList` control is shown. Immediately after you create the `PersonList` object, set the `UserID` property to the module-level variable in `MainForm` that is keeping track of the current user:

```
Private Sub btnShowList_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnShowList.Click
    objPersonList = New PersonList
    objPersonList.UserID = mCurrentUserID

    If objPersonalDetails IsNot Nothing Then
        pnlMain.Controls.Remove(objPersonalDetails)
        objPersonalDetails = Nothing
    End If

    pnlMain.Controls.Add(objPersonList)
    objPersonList.Dock = DockStyle.Fill
End Sub
```

20. You can run the application now and observe the results. First, the splash screen form will be shown for a moment as the application prepares the login form. In the bottom-right corner will be the current version of the application, retrieved from the `My.Application` object.

After a moment, the login form will be displayed, complete with the user name already popu-lated. If this is the first time through, the program will display a message indicating that you

should enter a new password; otherwise, you'll need to enter the password you created the first time.

If the passwords don't match, you'll be presented with an error message, as shown in Figure 8-7, and returned to the login screen. Otherwise, the login screen will close and the main form will be displayed.

Once the form is displayed, you can show the Person list; and it displays only records associated with the user who is currently logged on.
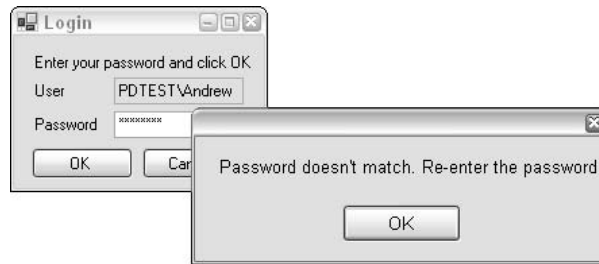
Figure 8-7

Summary

Visual Basic Express is a development tool that gives you more than you could ever imagine to make creating programs easy. The `My` namespace, a part of .NET developed exclusively for Visual Basic programmers, adds to the already impressive number of classes and commands included for your use.

And if that isn't enough, the capability to use code snippets, partial classes, and generics fills the toolkit almost to bursting. With these programming enhancements, you can create applications that serve complex and convoluted purposes easily and without fuss.

In this chapter, you learned to do the following:

❏ Access hard-to-get-to pieces of the system using `My`

❏ Build partial classes for maximum flexibility in reusing code

❏ Create a splash screen and login form for an application

Exercises

1. Use the code snippet library to draw a pie chart on a form. The pie chart snippet can be found by selecting Creating Windows Forms Applications ➪ Drawing.

2. Create a class from two partial classes whereby one defines two variables and the other combines them together.

9

Getting into the World

Perhaps the most obvious difference between Visual Basic Express and its professional counterpart, Visual Basic 2005, is the capability it affords you to create applications for the web. In Visual Basic 2005, programmers have the capability to create websites that use Visual Basic code to control how they appear and what actions are performed when the individual pages should be displayed.

In addition, *web services* — special web applications designed to be used by other programs — can also be implemented using Visual Basic 2005. Visual Basic Express does not allow you to do that. Instead, it has been designed to enable you to make programs that run in a normal Windows environment only.

However, this doesn't mean you can't take advantage of the Internet in your applications — far from it. Visual Basic Express provides numerous ways of accessing the web, and by combining Visual Web Developer Express with the information about the Visual Basic language you've learned in this book, you can design web applications, too.

In this chapter, you learn about the following:

❏   The `WebBrowser` control and how to use it effectively

❏   Implementing web services in your applications to retrieve information from the Internet

❏   Using Visual Web Developer Express to create web applications

Creating a Web Browser

The `WebBrowser` control is like having a scaled-back version of Internet Explorer packaged up and ready for you to use anywhere you would like in your own applications. In Chapter 1 of this book, you created an application that contained a `WebBrowser` control with a simple navigation system.

The advantage of having the `WebBrowser` control is that you can embed Internet pages into your program, rather than have users access the web via their normal web browser. You can customize the control's appearance and control the functionality used, and you can keep track of what users do in the web browser by tracking the various events that are raised when they browse the web.

In the first chapter, you placed a `WebBrowser` control on a form and enabled it to navigate to specified URLs. This is obviously the main function of using the web browser, but there are several other properties and methods that are worth taking a look at.

WebBrowser Properties

When you embed a web browser in your own program, you may want to restrict users from being able to perform certain actions. For example, Internet Explorer provides a rich right-click menu that enables users to view the source of the page, print it, and even open new browser windows. This functionality could enable your users to do things that you don't want them to do. The `WebBrowser` control enables you to disable this menu with a single property — `IsWebBrowserContextMenuEnabled`. Set this property to `False`, either at design time or while the program is running, and the right-click menu will not display.

Two other properties that can control how users interact with the web browser also toggle features:

❑   `AllowWebBrowserDrop` controls whether the web browser control will accept drag and drop actions by the user. An example might be dragging a hyperlink to the browser window. Again, as this allows users to perform actions that may be outside the scope you intend for your program, you can disable it easily by setting the property to `False`.

❑   The `WebBrowserShortcutsEnabled` property enables you to disable keyboard shortcuts that could be used to invoke various commands exposed by Internet Explorer through the `WebBrowser` control, such as Ctrl+N to create a new window and Ctrl+P to print the current page. Again, simply set this property to `False` to disable this functionality.

Several properties are provided to give you feedback about the current state of the internal browser object. `IsOffline` returns a value of True if the user is browsing the web in Offline mode, a special mode of Internet Explorer that can display locally cached pages only. `CanGoBack` and `CanGoForward` let your program know whether there are web page locations in the backward or forward history lists. These properties are used in conjunction with the `GoBack` and `GoForward` methods, which are covered in a moment.

When looking at a typical Internet Explorer window, you'll notice common areas of the interface that provide information to the user. In Figure 9-1, the Internet Explorer window has four areas marked and labeled:

❑   **Area 1**—The title bar of Internet Explorer contains the heading information about the web page that is currently being displayed. This text is accessible through the `DocumentTitle` property of the `WebBrowser` control, which enables you to display the contents somewhere appropriate in your own application.

❑   **Area 2**—The address bar at the top of the Internet Explorer window shows the current URL that is being shown, and enables users to change the web page by entering a new web site address. The `Url` property of the `WebBrowser` emulates both of these features — it returns the currently loaded URL of the `WebBrowser` control and, if changed programmatically, will automatically attempt to navigate to the new location.

❏ **Area 3** — Internet Explorer displays the actual content of the web page in this area. You can retrieve this information in your program through the `Document` and `DocumentText` properties. `DocumentText` is a `String` property that returns the entire web page as a string of text, including the HTML tags and attributes. It's useful for storing the HTML for later use. `Document` returns an `HTMLDocument` object that is then used to process the content of the web page itself.

Area 4

Area 3

Area 2   Area 1
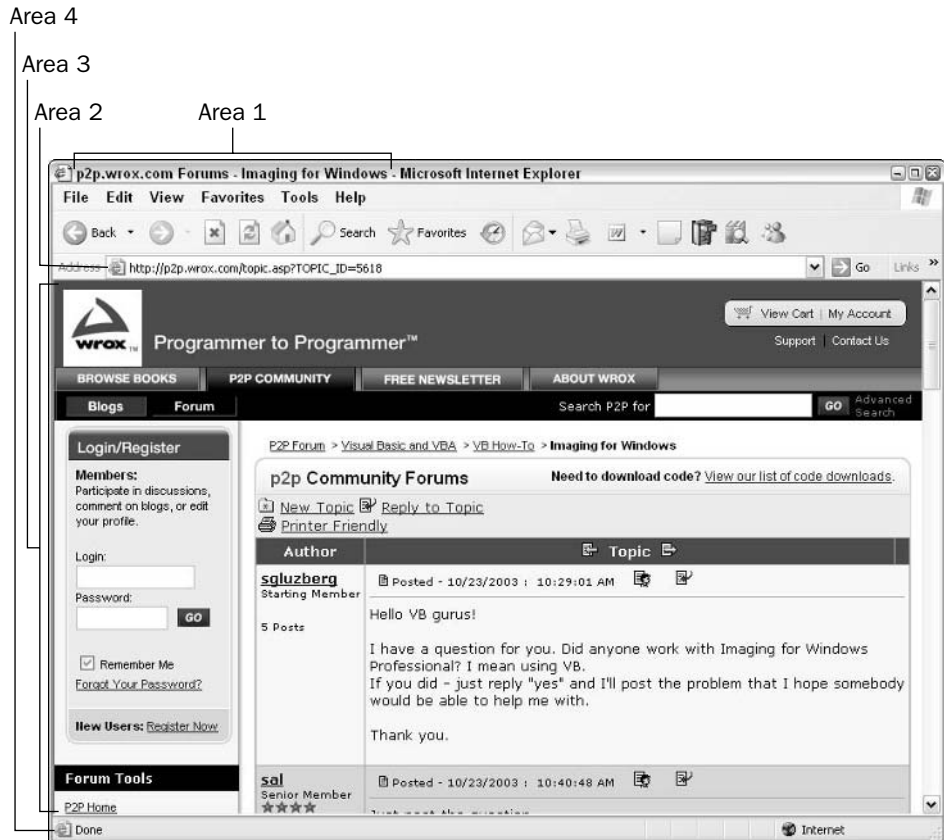
Figure 9-1

❏ **Area 4** — One valuable aspect of browsing the web with a browser such as Internet Explorer is the feedback you are provided as the page loads. The status bar is constantly updated with information about the page being loaded and displayed. `StatusText` is a String property in the `WebBrowser` control that enables your program to retrieve that information and display it yourself.

WebBrowser Methods

The methods exposed by the `WebBrowser` control give you programmatic access to the common actions that can be performed in Internet Explorer. In Chapter 1 you used the `Navigate` method to tell the web browser object to load a particular URL. The `Navigate` method is actually overloaded, which means that there is more than one way of calling it. In this case, `Navigate` provides three different functions:

❑ `Navigate(URL)` — Tells the `WebBrowser` control to load the page located at the specified URL.

❑ `Navigate(URL, TargetFrame)` — Does the same as the default `Navigate` method but specifies a section of the currently loaded page to contain the results of the navigation. This can be useful if you know how the HTML document is structured internally and you want to populate only certain sections of the page with the new information.

❑ `Navigate(URL, NewWindow)` — This version of `Navigate` is likely to be the least used. It starts up a new instance of Internet Explorer and loads the page in that instead of your own application's web browser object.

The other methods of the `WebBrowser` you've used already are the `GoHome` and `GoBack` functions in the exercises at the end of Chapter 1. `GoHome` tells the browser object to navigate to the default home URL for Internet Explorer, while `GoBack` navigates back one page to the previous page the user was viewing.

Coupled with these two methods are `GoForward` and `GoSearch`. `GoForward` will navigate forward through cached pages in the forward history list. `GoSearch` will open the default search page as specified in the options for Internet Explorer.

Besides these navigation controls, three main functions are commonly required: `Refresh`, `Stop`, and `Print`. These are all self-explanatory and emulate their corresponding Internet Explorer toolbar buttons. Because these methods are available, you can easily build a functional web browser into your application with very little code required.

Other methods worthy of a mention are the `Show...Dialog` functions. These five methods each show a commonly used dialog window within Internet Explorer, giving you the capability to show the dialogs in your own program:

❑ `ShowPageSetupDialog` — Brings up the Page Setup dialog, enabling the user to customize how a page should be printed.

❑ `ShowPrintDialog` — If you invoke the Print method, it will send the currently loaded web page to the default printer using the default settings. Using the Print Dialog, the user can customize where and how the page should be printed and specify how many copies are wanted.

❑ `ShowPrintPreviewDialog` — Yes, you can provide full print preview functionality of the web page simply by calling this method. Be aware, however, that users will be able to run any of the commands in the Preview dialog, such as Page Setup and Print.

❑ `ShowPropertiesDialog` — This method brings up the Properties dialog that provides information about the currently loaded page.

❑ `ShowSaveAsDialog` — This gives users the capability to save the web page with one simple method call.

WebBrowser Events

The `WebBrowser` control also raises several events that your application can intercept and handle. This capability to determine when things have occurred within the browser object, along with the methods and properties that the control gives you, provides enormous scope to control how the web browser is displayed within your application and how the rest of your program reacts based on its content.

While many events could be handled, a handful are important enough to be covered here. First and foremost are the `Navigating` and `Navigated` events. `Navigated` is fired when a page is found and begins to be loaded into the browser object. At this point, you can start using the `Document` properties discussed earlier to interrogate the content of the new page.

The `Navigating` event is raised by the `WebBrowser` control when the browser object is about to load a new page. You can use this event to cancel unwanted page loads, and it is often used to restrict web browser functionality to only a set of allowable pages and URLs.

While `Navigated` indicates that the `Document` properties now refer to the newly loaded page, it's not until the `DocumentCompleted` event is raised that you can be confident that the entire contents have been downloaded. The following table illustrates the normal order of these three events as they occur when the user clicks a link within the browser.

WebBrowser Actions	Program Impact
`Navigating` event is fired.	Program can cancel the navigation.
`WebBrowser` attempts to locate and begin to load the page. If successful, `Navigated` is fired.	Document properties can be used to determine the state of the page load.
Once the page is finished loading, the `DocumentCompleted` event is fired.	Document properties now contain the fully loaded HTML page.

Besides these main events, you may also want to handle several events that inform your application when information has been altered. The `StatusTextChanged` event is raised whenever the internal browser object has a different status. This is the event that Internet Explorer uses to determine when to update its status bar, and you can do the same thing in your own application.

The `DocumentTitleChanged` event is used for a similar purpose — this time it's the text to be displayed in the title bar area of Internet Explorer that has changed, with the `DocumentTitle` property interrogated to determine the new value.

`ProgressChanged` is an event that can be useful for your application's handling of the web browser's loading state. The event includes an estimate of the loading document's total number of bytes, along with how many bytes have been downloaded so far. This enables your program to include some sort of progress indicator to inform users about how much of the page loading process has been completed.

The Personal Organizer application you've been building throughout this book currently does not have any Internet capabilities. Later in this chapter you'll use web services to gather information from Amazon.com, but what would make another nice feature in the program is the capability to browse certain web sites from within the program itself.

In the next Try It Out, you'll create a new user control that encapsulates the `WebBrowser` control, along with a select number of buttons, and add code to the `MainForm.vb` file of the Personal Organizer application to show this control when the user requests it.

Try It Out   Creating a Custom Web Browser Control

1. Start Visual Basic Express and open the Personal Organizer solution you've been working with. If you have not completed the previous chapter's exercises, you will find an up-to-date solution in the `Chapter 09\Personal Organizer Start` folder of the code download you can get on `www.wrox.com`.

2. Create a new user control by selecting Project ⇨ Add User Control. Name the control `POWebBrowser.vb` and click OK to add it to your solution.

3. Open the new control in Design view and first add a `ToolStrip` control, followed by a `WebBrowser` control to the design surface. Adding them in this order will automatically dock the `ToolStrip` to the top of the control's area and fill the remaining space with the `WebBrowser`.

Set the following properties for the `WebBrowser` control:

- ❑ `Name — MyWebBrowser`
- ❑ `IsWebBrowserContextMenuEnabled — False`
- ❑ `AllowWebBrowserDrop — False`
- ❑ `Url — C:\PersonalFavorites.html`
- ❑ `WebBrowserShortcutsEnabled — False`

Set the `GripStyle` property of the `ToolStrip` to `Hidden`, and add six buttons to the strip with the following Text properties (you can use the Items collection editor to set the Text properties):

- ❑ `&Back`
- ❑ `&Forward`
- ❑ `&Home`
- ❑ `&Stop`
- ❑ `&Refresh`
- ❑ `&Close`

The ampersand (`&`) symbol will be automatically translated into a keyboard menu shortcut. Therefore, when the user holds down the Alt key and presses B, the program will emulate the Back button being clicked. It will also display a line underneath the letter that identifies the shortcut so the user is aware of the keyboard shortcut. The user interface of the User Control should appear similar to Figure 9-2.
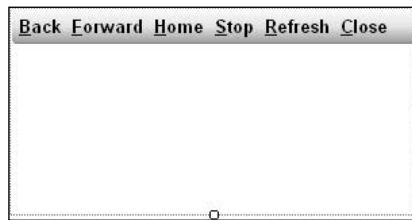
Figure 9-2

4. Now you'll implement the basic navigation functionality of your web browser. Add the following line of code to the `Click` event of the Back button (remember to double-click the button in Design view and Visual Basic Express will automatically create a subroutine that will handle the `Click` event for you):

```
Private Sub BackToolStripButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles BackToolStripButton.Click
    MyWebBrowser.GoBack()
End Sub
```

It's always a good idea to check whether a function can be performed, so the `CanGoBack` property is checked first to determine whether there are pages in the back history. Because it's a Boolean, you can omit the `= True`, which results in code that reads almost like regular English. If there are pages in the history, then the `GoBack` method of the `WebBrowser` control is called:

```
Private Sub BackToolStripButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles BackToolStripButton.Click
    If MyWebBrowser.CanGoBack Then MyWebBrowser.GoBack()
End Sub
```

5. Repeat this process for the Forward, Stop, and Refresh buttons (you don't need to do the checks for Stop and Refresh):

```
Private Sub ForwardToolStripButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles ForwardToolStripButton.Click
    If MyWebBrowser.CanGoForward Then MyWebBrowser.GoForward()
End Sub

Private Sub StopToolStripButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles StopToolStripButton.Click
    MyWebBrowser.Stop()
End Sub

Private Sub RefreshToolStripButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles RefreshToolStripButton.Click
    MyWebBrowser.Refresh()
End Sub
```

6. The Home button is the first of two special cases. If you simply implemented the `GoHome` method, users would go to their default home page found in the options of Internet Explorer. Because you want to retain control over what is displayed in your program, use the `Navigate` method instead to load your default page:

```
Private Sub HomeToolStripButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles HomeToolStripButton.Click
    MyWebBrowser.Navigate("C:\PersonalFavorites.html")
End Sub
```

You'll notice that the location both in the `Url` property of the `WebBrowser` and in the `Navigate` method here is specified as `C:\PersonalFavorites.html`. This is a very simple web page created for this application that contains several commonly used websites. The HTML page can be found in the `Chapter 09` folder of the code download for this book. If you choose to keep this page in a different location, make sure you change it in both places.

7. The last button — `Close` — will be used to tell the application that the user would like to close the web browser window. To achieve this, you first need to create an event for the user control. As explained in Chapter 6, adding events to your own controls is achieved by first defining the signature of the event at the top of your user control code, and then by telling Visual Basic to raise the event through the `RaiseEvent` command. Define the event at the top of your code as the first line within the class definition:

```
Public Event CloseRequested()
```

Then, in the Close button click, raise the event like so:

```
Private Sub CloseToolStripButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles CloseToolStripButton.Click
    RaiseEvent CloseRequested()
End Sub
```

8. Save and build your entire application to confirm that everything compiles. This will also compile the user control so that it can be used by the main form.

9. Return to the Design view of `MainForm.vb` and add a new button underneath the other two already there. Set the button's properties as follows:

❑   Name — `btnWeb`

❑   Text — Web

At this point, your main form's interface should now look like the one shown in Figure 9-3.
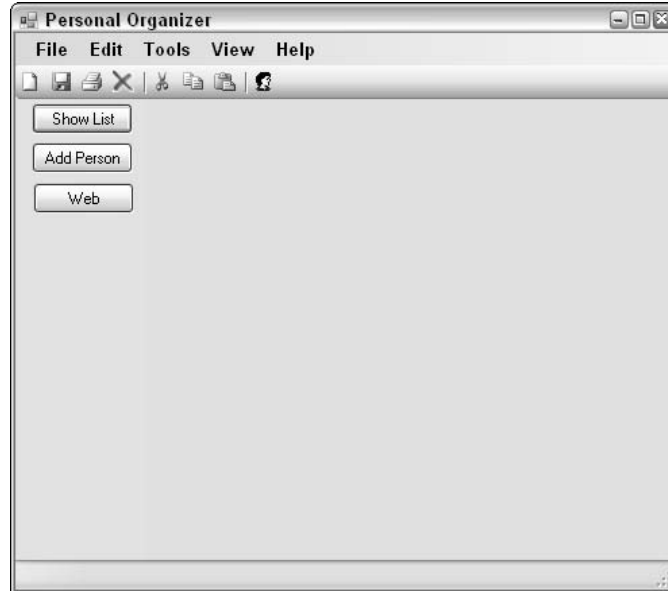
Figure 9-3

10. Double-click the button to create the `Click` event handler routine. At this point, you need to implement code similar to that in Chapter 5 for the other two user controls. This time, however, you need to check for the existence of two, rather than just one:

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

        objPOWebBrowser = New POWebBrowser

        If objPersonalDetails IsNot Nothing Then
            pnlMain.Controls.Remove(objPersonalDetails)
            objPersonalDetails = Nothing
        End If
        If objPersonList IsNot Nothing Then
            pnlMain.Controls.Remove(objPersonList)
            objPersonList = Nothing
        End If

        pnlMain.Controls.Add(objPOWebBrowser)

        objPOWebBrowser.Dock = DockStyle.Fill

End Sub
```

11. Now define the `objPOWebBrowser` variable at the top of the `MainForm` code, directly underneath the definition of the other two user control objects:

```
Private objPersonList As PersonList
Private objPersonalDetails As PersonalDetails
Private objPOWebBrowser As POWebBrowser
```

12. To clean up the other code, you should also check for the existence of the `POWebBrowser` control before showing the `PersonalDetails` or `PersonList` controls. Add the following code in the `btnShowList_Click` and `btnAddPerson_Click` routines immediately before you add the control to the Panel:

```
        If objPOWebBrowser IsNot Nothing Then
            pnlMain.Controls.Remove(objPOWebBrowser)
            objPOWebBrowser = Nothing
        End If
```

13. Run the application and click the Web button to show the web browser user control. Note how it behaves in a similar way to your other controls, filling the available area. Try out the links in the loaded page, as well as the various buttons. The only one that is not working at this point is the Close button. This is because even though you are correctly raising the event, the main form is not handling it. Terminate the application and return to the code view of `MainForm.vb`.

14. At this point, even though the `POWebBrowser` object has been defined and does have events, the `MainForm` code cannot intercept the events themselves. This is because the definition of the user control object did not specify that events are associated with the control. To confirm that this is the case, open the Class Name drop-down list at the top of the code window. Scrolling through the list, you'll notice that `objPOWebBrowser` is not present.

The `WithEvents` keyword is used to tell Visual Basic Express that the object will have events that the application needs to be able to handle. If you do not include this keyword, your program will not be able to receive any of the events, even though they might be raised by the object. You'll also notice that the IntelliSense of Visual Basic Express will display only objects that have events. Change the definition of `objPOWebBrowser` to include the `WithEvents` keyword:

```
Private WithEvents objPOWebBrowser As POWebBrowser
```

15. Now you want to intercept the event you created earlier — `CloseRequested`. In the Class Name drop-down list, find and select `objPOWebBrowser`. Then, in the Method Name drop-down, scroll down to `CloseRequested` and select it from the list. Visual Basic Express will automatically create an event handler subroutine to handle the `CloseRequested` event. You can copy and paste the code used to determine whether the web browser control exists, and, if so, destroy it. Your final subroutine should look like this:

```
Private Sub objPOWebBrowser_CloseRequested() Handles _
    objPOWebBrowser.CloseRequested
    If objPOWebBrowser IsNot Nothing Then
        pnlMain.Controls.Remove(objPOWebBrowser)
        objPOWebBrowser = Nothing
    End If
End Sub
```

16. Another thing you may have noticed when you ran the application in step 13 is that the Back and Forward buttons are always enabled. It would be nice to disable these buttons when they cannot be used, similar to the way Internet Explorer does with its own Back and Forward buttons. Return to the code view of the `POWebBrowser` control. Add an event handler routine for the `Navigated` event of the `MyWebBrowser` object (refer to step 14 for finding the event). In the subroutine, add the following code:

```
Private Sub MyWebBrowser_Navigated(ByVal sender As Object, _
    ByVal e As System.Windows.Forms.WebBrowserNavigatedEventArgs) Handles _
    MyWebBrowser.Navigated
    BackToolStripButton.Enabled = MyWebBrowser.CanGoBack
    ForwardToolStripButton.Enabled = MyWebBrowser.CanGoForward
End Sub
```

Now whenever the `WebBrowser` control navigates to a new page, the code will check the `CanGoBack` and `CanGoForward` properties. Because they are both Boolean properties, like the Back and Forward buttons' `Enabled` properties, you can simply assign one to the other. As a result, if the `CanGoBack` property returns `True`, then the Back button will be enabled. If `CanGoBack` is `False`, then the Back button will be grayed out and users cannot click it. The same is true for the Forward button and `CanGoForward`.

17. Run the application again; click the Web button to bring up the web browser. Navigate through the pages and note how the Back and Forward buttons are enabled only when there are pages in the back and forward history lists (see Figure 9-4). When you're ready, click the Close button and note how the main form now handles the event and closes the browser.

You've now created a robust web browser in your Personal Organizer application. Customize the `PersonalFavorites.html` file to contain your own commonly visited sites so you can browse them without having to open up Internet Explorer.

Figure 9-4

Web Services

A *web service* is a specialized kind of program that is designed to run on the Internet with other applications calling its functions. You can think of a web service as a kind of remote object complete with publicly available methods that other applications can call to retrieve information or invoke specific actions.

Visual Basic Express provides support for web services based on the open standard protocols of Extensible Markup Language (XML) and Simple Object Access Protocol (SOAP). In Chapter 12, you'll learn all about XML and how it can be used to store and format data, but for now all you need to know is that XML is used to format information that is passed to a web service, and to define the structure of the data returned to the calling application.

SOAP is a communications protocol that can wrap a message into a standard structure that can then be passed over the Internet to the web service. The web service can then unwrap the XML defining the particular request, process it internally, and generate XML for a response. This response object is then returned, again via SOAP, to the calling program. This *request and response* system is a fundamental communications method used in many different Internet communications systems.

Many other languages can use XML web services but they often must use other communications protocols to call them, such as HTTP POST or REST. These methods involves constructing a URL from the web service location, adding the required parameters using standard URL addressing constructors, and then invoking the final URL.

Visual Basic Express makes using web services (called *consuming* web services) a much more straightforward process by giving you the capability to add the location of the web service in the IDE and then create an object representation of the web service methods for use in your program. Once the web service location has been added to your project, you can create and use objects based on it in much the same way as any other objects.

Visual Basic Express does not allow you to create your own web services. Instead, you need to use Visual Studio 2005 or Visual Web Developer Express to create customized web services. However, that shouldn't stop you, as several websites provide directories of publicly available web services.

These directories also follow a specific standard — UDDI. UDDI, which stands for *Universal Description, Discovery, and Integration,* enables businesses to register their web services in a central location, often categorized into groups of similar services. Other businesses and individuals can browse through these directories looking for a service that meets their needs.

Many UDDI libraries are available, although some share information, so you can usually find the same web service listed in multiple directories. Microsoft (`uddi.microsoft.com`) and IBM (`uddi.ibm.com`) both provide detailed lists of web services that can be used by your Visual Basic Express programs, but smaller, specialized web directories can sometimes provide an easier navigation system to find what you need. For example, Microsoft's UDDI library first requires you to choose a categorization scheme, none of them very clear, to browse the directory listings.

Conversely, a website such as BindingPoint (`www.bindingpoint.com`) takes you directly to a simple category listing which is straightforward to navigate to find the web service that best suits your needs.

To add a web service to your Visual Basic Express program, you use the Project ➪ Add Web Reference menu command, or right-click on the project in the Solution Explorer and choose Add Web Reference. Both will present you with the Add Web Reference Wizard, as shown in Figure 9-5.

If you know the location of the web service, you can enter it directly in the URL text field; if you have not located one yet, you can use the various browsing options provided. Once you have located the web service you would like to add to your project, the display pane in the wizard will show you the list of methods that are available in the web service. Click the Add Reference button to add the web service definition to the project.

> You can set the name of the web reference at this point, but you can also change it later through the Properties window.

As mentioned, using a web service in your code is similar to using any other class. You must first define an instance of the web service class you would like to use, and then you set the properties and invoke the methods that you need. Usage of a simple web service might look like this:

```
Dim myWebServiceFunction As New WSName.WSClass
myWebServiceFunction.SomeMethod
```
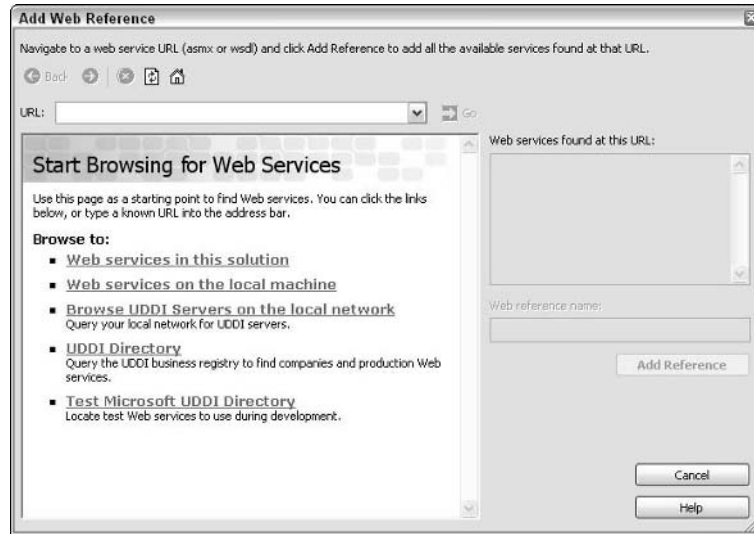
Figure 9-5

In the following Try It Out, you'll create a simple application that uses a web service to calculate the difference between two dates. While the method output is simple, it illustrates the way a web service can be used to extend your application's functionality.

Try It Out   Consuming a Web Service

1. Rather than browse through the often confusing UDDI that Microsoft has provided, you'll search through a specialized web service directory to find the appropriate web service. Start your web browser and navigate to www.bindingpoint.com. In the Categories listing, locate the Calendar group and click the link.

2. Lucky for you! The first *web method* (functions exposed by a web service are called web methods, or *methods* for short) in the Calendar category is called Date Difference and is described as a method for calculating the difference between two dates — perfect for our purposes. Click on the title to get the technical information about the web service and locate the Access Point URL.

   The Access Point URL is where the web service itself resides. In this case, it's www.vinsurance .com/datedifference/datedifference.asmx.

3. Start Visual Basic Express and create a new Windows Application project. Name it TestWebService.

4. Use the Project ⇨ Add Web Reference menu command to bring up the Add Web Reference Wizard. In the URL field, enter the Access Point URL you found in step 3 and click the Go button to instruct the wizard to download the web service information.

   After a moment, the main description pane will be populated with a list of methods available in this web service. As you can see in Figure 9-6, there is one method, DateDifference. Change the Web Reference name to WSDateDiff and click Add Reference to add it to your project.

5. Open the form in Design view and add a `Button`, a `TextBox` and two `DateTimePicker` controls. Change the `Text` property of the `Button` to `Calculate Difference` and double-click it to have Visual Basic Express automatically create the event handler routine for you.

6. In the `Click` event handler, you want to create an instance of the web service, pass the values of the two `DateTimePicker` controls to the `DateDifference` method, and assign the return value to the `Text` property of the `TextBox` so users can see the answer:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    Dim myDateDiff As New WSDateDiff.DateDifferenceService
    TextBox1.Text = myDateDiff.DateDifference(DateTimePicker1.Value, _
        DateTimePicker2.Value)
End Sub
```
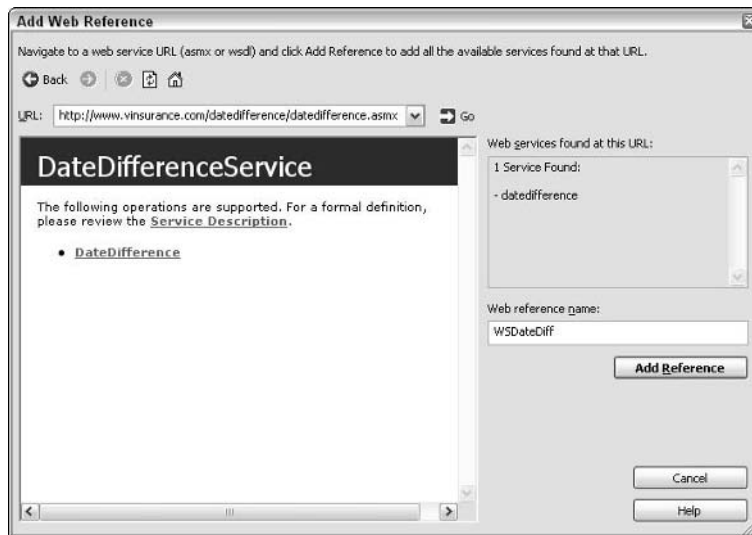

Figure 9-6

As you can see from this simple example, creating the web service object is the same as any other object. You define a variable as the web service type and then instantiate it as a new object. Using the IntelliSense provided by Visual Basic Express, when you add the period of the name of your web service object, you'll be presented with a list of available methods. The only one you need worry about is the `DateDifference` method. Likewise, IntelliSense will tell you what parameters the method is expecting — much easier than trying to create a URL with the correctly formatted information.

7. Amazingly, that's all you need to do. Run the application, choose two different dates, and click the Calculate Difference button. After a few moments, the text box will be populated with a value representing the number of days between the two dates you chose. It takes a few seconds because Visual Basic needs to access the Internet, find the web service, pass the information over in a SOAP-formatted envelope containing an XML representation of the data, wait for the response, and then process the results into a string ready for the text box to use.

Commercial Web Services

Web services are not restricted to simple calculations. Companies use web services to provide functions to their clients or employees so that their data can be processed. Many companies use web services in conjunction with the data driving their web sites to dynamically update the content of the site's pages. Other corporations offer strictly regulated services to business partners to pass financial data back and forth.

With web services growing in popularity, many organizations are starting to provide complex web services so that developers can easily incorporate the information into their own applications. Government organizations can provide complex functionality, such as registering and retrieving company registration information, or simple functionality, such as returning the locations included in a specified postal code.

Large service-oriented companies are also providing their information as a service (or many services). Websites such as Amazon, Google, and eBay all provide developers with access to their data through web services, and often allow the application to interact with the processes found on their website. For example, eBay's developer kit allows not only the retrieval of auction descriptions, but also the capability to add and modify listings to be published on their site. Amazon's web service provides the capability to add items to a user's cart as well as return all kinds of search results.

To control the usage of their web services, companies like these require each developer to register in a development program and pass unique identifying keys with each call to their web service methods. However, once that registration process has been performed, often their web services can be used just like any other publicly available web service. In the following Try It Out, you'll register for Amazon's web service program so that you can use their web services in the Personal Organizer application you've been creating throughout this book.

Try It Out   Web Service Registration

1. Open your web browser and navigate to the main Amazon web page—`www.amazon.com`.

2. Scroll down the page. On the left-hand side, you should find a Make Money box with a link labeled Web Services (see Figure 9-7). Clicking this link will take you to the main Web Services area of the Amazon website. While you can download the various samples and the documentation to help you create applications with the Amazon web service, you won't be able to use them until you complete your registration and receive your Subscription ID.

3. Click on the Register for AWS link at the top-left corner of this main page and you will be prompted to sign in to your Amazon account. You'll need to register as an Amazon member if you have not previously done so, but if you regularly use Amazon to purchase items online, you can use your existing membership. Once you've successfully logged in, you'll be presented with a simple form to fill out that is used to identify you, along with the license agreement you must accept to be able to use Amazon web services. Click the Continue button to confirm the registration.

    You should review the limitations you will be under if you accept the license agreement, although they are very generous considering it's a free service.

4. After a moment you will be presented with a confirmation screen, and you will receive an e-mail message containing the Subscription ID you will need to use the web service.

    In every Amazon web service function call that you write in your code, you will need to include this Subscription ID value to identify yourself to Amazon. Apart from that, you're done.

183

Figure 9-7

The Amazon web service is much more complicated than the Date Difference web service described earlier in this chapter. It comes with many methods and custom-built complex class structures that provide you with the information returned from a call to the web service itself.

It is beyond the scope of this book to detail the many functions you can perform using the Amazon web service, but I encourage you to read through the documentation that is included as part of the program to familiarize yourself with the various methods and objects you can use. For this chapter, you only need to understand the `ItemSearch` method and how it can be used to find items within a particular search index that meet a simple set of criteria.

> For more in-depth information about this topic, check out Denise Gosnell's good treatment of the subject in Professional Web APIs: Google, eBay, Amazon.com, MapPoint, FedEx (Wrox 2005).

Amazon's ItemSearch

Using `ItemSearch`, you can search through Amazon's many different databases looking for items that meet various criteria. This could be author names, musical group details, manufacturer information, or a more generic set of keywords.

You'll find detailed information about `ItemSearch` in the online documentation. At the main Web Services page on Amazon's site, find the link to the left that is labeled Documentation. From the Documentation page, you'll find links to currently supported versions of their web services. At this point, the documentation can be read online or downloaded in Adobe Acrobat format (PDF).

The details about `ItemSearch` can be found in the Operations section of the API Reference. The documentation contains samples as well as detailed descriptions about each parameter required for the call. Reviewing this list shows that the only required fields are the `Operation` and the `SearchIndex` parameters. The first one, `Operation`, simply identifies this particular method to the application program interface (API), and, as you'll see in a moment, is embedded in the call to the web service, so the only field you need to populate in code to make the `ItemSearch` method call successful is the `SearchIndex` to tell the web service which database to look in.

In Visual Basic Express, the way in which the web service is called differs from the way the documentation describes it. Rather than the `ItemSearch` method simply accepting one request containing `SearchIndex` and any other optional parameters to refine the query, it takes an `ItemSearch` object that can contain a collection of these requests, conveniently called `Request`.

For each request you want to make of the web service, you create an `ItemSearchRequest` object, populate it with the required parameters, and add it to the `ItemSearch`'s `Request` property. Once you have set up the requests, you then need to invoke the `AWSECommerceService`'s `ItemSearch` method, passing in the `ItemSearch` object and assigning the response from the web service to an `ItemSearchResponse` object. Putting all of this together, the program might flow like this:

1. Create an `AWSECommerceService` object.

2. Create an `ItemSearch` object.

3. Create a collection of `ItemSearchRequest` objects and populate each one with parameters, including the required `SearchIndex`.

4. Call the `ItemSearch` method of the `AWSECommerceService` object, passing the `ItemSearch` object created in step 2.

5. Assign the return of the web service method call to an `ItemSearchResponse` object.

6. Process the contents of the `ItemSearchResponse` object to determine the results of the search attempt.

The main component of the `ItemSearchResponse` object is a collection of items that met the search criteria (assuming the search worked). You could process this collection as is, or build a dataset from the results and populate databound controls with the dataset contents.

In the following Try It Out, you'll create a method to retrieve suggested gift ideas for a person in your Personal Organizer database based on their likes and the category you've chosen. This will demonstrate how easy it is to use even the most complex web services in Visual Basic Express.

Try It Out   Adding "Suggested Gift Ideas"

1. Open the Personal Organizer solution you have been working on. If you have not completed the previous Try It Out in this chapter and would like to continue from where it ended, in the downloaded code for this book (available at `www.wrox.com`) you'll find a project in the `Chapter 09\Personal Organizer Gift Idea Start` folder that contains everything up to this point.

2. You're going to create a new form that will retrieve information based on the categories and favorite things information that you added to the Person database table in Chapter 3. This form will be accessible from the `PersonDetails` control and should send back information to the control about selected items.

3. Add a new form to the project with the Project ⇨ Add Windows Form menu command and call it `GetGiftIdeas.vb`. To this form you'll need to add a number of items that will control how the Amazon web service will be called. The `PersonDetails` control will pass to the form the value contained in the Favorites text field, along with the six category Boolean flags. Add a `TextBox` and six `RadioButton` controls to the form. To make the user interface a bit cleaner, you can use a `GroupBox` control to contain the `RadioButtons`. You may also want to add a descriptive label next to the `TextBox` so users can determine what it contains.

Finally, add three `Buttons` for the various actions that will be available, and a `CheckedListBox` in which the results can be displayed.

4. Set the properties of the controls you added in the previous step as follows:

- ❑ `TextBox Name` — `txtFavorites`
- ❑ `TextBox ReadOnly` — `True`
- ❑ `TextBox Anchor` — `Top, Right, Left`
- ❑ `Button1 Name` — `btnSearch`
- ❑ `Button1 Text` — `Search`
- ❑ `Button1 Anchor` — `Bottom, Left`
- ❑ `Button2 Name` — `btnCancel`
- ❑ `Button2 Text` — `Cancel`
- ❑ `Button2 Anchor` — `Bottom, Right`
- ❑ `Button3 Name` — `btnSave`
- ❑ `Button3 Text` — `Save`
- ❑ `Button3 Anchor` — `Bottom, Right`
- ❑ `CheckedListBox Name` — `clbResults`
- ❑ `CheckedListBox Anchor` — `Top, Bottom, Left, Right`
- ❑ `RadioButton1 Name` — `radBooks`
- ❑ `RadioButton1 Text` — `Books`
- ❑ `RadioButton2 Name` — `radVideos`
- ❑ `RadioButton2 Text` — `Videos`
- ❑ `RadioButton3 Name` — `radMusic`
- ❑ `RadioButton3 Text` — `Music`
- ❑ `RadioButton4 Name` — `radToys`
- ❑ `RadioButton4 Text` — `Toys`
- ❑ `RadioButton5 Name` — `radVideoGames`
- ❑ `RadioButton5 Text` — `Video Games`
- ❑ `RadioButton6 Name` — `radApparel`
- ❑ `RadioButton6 Text` – `Apparel`

When you're done, the form's layout should look similar to what is shown in Figure 9-8. Note that the RadioButtons have been grouped in a GroupBox control with a title of Type of search.
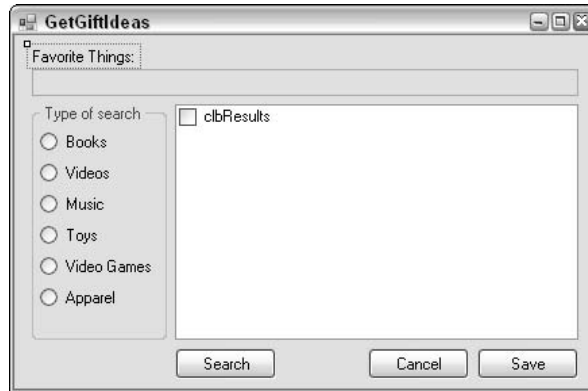
Figure 9-8

5. When the PersonDetails control shows this new form, it will need to pass the information about the selected Person to it. To do that, you'll need to create several properties that will be accessible from outside the form, and when the form loads, populate and set the various form components from this data.

    Go to the code view of the GetGiftIdeas.vb form and add properties for a String variable to store the Favorites, another String to keep track of who the gifts are for, and six Boolean flags to indicate the preferred categories of the person being processed. Note that creating a property for a form is done the same way as creating properties for other classes and user controls, as described in Chapter 6.

6. Define the module-level variables that will store the data:

```
Private msDisplayName As String
Private msFavorites As String
Private mbCategoryBooks As Boolean
Private mbCategoryVideos As Boolean
Private mbCategoryMusic As Boolean
Private mbCategoryToys As Boolean
Private mbCategoryVideoGames As Boolean
Private mbCategoryApparel As Boolean
```

7. Create a write-only property for each one. Write-only properties do not need to be passed back to the part of the program that is using the object, and they are useful for initializing information in the object, such as what you're going to do. Each property definition will look like this:

```
Public WriteOnly Property DisplayName() As String
    Set(ByVal value As String)
        msDisplayName = value
    End Set
End Property
```

8. To customize the form with the information passed over, add code to the form's `Load` event. The title bar of the form should be set to include the name to be displayed, while the `TextBox` will have the `msFavorites` variable assigned to its `Text` property:

```
Private Sub GetGiftIdeas_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    Me.Text = "Get gift ideas for " & msDisplayName

    txtFavorites.Text = msFavorites
End Sub
```

To highlight the preferred categories for the selected person's details, you'll change the text color of the `RadioButtons` that correspond with their categories to red, and you will set the first preferred category `RadioButton`'s `Checked` property so that it is selected by default. To do this, add a conditional logic block that checks the module-level Boolean, as shown here:

```
If mbCategoryBooks = True Then
    radBooks.ForeColor = Color.Red
    radBooks.Checked = True
End If
```

Repeat this block of code for each category, making sure you put them in reverse order. This will enable the `Checked` property of the `RadioButtons` to be set properly. Because only one `RadioButton` in a group can be selected at any one time, setting the `Checked` property of any one button to `True` resets all of the others to `False`. The final subroutine should look like this:

```
Private Sub GetGiftIdeas_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    Me.Text = "Get gift ideas for " & msDisplayName

    txtFavorites.Text = msFavorites

    If mbCategoryApparel = True Then
        radApparel.ForeColor = Color.Red
        radApparel.Checked = True
    End If
    If mbCategoryVideoGames = True Then
        radVideoGames.ForeColor = Color.Red
        radVideoGames.Checked = True
    End If
    If mbCategoryToys = True Then
        radToys.ForeColor = Color.Red
        radToys.Checked = True
    End If
    If mbCategoryMusic = True Then
        radMusic.ForeColor = Color.Red
        radMusic.Checked = True
    End If
    If mbCategoryVideos = True Then
        radVideos.ForeColor = Color.Red
        radVideos.Checked = True
    End If
    If mbCategoryBooks = True Then
        radBooks.ForeColor = Color.Red
        radBooks.Checked = True
    End If
End Sub
```

9. To confirm that this works as expected, you will add a button to the `PersonDetails` control that will create an instance of this new form, populate the properties with information, and then show the form. Open the `PersonDetails` control in Design view and add a `Button` control next to the Category checkboxes. Set its name to `btnGetGiftIdeas` and its text to `Get Gift Ideas` so it looks like what is shown in Figure 9-9.

Figure 9-9

10. Double-click the button to create an event handler routine for its `Click` event and open the code view. You'll need to create an instance of the form just as you would for any other object:

```
Private Sub btnGetGiftIdeas_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnGetGiftIdeas.Click
    Dim frmGetGiftIdeas As New GetGiftIdeas
End Sub
```

Then, using a `With` block to shortcut the setting of multiple properties (as outlined in Chapter 6), set the public properties of the `GetGiftIdeas` form with the values of the `PersonalDetails` control components:

```
Private Sub btnGetGiftIdeas_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnGetGiftIdeas.Click
    Dim frmGetGiftIdeas As New GetGiftIdeas
    With frmGetGiftIdeas
        .DisplayName = txtFirstName.Text + " " + txtLastName.Text
        .Favorites = txtFavorites.Text
        .CategoryBooks = chkBooks.Checked
        .CategoryVideos = chkVideos.Checked
        .CategoryMusic = chkMusic.Checked
        .CategoryToys = chkToys.Checked
        .CategoryVideoGames = chkVideoGames.Checked
        .CategoryApparel = chkApparel.Checked
    End With
End Sub
```

The final requirement is to display the form. Because the `GetGiftIdeas` form should be closed properly without returning to the main part of the program, use the `ShowDialog` method to force it to always be on top. The final subroutine should appear as follows:

```
Private Sub btnGetGiftIdeas_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnGetGiftIdeas.Click
    Dim frmGetGiftIdeas As New GetGiftIdeas
    With frmGetGiftIdeas
        .DisplayName = txtFirstName.Text + " " + txtLastName.Text
        .Favorites = txtFavorites.Text
        .CategoryBooks = chkBooks.Checked
        .CategoryVideos = chkVideos.Checked
        .CategoryMusic = chkMusic.Checked
        .CategoryToys = chkToys.Checked
        .CategoryVideoGames = chkVideoGames.Checked
        .CategoryApparel = chkApparel.Checked
    End With
    frmGetGiftIdeas.ShowDialog()
End Sub
```

11. Run the application, select a person from the list, and view his or her details. Once the `PersonDetails` control is displayed and populated with the person's information, click the Get Gift Ideas button to load the form with the details. Figure 9-10 shows an example of what this might look like.
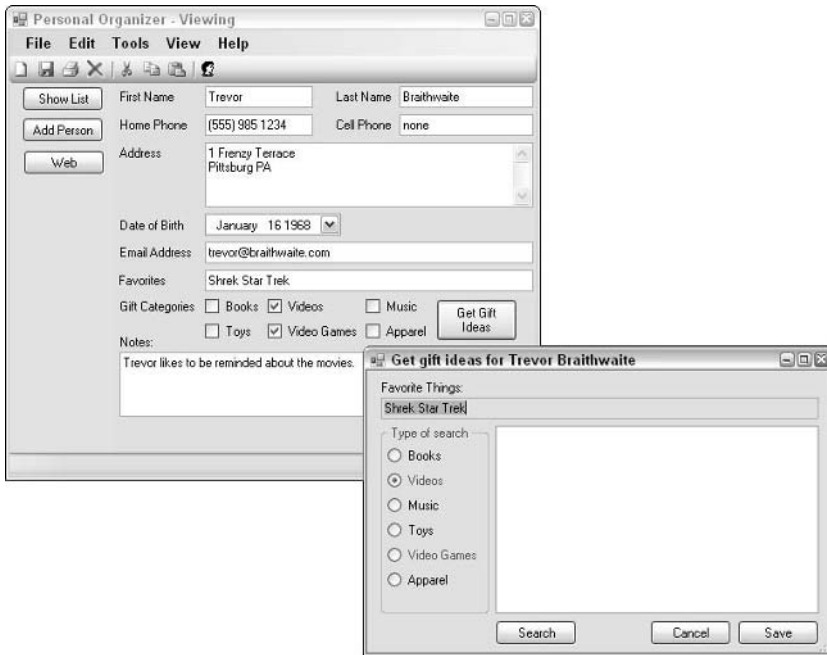
Figure 9-10

12. Now it's time to add the reference to the Amazon web service and use it when the Search button is clicked. When the user invokes the search, the Amazon web service should be called with the person's favorite things as keywords and the `SearchIndex` set to the type of search selected. Then the results should be displayed in the `CheckedListBox`.

Add the web service reference by using the Project ➪ Add Web Reference menu command. In the URL text field, enter the full location of the Amazon web service—`http://webservices .amazon.com/AWSECommerceService/AWSECommerceService.wsdl`—and click the Go button to let the wizard resolve the reference and display the available functions. Rename the Web reference to `AmazonWS` and click the Add Reference button.

13. Double-click the Search button in the Design view of `GetGiftIdeas.vb` to generate the button's `Click` event handler routine. You first need to create an instance of the `AWSECommerceService` class. This is the class that provides access to the various web service methods you can call. In addition, you need an instance of the `ItemSearch` class to build the actual web service request:

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

    Dim awsAWSE As New AmazonWS.AWSECommerceService
    Dim awsItemSearch As New AmazonWS.ItemSearch
End Sub
```

As discussed earlier in the chapter, you always need to include your own assigned Subscription ID to every function call, so do that next (replace the fictitious value used here with your own `SubsriptionID`):

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

    Dim awsAWSE As New AmazonWS.AWSECommerceService
    Dim awsItemSearch As New AmazonWS.ItemSearch

    With awsItemSearch
        .SubscriptionId = "PutYourValueHere"
    End With
End Sub
```

14. The next thing that needs to be initialized before calling the web service is an `ItemSearchRequest` collection that is assigned to the `Request` property of `awsItemSearch`. As you will perform only one search at a time, you can do this by creating an array of one object:

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

    Dim awsAWSE As New AmazonWS.AWSECommerceService
    Dim awsItemSearch As New AmazonWS.ItemSearch

    With awsItemSearch
        .SubscriptionId = "PutYourValueHere"
        Dim awsItemSearchRequest(0) As AmazonWS.ItemSearchRequest
        awsItemSearchRequest(0) = New AmazonWS.ItemSearchRequest
    End With
End Sub
```

The `ItemSearchRequest` collection needs to be filled out with the information needed to perform the search successfully. The `Keywords` property should be set to the `SelectedText` property of `txtFavorites`. You use the `SelectedText` property so that the user can select only part of the person's favorite list for the search. The only other thing to do is set the `SearchIndex` property (remember that this parameter is required so that the `ItemSearch` knows which database to search). There are numerous database options in Amazon (33 to date), but you only need to deal with six. Build an `If-Then-ElseIf` conditional block so that `SearchIndex` is set with the appropriate value.

The `ItemSearchRequest` collection is then assigned to the `ItemSearch Request` object:

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

    Dim awsAWSE As New AmazonWS.AWSECommerceService
    Dim awsItemSearch As New AmazonWS.ItemSearch

    With awsItemSearch
        .SubscriptionId = "PutYourValueHere"
        Dim awsItemSearchRequest(0) As AmazonWS.ItemSearchRequest
        awsItemSearchRequest(0) = New AmazonWS.ItemSearchRequest
        awsItemSearchRequest(0).Keywords = txtFavorites.SelectedText

        If radBooks.Checked = True Then
            awsItemSearchRequest(0).SearchIndex = "Books"
        ElseIf radVideos.Checked = True Then
            awsItemSearchRequest(0).SearchIndex = "Video"
        ElseIf radMusic.Checked = True Then
            awsItemSearchRequest(0).SearchIndex = "Music"
        ElseIf radToys.Checked = True Then
            awsItemSearchRequest(0).SearchIndex = "Toys"
        ElseIf radVideoGames.Checked = True Then
            awsItemSearchRequest(0).SearchIndex = "VideoGames"
        Else
            awsItemSearchRequest(0).SearchIndex = "Apparel"
        End If

        .Request = awsItemSearchRequest
    End With
End Sub
```

15. You are now ready to call the web service. The return value from the `ItemSearch` method call is an `ItemSearchResponse` object, so you'll need to define one to store the response. This can be done on the same line as the actual call. Insert the following line immediately below the `End With` statement:

```
Dim awsItemSearchResponse As AmazonWS.ItemSearchResponse = _
    awsAWSE.ItemSearch(awsItemSearch)
```

16. To process the response object, you first need to determine whether any results were returned. There could be many reasons for the failure to find any results, so you should inform the user by displaying the error message that Amazon returned:

```
    With awsItemSearchResponse
        If .Items(0).TotalResults = 0 Then
            MessageBox.Show(.Items(0).Request.Errors(0).Message)
        End If
    End With
```

17.   In the event that Amazon did actually return a set of results, you want to populate the `CheckedListBox` with the names of the items found. By default, the `ItemSearch` will return the first ten search results, which is enough for this application. First, clear the `CheckedListBox` contents and then add each item by writing a loop that will run as many times as there are items, adding each `Title` attribute to the `CheckedListBox`. Use the `Add` method, which enables you to set the value of the checkbox to `True` so that by default all results are marked. The final subroutine will look like this:

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

    Dim awsAWSE As New AmazonWS.AWSECommerceService
    Dim awsItemSearch As New AmazonWS.ItemSearch

    With awsItemSearch
        .SubscriptionId = "PutYourValueHere"
        Dim awsItemSearchRequest(0) As AmazonWS.ItemSearchRequest
        awsItemSearchRequest(0) = New AmazonWS.ItemSearchRequest
        awsItemSearchRequest(0).Keywords = txtFavorites.SelectedText

        If radBooks.Checked = True Then
            awsItemSearchRequest(0).SearchIndex = "Books"
        ElseIf radVideos.Checked = True Then
            awsItemSearchRequest(0).SearchIndex = "Video"
        ElseIf radMusic.Checked = True Then
            awsItemSearchRequest(0).SearchIndex = "Music"
        ElseIf radToys.Checked = True Then
            awsItemSearchRequest(0).SearchIndex = "Toys"
        ElseIf radVideoGames.Checked = True Then
            awsItemSearchRequest(0).SearchIndex = "VideoGames"
        Else
            awsItemSearchRequest(0).SearchIndex = "Apparel"
        End If

        .Request = awsItemSearchRequest
    End With
    Dim awsItemSearchResponse As AmazonWS.ItemSearchResponse = _
        awsAWSE.ItemSearch(awsItemSearch)

    With awsItemSearchResponse
        If .Items(0).TotalResults = 0 Then
            MessageBox.Show(.Items(0).Request.Errors(0).Message)
        Else
            clbResults.Items.Clear()
            For iCounter As Integer = 0 To .Items(0).Item.Length - 1
                clbResults.Items.Add(.Items(0).Item(iCounter).ItemAttributes.Title, _
                True)
            Next
```

193

```
        End If
    End With
End Sub
```

18. At this point, you can run the application, select a person, click the Get Gift Ideas button, and then choose which type of search to run. Clicking the Search button will invoke the Amazon web service, which, if successful, will then populate the `CheckedListBox` with the results, as illustrated in Figure 9-11.

Figure 9-11

19. The Cancel button should close the form without doing anything, so go ahead and add a `Click` event handler to tell the `GetGiftIdeas` form to close, which will return control to the main application. Because the main application needs to know whether the form was canceled or the list was saved, create a module-level variable called `mbCancelled` and a public read-only property that returns the value at the top of the form's code:

```
Private mbCancelled As Boolean
Public ReadOnly Property Cancelled() As Boolean
    Get
        Return mbCancelled
    End Get
End Property
```

The Cancel button's `Click` event handler can be written like this:

```
Private Sub btnCancel_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnCancel.Click
    mbCancelled = True
    Me.Hide()
End Sub
```

20. The Save button's `Click` event will also set the `mbCancelled` variable, this time to `False`, and call the `Hide` method to return control to the main part of the application. First, however, it needs to build a list of the items in the `CheckedListBox` control that have been marked for saving into a string that can be passed back to the calling part of the program. Create a module-level variable called `msGiftSuggestions` and again create a read-only property that will let other parts of the code retrieve the value:

```
Private msGiftSuggestions As String
Public ReadOnly Property GiftSuggestions() As String
    Get
        Return msGiftSuggestions
    End Get
End Property
```

Create a looping piece of code that concatenates the marked titles into a readable string and then assign the result to `msGiftSuggestions`. The `CheckedListBox` has a collection called `CheckedItems` that contains only those items in the list that have their checkbox marked, so you can easily iterate through the list. The Save button's `Click` event handler should look like this:

```
Private Sub btnSave_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnSave.Click
    Dim sGiftIdeasList As String = "Suggested gift ideas: "

    For iCounter As Integer = 0 To clbResults.CheckedItems.Count - 1
        If iCounter > 0 Then sGiftIdeasList += ", "
        sGiftIdeasList += clbResults.CheckedItems(iCounter).ToString
    Next

    msGiftSuggestions = sGiftIdeasList
    mbCancelled = False
    Me.Hide()
End Sub
```

21. The last step is to return to the `PersonalDetails` control and handle when the `GetGiftIdeas` form is closed. After the `ShowDialog` method, you should first check the `Cancelled` property; and if the form was not cancelled, then add the value in the `GiftSuggestions` property to the `Notes` field of the person:

```
If frmGetGiftIdeas.Cancelled = False Then
    txtNotes.Text += frmGetGiftIdeas.GiftSuggestions
End If
```

You're done. Run the application and go through the process of searching for items that match one of the Person records' favorites. Mark several of the results and click the Save button to add the results to the Notes area. The final result will look similar to what is shown in Figure 9-12.
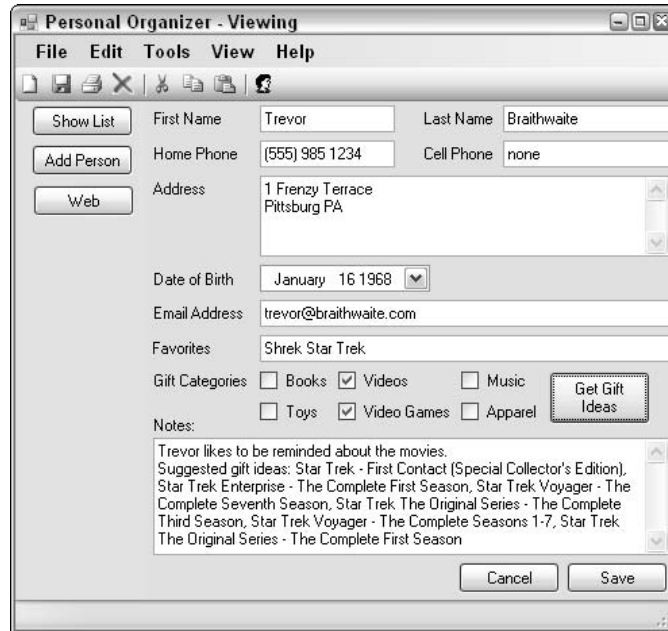
Figure 9-12

Visual Web Developer 2005 Express

As mentioned earlier in this chapter, while Visual Basic Express cannot create web applications, Microsoft has another product in the Express range that can—Visual Web Developer 2005 Express. The beauty of Visual Web Developer Express is that you can use your knowledge of Visual Basic code to write the code underneath any web application you may choose to create.

To show you how similar the process is, the following Try It Out will walk you through the process of creating a simple web service. In fact, it will create a web service that does the same thing as the first example you created in this chapter—calculate the difference between two dates as a number of days.

Try It Out   Using Web Developer Express

1. Start Visual Web Developer 2005 Express Edition (if you haven't installed it, the instructions on how to do so can be found in the Exercises section at the end of Chapter 1).

2. Select the File ⇨ New File menu command, and when the New File dialog is displayed, expand the Web list so you can see Visual Basic. From the Visual Basic templates, choose Web Service and click Open.

3. By default, Web Developer Express creates all the necessary code you will need except for the actual web service method definition. However, it even provides a sample of how to create this with the standard Hello World function.

The `WebMethod()` attribute preceding the function definition tells the underlying Visual Basic compiler that the associated function is a web service method that should be published for consumption. To add additional methods to your web service, you would create additional functions with this `WebMethod()` attribute.

The `DateDifference` method should take two dates as parameters and then calculate the difference using the built-in `DateDiff` function. The function then returns the result to the calling application or website:

```
<WebMethod()> _
Public Function DateDifference(ByVal dtFirstDate As Date, _
    ByVal dtSecondDate As Date) As String

    Dim lResult As Long = DateDiff(DateInterval.Day, dtFirstDate, dtSecondDate)
    Return CType(lResult, String)

End Function
```

4. Click the View in Browser button on the toolbar, and after a moment, your default web browser will start up and navigate to the web service you just created. Select `DateDifference` from the list, enter two dates, and click the Invoke button.

The result will be shown in formatted XML, as shown in Figure 9-13. As you can see, creating web applications, even web services, can be achieved using Web Developer Express and Visual Basic code.
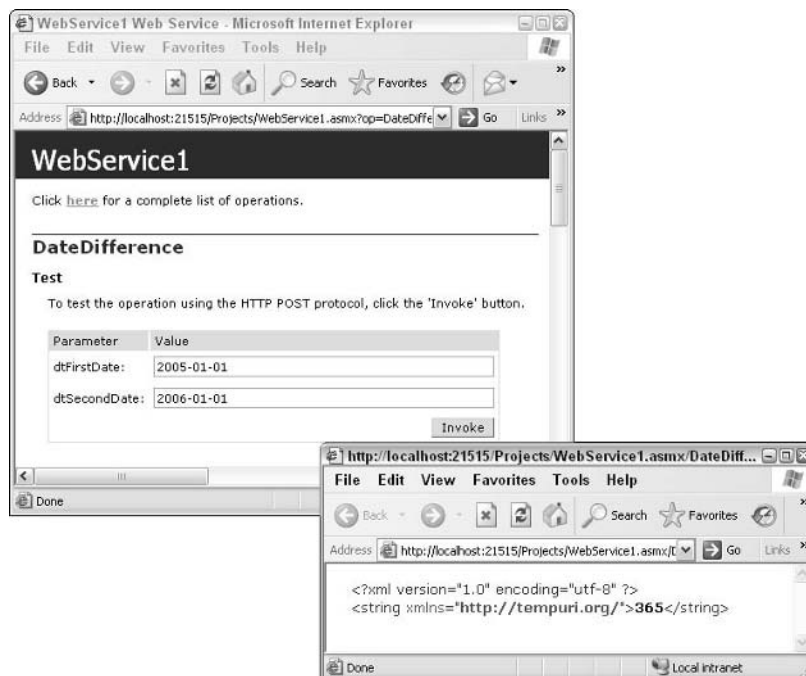
Figure 9-13

Summary

Even though Visual Basic Express does not offer the capability to write applications for the web, you can still harness the power of the Internet in your applications. Whether it is through embedding a web browser right into your application or by consuming web services available all over the Internet, the programs you create can be accessed online.

In this chapter, you learned to do the following:

❑   Implement the `WebBrowser` control in your own applications

❑   Consume web services to retrieve information from the web

❑   Use Web Developer Express to create a web service with Visual Basic Express code

Exercise

1.   In the Try It Out that added the Amazon web service to your Personal Organizer application, the `PersonalDetails` control can save the search results only when the `GetGiftIdea` form is closed. Change the program so that the `GetGiftIdea` form raises an event when the Save button is clicked, which the `PersonalDetails` control should handle and add the message to the Notes field. The Save button should also not close the `GetGiftIdea` form, so the user can perform multiple searches.

10

When Things Go Wrong

While Visual Basic Express automates a great deal of the process of creating a program, there's always the chance that something can go wrong. With the user interface elements being placed on forms with simple drag-and-drop actions, and code being syntactically checked while you type it, you might think you're safe from problems that often plague other developers' programs.

Unfortunately, that assumption is incorrect. Typographical errors, ignoring warnings by the Visual Basic Express environment, and unexpected external influences can all cause problems. The errors that are caught by the compiler are obvious to find, and hopefully to fix. However, the implicit issues that do not cause compilation errors are the ones you need to watch for.

The creators of Visual Basic Express understood that these kinds of problems often crop up and have built a number of features into the environment to help you solve them. From writing code defensively so that you catch and handle errors when they occur to breaking into the program while it's running and being able to examine the various components and their data, Visual Basic Express even makes fixing your broken application as efficient as possible.

In this chapter, you learn about the following:

- ❏   The importance of handling errors
- ❏   Debugging the code as it runs to find problem areas
- ❏   The Edit and Continue feature of Visual Basic Express

Protecting Your Code

First and foremost, the best offense against bugs in your code is a good defense. Therefore, rather than write your program without protection, you should use the code structures that Visual Basic Express offers you to detect when an error occurs and deal with it appropriately.

Try, Try, and Try Again

If you're concerned about your code potentially breaking while a user is running it — and you probably should be if it does anything more than add two numbers together — Visual Basic Express gives you a program logic block to intercept errors as they occur and to deal with whatever the problem may be. This structure also enables your program to continue to function even when an error does occur. Here are some examples of things that could go wrong:

❑   Your program tries to divide a number by zero

❑   Part of your application tries to create a file that already exists

❑   A database function cannot find the database to connect to

❑   A call to another program crashes

❑   An object you created has not been initialized before being used

There are hundreds of potential situations like these in which your program could end unexpectedly, but the `Try` block enables you to write code to handle them all. `Try` blocks tell the Visual Basic compiler that the code found within the `Try` and `End Try` statements is potentially unsafe, and that if an error occurs, rather than leave it up to Visual Basic, you intend to handle it yourself.

To intercept the error, you need to specify what errors you want to know about and tell the compiler what code to execute if an error did indeed occur. This is done with a `Catch` clause within the `Try` block, with the code on the `Catch` line identifying the error type it handles along with an object to store information about the error. The syntax of the `Try` block thus looks like this:

```
Try
    Code for normal execution goes here
Catch errObject As Exception
    Code in the event of an error goes here
End Try
```

An exception is the most general type of error. It serves as a catchall container that intercepts all errors. However, if you want to execute different code based on different error types, you just code multiple `Catch` statements, with each one having a different `Exception` object:

```
Try
    Code for normal execution goes here
Catch errDBZObject As DivideByZeroException
    Code in the event of a divide by zero error goes here
Catch errNRObject As NullReferenceException
    Code in the event of a null reference error goes here
End Try
```

Other .NET languages have the `Try` block in one form or another, but Visual Basic Express goes an extra step further by enabling the `Catch` sections to be executed conditionally. By using a `When` clause, you can break down the error handling even further, shunting off any errors that have occurred to different processing:

```
Try
    Code for normal execution goes here
```

```
    Catch errDBZObject As DivideByZeroException When MyNumber = 0
        Code in the event of a divide by zero error goes here if MyNumber = 0
    Catch errNRObject As DivideByZeroException
        Code in the event of a divide by zero error goes here if MyNumber is not 0
    End Try
```

The last piece of the `Try` block's structure is a clause that enables you to tell the Visual Basic compiler to execute a section of code whether an error has occurred or not — `Finally`. Anything in the `Finally` clause will be executed either after all of the normal code has been executed or, in the event of an error, after the normal code up to the point of the error and then the code within the appropriate `Catch` block has been executed. One handy use of the `Finally` clause might be to close a file you've been processing. This would enable you to ensure that the file is closed even if an error occurred. The placement of the `Finally` clause is after all the `Catch` clauses:

```
    Try
        Code for normal execution goes here
    Catch errObject As Exception
        Code in the event of an error goes here
    Finally
        Code to be executed every time
    End Try
```

To show how this code works, the following simple Try It Out project walks through creating an application with a piece of code that will produce an exception.

Try It Out   Using Try and Catch

1. Start Visual Basic Express and create a new Windows Application project. Place a button on the form and double-click it to create the `Click` event handler routine.

2. Add the following code to the routine:

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

    Dim MyFirstNumber As Integer = 1
    Dim MySecondNumber As Integer = 0

    Try
        MyFirstNumber = CType(MyFirstNumber / MySecondNumber, Integer)
        MessageBox.Show("We got past the divide by zero")
    Catch ex As Exception
        MessageBox.Show("An error has occurred")
    Finally
        MessageBox.Show("This message will always be displayed")
    End Try
End Sub
```

When you run the application and click the button, the integer variables will be defined and initialized with their values of 1 and 0. Then, within the `Try` block, a divide by zero calculation will be attempted. Because this is not allowed, the code will then jump to the `Catch` block and the error message will be displayed, followed by the message in the `Finally` block.

Knowing information about the error can be extremely helpful, which is why the `Catch` clause always specifies an `Exception` object. This object contains useful data relating to the error, including a *stack trace*, which indicates what code was executed immediately prior to the error occurring; and if the error was actually caused by another exception elsewhere in the code, an `InnerException` object is provided that contains that information.

The two members of the `Exception` object that you are most likely to use, however, are the `Message` property and the `ToString` method. The `Message` property returns a human-readable description of the error. It's useful because it can tell the user what happened.

The `ToString` method is used to generate a full description of the error that has occurred. Just as in many other objects, the `ToString` method concatenates important information — including the `Message` property — and is quite handy for determining where the error has occurred. To illustrate the differences, consider Figure 10-1, which shows the error that occurs when you run the code in the previous Try It Out activity. The top line is the `Message` property, while the rest of the text is the result of the `ToString` method.
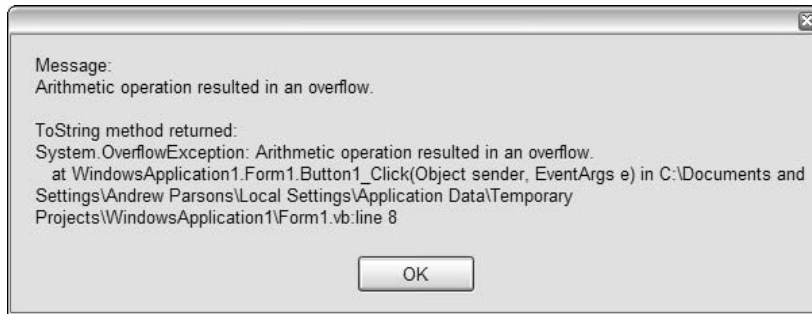
Figure 10-1

While the `Message` property does actually inform the user of what went wrong, the `ToString` method's return value provides much more information, including the type of `Exception` that occurred and exactly where the error happened. This enables you to go straight to the point at which the error occurred — the last piece of information indicates that the error occurred on line 8, which is the actual divide-by-zero line.

If an error occurs outside a `Try-Catch` block, Visual Basic Express still tries to help out. When it encounters an unhandled exception, it will pause the execution of the program on the line with the problem and pop up a detailed smart dialog window about the error, as illustrated in Figure 10-2.

Each of the lines in the Troubleshooting Tips section is a hyperlink to a location in the documentation that provides you with advice about what you can do to avoid this kind of error in the future, while the Actions section gives you access to things that may fix the problem that occurred. For example, in Figure 10-2, Visual Basic Express can temporarily create an appropriate Security Permission to allow the program to get past the Security Exception that has occurred.
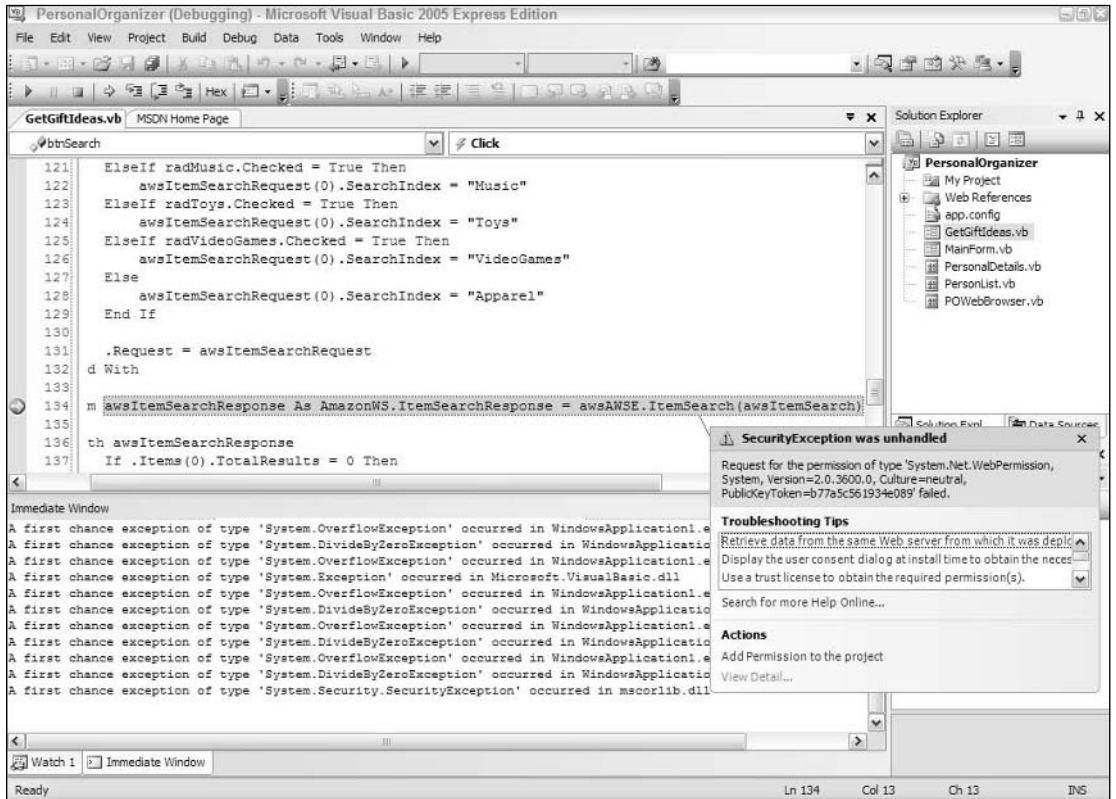
Figure 10-2

Let the Others Know!

When an error does occur, sometimes you can't do too much about it in the location where it has happened. This is particularly true if you divide your program into many functions and classes and the exception is raised in some low-level routine.

This routine may be able to intercept the error and inform the user, but you might need to perform different actions depending on when it has occurred. For example, if you have a file access routine that is shared when you first start the application and at regular intervals in the background of your program's execution, you might want to display an error message at startup time, but allow the background processing to continue, effectively ignoring the error.

You could write detailed logic processing in the `Catch` block of the low-level function, but you might miss something that way, and it will usually require additional data to be stored so you know the current state of the application.

Visual Basic Express provides you with the `Err` object, which has a method perfectly suited to the task—`Raise`. The `Raise` method will cause the function or subroutine that is currently executing to return an error to the section of code that called it. This calling code can then have its own `Try-Catch` block to

handle any errors. In the example of the file access routine, it could raise the error back to the startup routine, which in turn would display an error, or to the background processing, which could ignore it.

The syntax of the `Raise` method has a number of parameters, but the only one that is required is an error number identifying the type of error. You have the option of just passing up the error number that occurred or setting your own number.

> *If you want to create your own error, Microsoft encourages you to use a special number range for custom exceptions. Add your desired error number to the special Visual Basic Express constant* `vbObjectError`. *This ensures that it doesn't get confused with any of the system-generated errors that can occur.*

An alternative to the `Err.Raise` method is to use the `Throw` statement. `Throw` specifies an exception object to the calling code and can be trapped by a higher-level `Try-Catch` block.

In the following Try It Out exercise, you'll create an application that has a low-level function with an error that is then trapped by the calling routines and processed.

Try It Out   Throwing Exceptions Around

1. Create a new Windows Application project and add two buttons to the form. Both buttons will call the same function that contains the error.

2. Create event handler routines for each button's `Click` event and add the following code:

```
Try
    CauseError()
Catch ex As Exception
    MessageBox.Show(ex.ToString)
End Try
```

3. Below the `Click` event handlers, create a new subroutine called `CauseError` that will generate an error and pass it back to the calling routine:

```
Private Sub CauseError()
    Dim MyFirstNumber As Integer = 1
    Dim MySecondNumber As Integer = 0

    Try
        MyFirstNumber = CType(MyFirstNumber / MySecondNumber, Integer)
    Catch ex As Exception
        Err.Raise(vbObjectError + 312)
    End Try
End Sub
```

The code first tries to divide a number by zero. When the exception occurs, the execution will fall into the `Catch` block, where the `Err.Raise` method is invoked to return to the calling code with a user-specified error.

4. Run the application and click either button. The error that is generated will look like the one shown in Figure 10-3. The `ToString` method has returned a decent amount of information, including where the error occurred in the `CauseError` subroutine, as well as what routine ultimately handled the error.
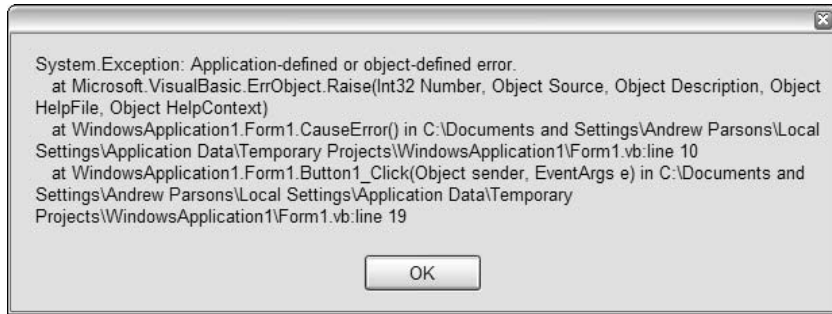
Figure 10-3

5. Stop the program and change the `Err.Raise` line to use the `Throw` statement instead. When using `Throw`, you must specify an `Exception` object that exists. In this case, you can either use the `Exception` object that was created when the divide by zero error occurred or you can create a new one. In the first Try It Out, you may have noticed that the `Exception` was a general arithmetic error, so be more specific by generating a `DivideByZeroException` instead:

```
Private Sub CauseError()
    Dim MyFirstNumber As Integer = 1
    Dim MySecondNumber As Integer = 0

    Try
        MyFirstNumber = MyFirstNumber / MySecondNumber
    Catch ex As Exception
        Throw New DivideByZeroException
    End Try
End Sub
```

6. Rerun the application and click one of the buttons. This time you'll get information about the divide-by-zero operation, but again you'll see a similar description about where the error occurred and what routine is handling it.

Troubleshooting Your Code

Sometimes your code isn't crashing but nor is it producing the results you want. A calculation may be incorrect, you might not be receiving events from an object you created, or one of many other potential problems may be occurring that enable the program to continue executing normally, although you know something is wrong.

Telling the Program to Stop

You can mark locations in the code where you want to stop the program and take a look at what's going on. These marks are known as *breakpoints*. Adding a breakpoint is straightforward — position the cursor on the line you want to halt at and press the F9 key. Pressing the F9 key again will remove the breakpoint. Alternatively, you can right-click the line and choose Insert Breakpoint from the Breakpoint submenu. Clicking in the area at the left-hand side of the code window will also toggle the breakpoint.

As the program runs and Visual Basic Express encounters a line of code with a breakpoint mark on it, the execution of the code is paused, and you will be presented with the code listing and the cursor positioned at the breakpoint line. At this point, you can check the content of variables and objects, and even modify them if need be. You can even change the code itself, which you'll see later in this chapter.

Once you've paused the program with a breakpoint, you can tell Visual Basic Express to continue by pressing the run key — F5 — or choosing Debug ⇨ Continue. The application will resume running from where it left off, with any changes you've made to the contents of variables or the code folded into the existing data.

In some cases, you may want to track through the program line by line as it progresses. For example, if you're trying to find which line of code changes a variable to an unexpected value, you could check the variable's content after each line is executed until you find it. To process the code in this way, Visual Basic Express provides two functions — Step Into (with a shortcut of F8) and Step Over (its shortcut is Shift+F8). Both of these commands can be found in the Debug menu.

Step Into will follow the execution of the program as far as it can. If the code encounters a subroutine call, Step Into will debug the subroutine as well. Step Over treats subroutine and function calls as a single line.

Figure 10-4 shows two functions, with a breakpoint marked in the first one (at line 7).
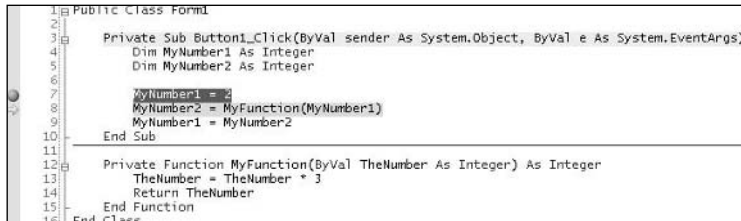

```
1  Public Class Form1
2
3      Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
4          Dim MyNumber1 As Integer
5          Dim MyNumber2 As Integer
6
7          MyNumber1 = 2
8          MyNumber2 = MyFunction(MyNumber1)
9          MyNumber1 = MyNumber2
10     End Sub
11
12     Private Function MyFunction(ByVal TheNumber As Integer) As Integer
13         TheNumber = TheNumber * 3
14         Return TheNumber
15     End Function
16  End Class
```

Figure 10-4

When the program is executed and encounters this breakpoint, it will halt processing at line 7. The following table illustrates how the Step Into and Step Over functions would follow this code.

Step Into	Step Over
Line 7	Line 7
Line 8	Line 8
Line 13	Line 9
Line 14	
Line 8	
Line 9	

Step Over is handy when you know that the result of the function or subroutine is safe and you are concentrating on the current function. Note that it still executes the code within the called function; it just doesn't break into it like Step Into does.

Figure 10-4 shows the default highlighting of breakpoints and the current statement marker. The lightly colored highlighted line indicates the current position of the code processing. In this screenshot, the breakpoint at line 7 was encountered and then the Step Into key was pressed to advance one line.

If you don't want to perform a section of code, you can jump over it by resetting the next statement. This is achieved by right-clicking the line of code you want Visual Basic Express to execute next and selecting the Set Next Statement command. For example, if you wanted to jump over line 8 in the example and instead execute line 9, you would right-click line 9 and select Set Next Statement.

Keeping Track of Variables

If all you could do were watch the code execute line by line or reset the execution point, it wouldn't be terribly useful. Fortunately, the Visual Basic Express environment gives you extensive access to the variables and objects that are available to the current line of code.

When the code execution is paused, you can position the mouse cursor over any variable you are interested in. A smart pop-up window will show the value contained in the field (see Figure 10-5).
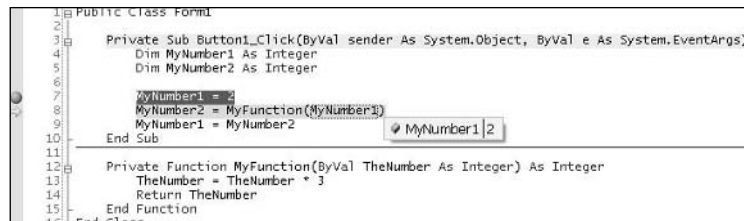
Figure 10-5

If the object is more complex, the pop-up window will include a plus sign (+) that can expand the properties of that object, enabling you to view each level of detail in turn. Figure 10-6 shows the Request object of the Amazon web service call created in Chapter 9. The Request object is shown as having a length of 1, which indicates that it has one member in its collection. This is then broken down and is identified as an `ItemSearchRequest` object, which in turn is broken down into its composite fields.

Using the Watch Windows

Although the method described in the preceding section can be performed on most fields and objects, occasionally an object cannot be viewed via this way, or you may want to be able to see multiple fields at the same time. Visual Basic Express gives you the capability to monitor fields by *watching* them. Several Watch windows are available to you while you're debugging your application.
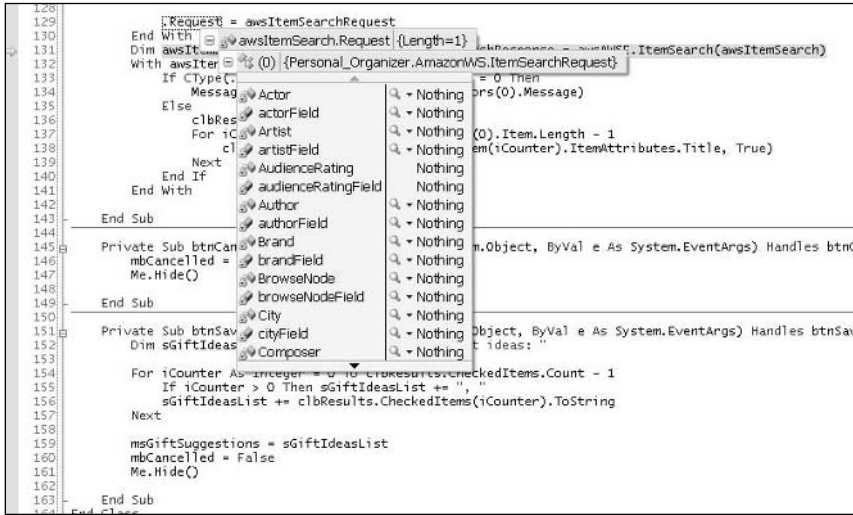
Figure 10-6

By default, the Watch windows are docked to the bottom of the IDE while you're running the application, sharing space with the Immediate window. You can view three main lists of variables: Locals, Autos, and Watch 1:

❑ The **Locals** Watch window contains every variable that has been defined in the local scope. Normally, this is the most useful list because it deals with the variables that are being processed at the time of the breakpoint. However, if the code references variables outside the scope of the local routine — for example, a module-level variable — you'll need to use one of the other two windows to keep tabs on them.

❑ The **Autos** Watch window adds temporary watches on every field that is in close proximity to the line currently being executed. This means that the entries in the Autos window can include variables defined outside the current routine, but that the list will change as you step to the next line, and then change again when the code continues to the next, and so on.

❑ The final Watch window — **Watch 1** (sometimes simply called **Watch**) — is where any manually added watches are added. You can add watches for any variable or object to this list. The benefit to using this list is that it contains watches only for the variables you're interested in, and you can define a list of local and global variables that doesn't change as you debug from one line to the next.

If a particular variable is not currently in scope, it will be marked as not being defined (as shown in Figure 10-7). In this screenshot, the last two variables — a `Request` object and a `String` variable — are not available from the current line of code.

To add variables to this watch list, locate the variable you want to monitor in the code, right-click it, and select Add Watch. To delete a variable when it's no longer needed, right-click its entry in the Watch window and select Delete Watch.

If the Watch window you're after is not visible in the IDE, you can activate it through the Debug ➪ Windows submenu.
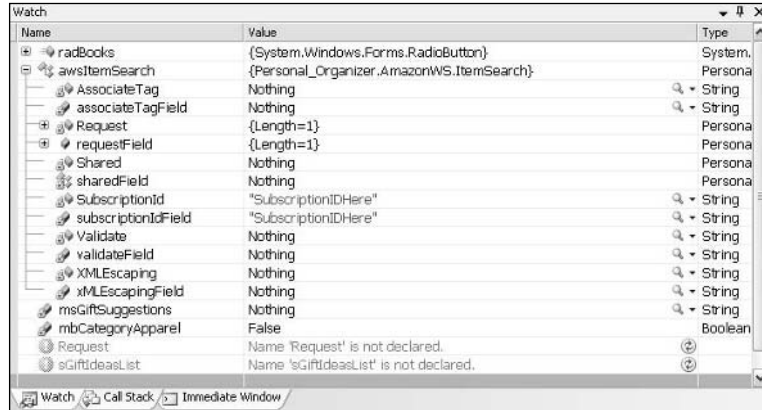
Figure 10-7

One great advantage of the Watch windows is that not only can you view the values of the variables and objects, but you can also change them. When you assign a different value to a variable and allow the program to continue to execute, it will use the new value instead of the old one, thereby enabling you to change the way the program executes.

In the example shown in Figure 10-7 the `SubscriptionId` field has a value of `SubscriptionIDHere`, which will cause the web service method call to fail. You could put a breakpoint directly before the web service is called and replace that value with your valid `SubscriptionID` and then let the program continue.

At times you may want the best of both worlds — the structured formatting of the Watch window and the capability to change the contents of the variables, along with the temporary nature of hovering the mouse cursor over the field. This is where the Quick Watch feature comes into play. Right-clicking a field in which you are interested, you can select the Quick Watch command to display a dialog window (similar to the one shown in Figure 10-8).

This window enables you to navigate through the various properties of the object you're looking at, changing them if needed, without adding the watched variable permanently to your Watch windows.

Using the Immediate Window

Visual Basic Express also gives you the capability to keep tabs on your application without pausing it at every line. The `Debug` object has a number of properties and methods that can be used to display information about your program in the Immediate window.

The most useful method of the `Debug` object is `WriteLine`. This function writes the string you specify to the Immediate window and has the default syntax of `Debug.WriteLine(YourMessageHere)`. Because the parameter can be any string, you can build a message much like you would for a dialog window or error display, like so:

```
Debug.WriteLine("Successfully processed file: " + MyFileName)
```
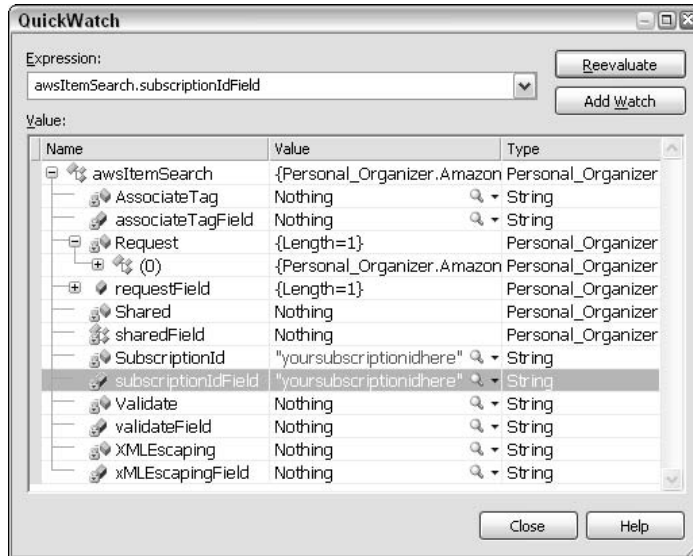
Figure 10-8

When this line is executed, a line will be added to the Immediate window, containing the success message along with the value of `MyFileName`. The `Debug` object has a number of other properties that can control how the information is displayed in the `Immediate` window. `Indent` and `Unindent` will move the information over to provide simple formatting. The `WriteLineIf` method will display the message only if the specified condition is met, and the `Write` and `WriteIf` methods will display the information without adding a new line.

The following Try It Out puts all of these actions together to produce some simple formatted output in the Immediate window.

Try It Out   Using the Debug Object
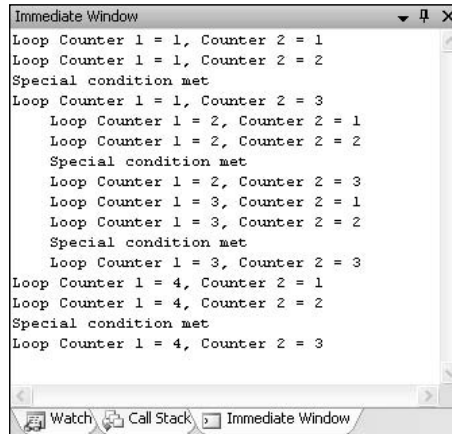
1. Create a new Windows Application project and add a button to the form.

2. Create a `Click` event handler routine for the button and add the following code:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    For iCounter1 As Integer = 1 To 4
        If iCounter1 = 2 Then
            Debug.Indent()
        ElseIf iCounter1 = 4 Then
            Debug.Unindent()
        End If
        For iCounter2 As Integer = 1 To 3
            Debug.WriteLine("Loop Counter 1 = " + iCounter1.ToString _
    + ", Counter 2 = " + iCounter2.ToString)
            Debug.WriteLineIf(iCounter2 = 2, "Special condition met")
        Next
    Next
End Sub
```

This code will perform two sets of loops, one inside the other, and display a number of messages in the Immediate window, identifying the value of the loop counters.

3. Place a breakpoint on the `End Sub` line by right-clicking the line and selecting Insert Breakpoint from the Breakpoint submenu, and run the program. You added the breakpoint so that you could see the output in the Immediate window right after it has been executed.

Figure 10-9 shows the output of this code. Every time `iCounter2` has a value of 2, the extra line is printed to the Immediate window, while the loops when `iCounter1` has a value of 2 and 3 are indented to the right.


```
Immediate Window                                    ▾ ╄ ✕
Loop Counter 1 = 1, Counter 2 = 1
Loop Counter 1 = 1, Counter 2 = 2
Special condition met
Loop Counter 1 = 1, Counter 2 = 3
        Loop Counter 1 = 2, Counter 2 = 1
        Loop Counter 1 = 2, Counter 2 = 2
        Special condition met
        Loop Counter 1 = 2, Counter 2 = 3
        Loop Counter 1 = 3, Counter 2 = 1
        Loop Counter 1 = 3, Counter 2 = 2
        Special condition met
        Loop Counter 1 = 3, Counter 2 = 3
Loop Counter 1 = 4, Counter 2 = 1
Loop Counter 1 = 4, Counter 2 = 2
Special condition met
Loop Counter 1 = 4, Counter 2 = 3

⬚ Watch  ⬚ Call Stack  ⬚ Immediate Window
```

Figure 10-9

Gone Too Far and Don't Want to Stop?

If often happens that you are running your application and all of a sudden the code breaks into debug mode because of an unforeseen error. If this happens right near the beginning of the run, you can just end the program, fix the error, and restart. That's what you have to do in most programming languages. Visual Basic Express gives you an alternative: Edit and Continue.

Edit and Continue enables you to break into the code while the application is still running, change a piece of the logic, and then continue the program's execution. This powerful feature is particularly handy if the program has run through a large number of operations and you don't want to go through the entire process again, particularly because you can easily see the problem when it has been presented to you by Visual Basic Express.

Obviously, this capability is not intended for you to make wholesale changes in your code, but for those situations when you discover a minor bug that will cause your application to function in an unexpected way. For example, suppose your program is crashing but you don't know why. You add a breakpoint to the beginning of the function where the exception is being raised and start stepping through the code line by line using Step Into.

After a few moments of tracking the code, you realize that a calculation is using an incorrect variable as part of the equation. Rather than stop the program, change the code and then restart, you can change the equation so that it uses the correct variable.

The following Try It Out takes a Visual Basic Express project that is experiencing a problem and walks through the process of debugging it and correcting the problem while in break mode.

Try It Out   Using Edit and Continue

1.   Locate the project solution `Problem Child.sln` in the `Chapter 10\Problem Child` folder from the code download you can find at `www.wrox.com` and open it in Visual Basic Express.

2.   Run the application, enter the name of a child and a problem, and click the Process button. The Results textbox should display a message according to the following table.

Condition	Message
A girl with a phobia of some kind	She's scared of something
A boy with a phobia of some kind	He's scared of something
Either a boy or a girl with a different problem	Sounds like \<name> might have a serious problem

However, a quick test will prove that the results are all mixed up. You'll have to fix the program.

3.   Stop running the application and add a breakpoint to the first line in the button's `Click` event handler, and restart the program. Click the button again to process the child's problem, and Visual Basic Express will break into the program.

4.   Use Step Into to trace through the program until you enter the `ProcessProblems` function in the `Child` class. The first problem is that the `InStr` function returns a value of 0 if the search string is not found—the exact opposite to what's intended. While you're still in debug mode, change the equals sign (=) to a greater than sign (>) and press F5 to resume normal processing.

Now the problem processing is working a lot better. It's detecting phobias correctly, but you may notice that it always refers to a child as a he regardless of which sex you chose.

5.   Breaking into the button's `Click` event, you might realize that the `Sex` property of the `Child` object is never set. Immediately before the call to the `ProcessProblems` method, insert the following code:

```
If radBoy.Checked = True Then
    myChild.ChildSex = Child.ChildSexes.Boy
Else
    myChild.ChildSex = Child.ChildSexes.Girl
End If
```

6.   Resume the program again and check whether girls with phobias now display correctly.

Edit and Continue is a powerful feature that enables you to change the code while it's still running. The change can be as minor as altering a variable name or operation in an equation or as complex as replacing a whole block of logic or adding a new set of code, as illustrated in the preceding Try It Out.

Summary

Even though you can still have problems when writing programs in Visual Basic Express, it gives you many troubleshooting tools to facilitate tracking down the issues and fixing them. Being able to view the content of any objects that are being processed is extremely valuable when determining what is going wrong. The capability to change the code on the fly and have the program continue with the new logic makes it even better.

In this chapter, you learned to do the following:

❑   Handle errors in your code so your application doesn't crash

❑   Harness the variety of debugging features Visual Basic Express gives you to find out the status of your program's objects and variables

❑   Use Edit and Continue to make changes to your program without having to stop it

In the next chapter, you'll begin to learn more advanced topics such as time-based logic and event processing that is contingent on other information. You will start to bring together all of the information you've learned throughout this book.

Exercise

1.   Open the Personal Organizer project you worked on in Chapter 9 and debug through the call to the Amazon web service. Try to determine how many items are returned from the call by looking at the `ItemSearchResponse` object in the Quick Watch window before the `CheckedListBox` is populated.

Part III
Making It Hum

11

It's Printing Time!

Over the course of the first two parts of this book, you have been introduced to a wide variety of features available in Visual Basic Express. The last few chapters serve to round out your knowledge of how to get the most out of this great development tool. First you'll learn how to print information from your programs and harness various system components such as timers and help and error providers—that's the subject of this chapter. After that, you'll learn about XML and how useful it is in Visual Basic Express, and delve into security and deployment of your applications.

In this chapter, you learn about the following:

❑   Using the `Timer` class to perform actions periodically

❑   The different Print controls and how to print documents

❑   Various system components that will add the finishing touches to your application

Timing Is Everything — Well, Almost

Most of the code you've written up to this point is reactive, based on what users are doing. If they click a button, the button's `Click` event is raised and your event handler routine kicks into gear. If they change the contents of a `TextBox`, the `TextChanged` event fires and again your code takes over and processes the change.

That's all good, but sometimes you're going to want to perform a function based on a regular schedule, independent of whether the user is doing something or not. That's where the `Timer` class comes in. You have two Timer objects available for use in Visual Basic Express, but both do the same thing. The `System.Timers.Timer` class is a more general class that can be used in any program and does not require a form in order to execute. Objects of this type can be created in code as follows:

```
Private WithEvents MyTimer As System.Timers.Timer
```

The `System.Windows.Forms.Timer` component is specifically designed for use on Windows Forms and will run properly only if defined within the context of a form. To add one of these

`Timer` controls to your form, locate the Components category in the Toolbox and drag a `Timer` object to the form. As it doesn't have any visible aspect, it will be added to the tray area below the form's design surface.

Alternatively, you can create one using code, in the same way as the generic `Timer` object:

```
Private WithEvents MyTimer As Timer
```

Regardless of which `Timer` object you use, you use two main properties to control the functionality of the timing mechanism:

❑ The `Interval` property contains the number of milliseconds the timer is to wait before firing its `Tick` event. This means if you want the timer to wait one second, you need to set the `Interval` property's value to 1,000, and an hour would be $1,000 \times 60 \times 60$, or 3,600,000.

❑ The `Enabled` property determines whether the timer is currently running. If `True`, then the timer is keeping track of the number of milliseconds since it was first started, or the last time the `Tick` event was raised. If disabled, then the timer sits there doing nothing.

The only event worth looking at is the one already mentioned — `Tick`. When the specified interval has elapsed, the timer object raises the `Tick` event so your program can do its scheduled processing. The `Tick` event of the Windows Forms timer uses the same event signature as most other control events:

```
Private Sub MyTimer_Tick(ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles MyTimer.Tick
```

To tell Visual Basic Express to begin timing the interval on a timer object, you can either set the `Enabled` property to `True` or call the `Start` method. Similarly, to halt the timing process, set `Enabled` to `False` or call the `Stop` method. If you ever need to change the `Interval` period, you should always stop the timer first so that it doesn't get confused about what kind of interval it is supposed to track:

```
MyTimer.Stop
MyTimer.Interval = 5000
MyTimer.Start
```

The `Timer` control will continue to raise the `Tick` event after every interval. This means that if you want the timer to time only one period, you must explicitly stop the `Timer`. In addition, and this can be a problem if a lot of processing is involved whenever the timer raises the `Tick` event, if you're in the event handler of the timer and the interval elapses again, yet another `Tick` event will be fired. To avoid this, always explicitly stop the timer when the `Tick` event is fired and then restart it when you're finished processing.

Interestingly, the generic `Timer` class has an additional property to avoid this kind of problem — `AutoReset`. Setting this property to `False` ensures that the timer fires only once and then stops processing time intervals.

A Use for Timers

One handy use for the `Timer` is to keep track of the state of information and then act accordingly. For example, it might be the case that whenever the date changes, you want your application to update a

label displaying today's date. Alternatively, it could be used to periodically check whether a file exists, and, if so, read the file to process the contents.

Visual Basic Express has another control that can be added to a form called the `NotifyIcon`. This element enables you to add an icon to the notification area in the bottom-right corner of your Windows desktop. From here you can provide your users with quick access to common commands for your application, and display important information as it occurs.

The `NotifyIcon` control is found in the Common Controls section of the Toolbox, and the two main properties you should set are the `Icon` (what appears in the notification area) and the `Text` (this is displayed when users hovers their mouse cursor over the icon). In addition to this, you can assign a `ContextMenuStrip` to the `NotifyIcon`. This context menu is displayed when the user right-clicks on the icon.

One additional handy feature of the `NotifyIcon` control is to use it to inform the user of important events in your application even if the application is not active. The `BalloonTip` properties and method are used to assign the settings of a customized tool tip and then display it for a specified number of seconds. The four associated elements are as follows:

❑   **BalloonTipTitle** — Contains the bold title text of the tool tip.

❑   **BalloonTipText** — Contains the main text of the tool tip when it appears.

❑   **BalloonTipIcon** — One of four icons to optionally display along with the message. These icons use the system-defined images so that your application is integrated with the rest of the operating system.

❑   **ShowBalloonTip** — This method tells the `NotifyIcon` to display the tool tip using the settings you've assigned. The default version contains one parameter, which specifies the number of seconds to display the tool tip before hiding it again. An additional version enables you to set all of these properties at once, which is handy for displaying temporary messages. The following two sets of code would display the same `BalloonTip`:

```
MyNotifyIcon.BalloonTipIcon = ToolTipIcon.Error
MyNotifyIcon.BalloonTipText = "There is a problem in your database!"
MyNotifyIcon.BalloonTipTitle = "Database Problem"
MyNotifyIcon.ShowBalloonTip(4)

MyNotifyIcon.ShowBalloonTip(4, "Database Problem ", _
    "There is a problem in your database!", ToolTipIcon.Error)
```

The `NotifyIcon` class enables you to react to the user's actions with several events. Whenever the `BalloonTip` is displayed, your program can intercept the `BalloonTipShown`, `BalloonTipClosed`, and `BalloonTipClicked`. This last event is useful to enable users to perform an action when they see the notification tool tip.

To see how `Timers` and `NotifyIcons` can work together, the next Try It Out adds a reminder system to your Personal Organizer application. Whenever someone's birthday is less than seven days away, the application pops up a message to remind the user about it. If a person's birthday already occurred in the last seven days, a different message warns users that they might have forgotten the day.

219

Try It Out   Using the Timer Effectively

1. Start Visual Basic Express and open the Personal Organizer application project you've been working on throughout the book. If you don't have the project up to date, you will find a version of the project in the `Code\Chapter 11\Personal Organizer Start` folder of the code download from `www.wrox.com`. This project contains everything done up to the beginning of Chapter 11.

2. Open the Main Form in Design view and add a `Timer` by clicking and dragging it from the Components category in the Toolbox to the design surface of the form. Name it `tmrReminders`. Add a `NotifyIcon` to the form in the same way — this time it will be found in the Common Controls category — and name it `niReminders`.

    For the `NotifyIcon` to be displayed in the notification area, you'll need to set its `Icon` property. The `Code\Chapter 11` folder contains a sample icon you can use for this purpose. You should also set the `Text` property to the name of the application or purpose — in this case, set it to Personal Organizer Reminders.

3. Create an event handler routine for the form's `Load` event and add the following code:

```
    Private Sub frmMainForm_Load(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Load
        With tmrReminders
            .Interval = 1000 * 60 * 60 * 6
            .Enabled = True
        End With
    End Sub
```

    This sets the interval to every six hours. For testing, you might like to reduce the interval to something more frequent, such as 10 seconds. Remember that you can also start the timer with the `Start` method.

4. Whenever the specified interval elapses, the timer raises its `Tick` event, so create an event handler routine for that event. The first thing to do is stop the timer and the last thing should be to restart it so that it keeps track of the interval again:

```
Private Sub tmrReminders_Tick(ByVal sndr As Object, ByVal e As System.EventArgs) _
   Handles tmrReminders.Tick
   With tmrReminders
     .Enabled = False

     .Enabled = True
   End With
End Sub
```

5. To determine whether any people with birthdays fit into the date range, you'll need to retrieve the `Person` rows from the table that belongs to the currently logged on user. This is done using the same database code that you've been working with since Chapter 7:

```
Private Sub tmrReminders_Tick(ByVal sndr As Object, ByVal e As System.EventArgs) _
   Handles tmrReminders.Tick
   With tmrReminders
     .Enabled = False
     Dim BirthdayPersonAdapter As New
_PO_DataDataSetTableAdapters.PersonTableAdapter
     Dim BirthdayPersonTable As New _PO_DataDataSet.PersonDataTable
```

```
      BirthdayPersonAdapter.Fill(BirthdayPersonTable)

      Dim BirthdayDataView As DataView = BirthdayPersonTable.DefaultView
      BirthdayDataView.RowFilter = "POUserID = " + mCurrentUserID.ToString
      With BirthdayPersonDataView
        If .Count > 0 Then
          ... process birthdates here
        End If
      End With
      .Enabled = True
    End With
  End Sub
```

6. You need to check each person's birth date in turn, and, if you find even one, set a flag so you can show the `NotifyIcon` object's `BalloonTip`. Create and initialize a Boolean flag to keep track of birthdays and a string variable to store all of the birthdays that occur in the next seven days. Do the same for birthdays that have occurred in the last seven days:

```
Dim bFoundBirthdaysToRemember As Boolean = False
Dim sBirthdayReminders As String = vbNullString
Dim bFoundBirthdaysForgotten As Boolean = False
Dim sBirthdaysForgotten As String = vbNullString
```

7. Loop through the `BirthdayDataView` and get each person's birthday:

```
For BirthdayCheckCounter As Integer = 0 To .Count - 1
  With .Item(BirthdayCheckCounter)
    Dim PersonBirthday As Date = CType(.Item("DateOfBirth"), Date)
  End With
End For
```

The problem with this date is that it contains the person's birth year as well. As you are interested only in the day and month of the person's birthday, you can compare these to the current day and month by creating a temporary date variable to store the current birthday date:

```
Dim PersonBirthdate As Date = CType(PersonBirthday.Month & "/" & _
    PersonBirthday.Day & "/" & Now.Year, Date)
```

8. You can now calculate the number of days between the birthday date and today's date. Use the `DateDiff` method that Visual Basic Express provides and specify the interval as `Day`:

```
Dim NumberOfDays As Long = DateDiff(DateInterval.Day, Now, PersonBirthdate) + 1
```

9. You can now check whether the interval is less than seven days. If the birthday has occurred in the past, this calculation will contain a negative number, so check for a value range of –7 through to 0 for birthdays that have occurred in the last seven days, and a value range of greater than –1 for birthdays yet to occur:

```
If NumberOfDays < 7 Then
    If NumberOfDays > -7 And NumberOfDays < 0 Then
        ... keep track of forgotten birthday here
    ElseIf NumberOfDays > -1 Then
        ... keep track of upcoming birthday here
    End If
End If
```

221

10. The two sets of code for the different conditions are quite similar. First you need to set the appropriate flag to keep track of birthdays that fit the criteria. Then you need to append the birthday information to the `String` variable you defined for that purpose.

To keep each person's birthday separate, include a newline character between each one. You don't need to do this for the first one, so check whether the string contains text already; if so, add the line feed. To ensure that the message is grammatically correct, you can use the `IIf` method to determine whether the period is a single `day` or multiple `days`. The only other thing you should do is ensure that you multiply the number of days by –1 if the number is negative:

```
If NumberOfDays < 7 Then
    If NumberOfDays > -7 And NumberOfDays < 0 Then
        bFoundBirthdaysForgotten = True
        If sBirthdaysForgotten <> vbNullString Then sBirthdayReminders &= vbCrLf
        Dim DayString As String = IIf(NumberOfDays = -1, " day", " days").ToString
        sBirthdaysForgotten &= .Item("NameFirst").ToString.Trim & " " & _
            .Item("NameLast").ToString.Trim & "'s birthday " & _
            (NumberOfDays * -1).ToString & DayString & " ago!"
    ElseIf NumberOfDays > -1 Then
        bFoundBirthdaysToRemember = True
        If sBirthdayReminders <> vbNullString Then sBirthdayReminders &= vbCrLf
        Dim DayString As String = IIf(NumberOfDays = 1, " day", " days").ToString
        sBirthdayReminders &= .Item("NameFirst").ToString.Trim & " " & _
            .Item("NameLast").ToString.Trim & "'s birthday in " & _
            NumberOfDays.ToString & DayString
    End If
End If
```

11. Once you've calculated the message strings and have determined that you have birthdays to remind the user about, you can then use the `NotifyIcon` to display the information. Set the `BalloonTip` properties as discussed earlier in this chapter and then call the `ShowBalloonTip` method. Note that the following code shows only one message at a time:

```
If bFoundBirthdaysToRemember Or bFoundBirthdaysForgotten Then
    With niReminders
        .Visible = True
        If bFoundBirthdaysToRemember Then
            .BalloonTipIcon = ToolTipIcon.Info
            .BalloonTipText = sBirthdayReminders
            .BalloonTipTitle = "Birthday Reminders"
        Else
            .BalloonTipIcon = ToolTipIcon.Warning
            .BalloonTipText = sBirthdaysForgotten
            .BalloonTipTitle = "Have you forgotten these dates?"
        End If
        .ShowBalloonTip(5)
    End With
End If
```

12. Run the application and wait for the interval you specified in the `Form`'s `Load` event handler. After that time, if you have any people with birth dates falling within the next seven days or the last seven days, you'll be notified, as shown in Figure 11-1. The final `Tick` event handler routine looks like this:

```
Private Sub tmrReminders_Tick(ByVal sndr As Object, ByVal e As System.EventArgs) _
  Handles tmrReminders.Tick
  With tmrReminders
    .Enabled = False
    Dim BirthdayPersonAdapter As New
_PO_DataDataSetTableAdapters.PersonTableAdapter
    Dim BirthdayPersonTable As New _PO_DataDataSet.PersonDataTable
    BirthdayPersonAdapter.Fill(BirthdayPersonTable)

    Dim BirthdayDataView As DataView = BirthdayPersonTable.DefaultView
    BirthdayDataView.RowFilter = "POUserID = " + mCurrentUserID.ToString
    With BirthdayPersonDataView
      If .Count > 0 Then
        Dim bFoundBirthdaysToRemember As Boolean = False
        Dim sBirthdayReminders As String = vbNullString
        Dim bFoundBirthdaysForgotten As Boolean = False
        Dim sBirthdaysForgotten As String = vbNullString
        For BirthdayCheckCounter As Integer = 0 To .Count - 1
          With .Item(BirthdayCheckCounter)
            Dim PersonBirthay As Date = CType(.Item("DateOfBirth"), Date)
            Dim PersonBirthdate As Date = CType(PersonBirthday.Month & "/" & _
              PersonBirthday.Day & "/" & Now.Year, Date)
            Dim NumberOfDays As Long = DateDiff(DateInterval.Day, Now, _
              PersonBirthdate) + 1
            If NumberOfDays < 7 Then
              If NumberOfDays > -7 And NumberOfDays < 0 Then
                bFoundBirthdaysForgotten = True
                If sBirthdaysForgotten <> vbNullString Then _
                  sBirthdayReminders &= vbCrLf
                  Dim DayString As String = IIf(NumberOfDays = -1, " day", _
                    " days").ToString
                  sBirthdaysForgotten &= .Item("NameFirst").ToString.Trim & " " & _
                    .Item("NameLast").ToString.Trim & "'s birthday " & _
                    (NumberOfDays * -1).ToString & DayString & " ago!"
              ElseIf NumberOfDays > -1 Then
                bFoundBirthdaysToRemember = True
               If sBirthdayReminders <> vbNullString Then _
                  sBirthdayReminders &= vbCrLf
                  Dim DayString As String = IIf(NumberOfDays = 1, " day", _
                    " days").ToString
                  sBirthdayReminders &= .Item("NameFirst").ToString.Trim & " " & _
                    .Item("NameLast").ToString.Trim & "'s birthday in " & _
                    NumberOfDays.ToString & DayString
              End If
            End If
          End With
        End For
        If bFoundBirthdaysToRemember Or bFoundBirthdaysForgotten Then
          With niReminders
            .Visible = True
            If bFoundBirthdaysToRemember Then
              .BalloonTipIcon = ToolTipIcon.Info
              .BalloonTipText = sBirthdayReminders
              .BalloonTipTitle = "Birthday Reminders"
            Else
```

```
                .BalloonTipIcon = ToolTipIcon.Warning
                .BalloonTipText = sBirthdaysForgotten
                .BalloonTipTitle = "Have you forgotten these dates?"
            End If
            .ShowBalloonTip(5)
          End With
        End If
      End If
    End With
    .Enabled = True
  End With
End Sub
```
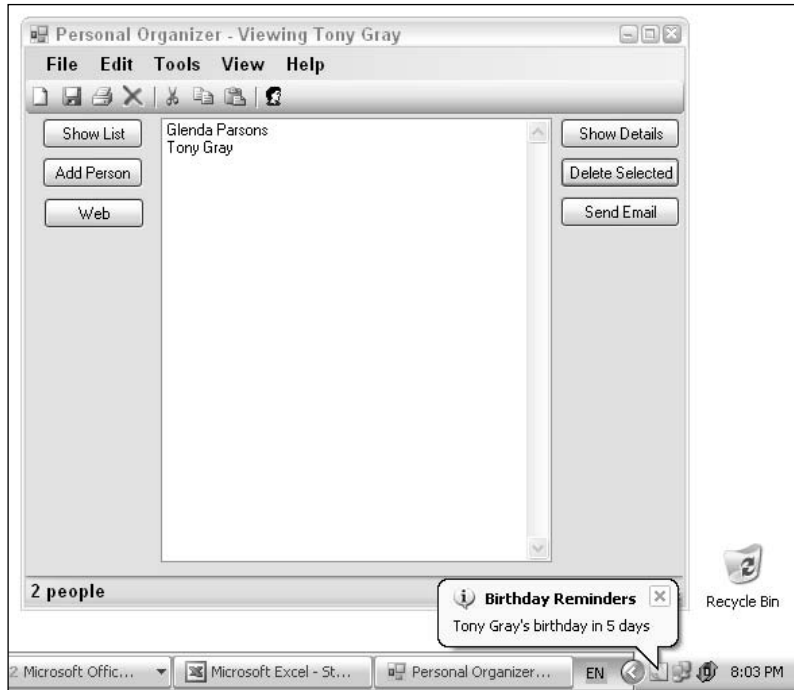

Figure 11-1

Printing

Visual Basic Express gives you five printing components that work together to provide a robust solution for implementing reporting capabilities into your application. Three of the controls give you direct access to the system dialogs for printing:

❑ **PageSetupDialog** — Enables your users to select various print page settings, such as paper size, margins and orientation, and access to a select printer.

❑ **PrintDialog** — Rather than print directly to the printer, you can use this dialog to give users several options before they print the document. Not only can they select the printer to do the printing, you can also optionally include the Page Range box to select specific pages, the Print to file option, and whether it is selected by default (see Figure 11-2).
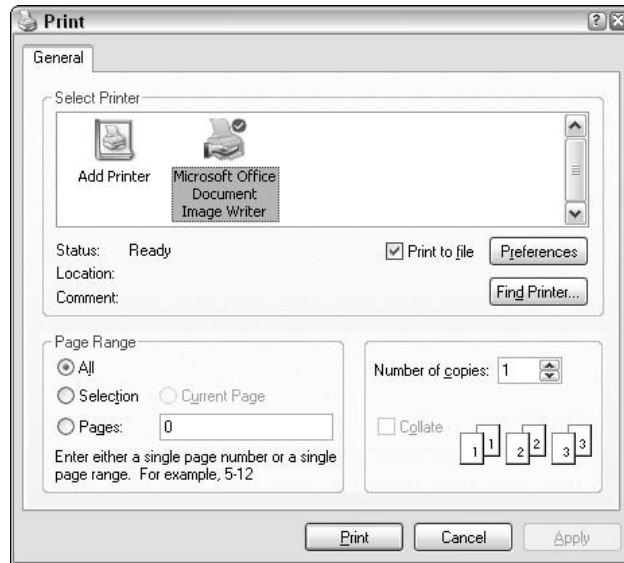
Figure 11-2

❑ **PrintPreviewDialog** — This dialog component gives your program the capability to display a preview page with a toolbar containing basic functionality to navigate through the previewed document.

The `PrintPreviewControl` is a component that you can use to embed preview capabilities right into one of your existing forms or controls. Rather than being presented as a separate window, as is the case with the `PrintPreviewDialog`, the `PrintPreviewControl` is dockable within your form and does not have any toolbars or other elements besides the actual previewed document. This means you need to implement any code that you require to enable users to navigate around or zoom in and out of the document.

Finally, all of these components revolve around the `PrintDocument` object. This class encapsulates the printing process, raising events when it is ready to print a page and taking commands about how and what to print and where on each page. The Print and Print Preview sets of functionality both use `PrintDocument` components to generate their output, and you can use the same `PrintDocument` object for both, thus ensuring that the previewed output is identical to the printed output.

The `PrintDocument` class works by raising a `PrintPage` event whenever the printer is ready to print a page. You can invoke the printing process by either having one of the `Print...Dialog` controls point to the `PrintDocument` and then call the `ShowDialog` method or manually, by calling the `PrintDocument`'s `Print` method.

Either way, the `PrintPage` event is where the main printing process takes place. The `PrintPage` event comes with two parameters. The first is the standard `sender` object that contains the control that causes the event to be raised in the first place. The second is a `PrintPageEventArgs` object, which is where you do all the work.

The `PrintPageEventArgs` object has a `Graphics` object that is used to control the printing of text and graphics. This `Graphics` object can also return the current printable area and enables you to measure the space taken up by an element you want to print.

The other property of interest is the `HasMorePages` Boolean flag. After you process your printing code in the `PrintPage` event, you should set the `HasMorePages` flag if there is more to print. If this is the last page to print, set `HasMorePages` to `False`, and the `PrintDocument` object finalizes the printing process. These two properties of the `PrintPageEventArgs` are used to formulate the logic you use in any printing process:

1. Start the print process.

2. The `PrintPage` event is raised.

3. For each element you want to print, first measure it to determine whether it will fit in the available space left and, if so, then draw it.

4. If you find that an element does not fit into the printable region, set the `HasMorePages` flag to `True` and remember where you are in the print process.

5. Once the `PrintDocument` object has finished processing the print requests you made in the `PrintPage` event, it fires another `PrintPage` if the `HasMorePages` was set to `True`; otherwise, it ends.

The following Try It Out walks you through the process of printing a report of the people registered in the Personal Organizer database. Working through it is the best way to understand how the Print controls work together to give you an effective printing system in your code. It uses the `PrintDialog` and `PrintPreviewDialog` controls in conjunction with the `PrintDocument` control to create a report to the printer the user chooses.

Try It Out   Printing

1. Return to Visual Basic Express and the Personal Organizer application project. To print a report, you'll first need to create the text for the report, so create a new function in the `GeneralFunctions.vb` module and call it `GenerateReport`. If you didn't complete the preceding Try It Out activity, you can find a copy of the project in the `Chapter 11\Personal Organizer Printing Start` folder of the downloaded code for this book. It's complete up to the start of this walkthrough.

   Define the function so that it accepts a `UserID` to restrict the reporting to a particular `POUser` and returns a string containing the report information. Use the same technique used in the other database functions to get a list of `PersonRows` that belong to the specified user:

```
Public Function GenerateReport(ByVal UserID As Integer) As String
    Dim GetPersonAdapter As New _PO_DataDataSetTableAdapters.PersonTableAdapter
    Dim GetPersonTable As New _PO_DataDataSet.PersonDataTable
    GetPersonAdapter.Fill(GetPersonTable)
```

```
        Dim ReportString As String = vbNullString
        For Each MyRow As _PO_DataDataSet.PersonRow In _
            GetPersonTable.Select("POUserID = " & UserID.ToString)
            With MyRow
                ... report generation goes here
            End With
        Next
        Return ReportString
    End Function
```

2. Create the contents of `ReportString` by concatenating the details about each user. Note that the first line includes a special set of characters, `$HDG`, that are used in the printing process to find the heading lines and format them differently on the page:

```
Public Function GenerateReport(ByVal UserID As Integer) As String
    Dim GetPersonAdapter As New _PO_DataDataSetTableAdapters.PersonTableAdapter
    Dim GetPersonTable As New _PO_DataDataSet.PersonDataTable
    GetPersonAdapter.Fill(GetPersonTable)

    Dim ReportString As String = vbNullString
    For Each MyRow As _PO_DataDataSet.PersonRow In _
        GetPersonTable.Select("POUserID = " & UserID.ToString)
        With MyRow
            ReportString &= "$HDG" & .NameFirst.Trim & " " & .NameLast.Trim & vbCrLf
            ReportString &= "Home Phone: " & .PhoneHome.Trim & vbCrLf
            ReportString &= "Email: " & .EmailAddress.Trim & vbCrLf
            ReportString &= "Birthday: " & .DateOfBirth.ToShortDateString & vbCrLf
        End With
    Next
    Return ReportString
End Function
```

3. Open `MainForm` in Design view and add a `PrintPreviewDialog` (named `prnprvDialog`), a `PrintDialog` (named `prnDialog`), and a `PrintDocument` (named `POPrintDoc`).

4. Define a module-level variable named `ReportString` to the code of `MainForm` and then add an event handler for the `Click` event of the File ⇨ Print menu item. Add the `Click` event of the Print toolbar icon as well so both are intercepted by the same event.

5. In the routine, first initialize the `ReportString` to be empty in case it has been used previously and then call the function to extract the information from the database. If the `ReportString` that is returned has data, then assign the `POPrintDoc` to the `Document` property of the `PrintDialog` object and show the dialog itself. Finally, if the user clicks the OK button on the dialog window, start the printing process by using the `POPrintDoc`'s `Print` method:

```
Private Sub printToolStripMenuItem_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles printToolStripMenuItem.Click, _
    printToolStripButton.Click
    ReportString = vbNullString
    ReportString = GenerateReport(mCurrentUserID)

    If ReportString <> vbNullString Then
        With prnDialog
            .Document = POPrintDoc
```

```
                  If .ShowDialog() = Windows.Forms.DialogResult.OK Then
                      POPrintDoc.Print()
                  End If
            End With
        End If
    End Sub
```

6. Do a similar thing for the File ➪ Print Preview menu item, but this time use the `PrintPreviewDialog` object you added instead:

```
Private Sub printPreviewToolStripMenuItem_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles printPreviewToolStripMenuItem.Click
    ReportString = vbNullString
    ReportString = GenerateReport(mCurrentUserID)

    If ReportString <> vbNullString Then
        With prnprvDialog
            .Document = POPrintDoc
            .ShowDialog()
        End With
    End If
End Sub
```

7. The only thing left to do now is create an event handler routine for the `PrintPage` event. This enables you to do the printing that is required. If you were certain that everything could fit on one page, you could just draw each line of the report to the printer regardless of how big it is. Unfortunately, you don't have that luxury because the report of the people in the database could be quite lengthy.

This means you'll need to calculate the size of each line as you go; and if it won't fit, stop the printing at that point and set the `HasMorePages` property to `True` so that another `PrintPage` event is raised. You need to keep track of what line you're up to in the report, so define a `Static` integer variable to store the `CurrentLinePosition`. `Static` is a special variable context that keeps the variable's value between function calls, so if you set it to 5 at the end of one `PrintPage` event process, the next time `PrintPage` is called the value will still be 5:

```
Private Sub POPrintDoc_PrintPage(ByVal sender As Object, ByVal e As _
    System.Drawing.Printing.PrintPageEventArgs) Handles POPrintDoc.PrintPage
    Static CurrentLinePosition As Integer

End Sub
```

8. Retrieve the current page settings so you know what the printable area is. The `PrintDocument` class has a property that stores this information — `DefaultPageSettings`. This object has the total size of the paper itself, as well as the settings for each of the margins, so the printable area is the total size minus the margins. You need this information stored in a special object structure called a `RectangleF` to pass into the graphic methods used by the `PrintDocument` object:

```
Dim PrintAreaHeight As Integer
Dim PrintAreaWidth As Integer
Dim LeftMargin As Integer
Dim TopMargin As Integer
With POPrintDoc.DefaultPageSettings
    PrintAreaHeight = .PaperSize.Height - .Margins.Top - .Margins.Bottom
```

```
        PrintAreaWidth = .PaperSize.Width - .Margins.Left - .Margins.Right
        LeftMargin = .Margins.Left
        TopMargin = .Margins.Top
    End With
    Dim PrintingArea As New RectangleF(LeftMargin, TopMargin, PrintAreaWidth, _
        PrintAreaHeight)
```

9. You also need to create a `StringFormat` object so that when you call the `Measure` and `Draw` functions, they know what action to take for the string. In this case, set it to `LineLimit`, which restricts the printing process to draw only whole lines. If a line can be only partially drawn, it is excluded from the function:

```
Dim PrintingFormat As New StringFormat(StringFormatFlags.LineLimit)
```

10. Next, you need to split the contents of `ReportString` into an array of `String`s representing each line of the report. You also need some local variables to store the number of lines and characters printed:

```
Dim NumberOfLinesFilled As Integer
Dim NumberOfLinesPrinted As Integer
Dim NumberOfCharactersToPrint As Integer
Dim ReportLines() As String = Split(ReportString, vbCrLf)
```

11. Loop through the array of strings. The idea now is that your code processes each element of the array until it cannot fit one into the printable area that's left. At that point, it needs to exit the loop and determine whether more lines remain to be printed:

```
For ReportCounter As Integer = CurrentLinePosition To ReportLines.GetUpperBound(0)
    ... determine the number of lines to be printed for this entry in the array
    If NumberOfLinesFilled > 0 Then
        ... the actual printing goes here
    Else
        Exit For
    End If
Next

CurrentLinePosition += NumberOfLinesPrinted
If CurrentLinePosition < ReportLines.GetUpperBound(0) Then
    e.HasMorePages = True
Else
    e.HasMorePages = False
    CurrentLinePosition = 0
End If
```

Notice that you need to reset the `CurrentLinePosition` to zero if you've reached the end of the printing process for this document.

12. You have two types of report lines in this report: normal detail lines and heading lines. They differ in their font settings, so you need to find the different types and create different font objects for each. The heading lines are prefixed with a special code, `$HDG`, which you created in the `GenerateReport` function. Look at each report line for that string, and if the prefix is found, strip it off using `Substring` and create a heading style font of Tahoma, 18-point Bold. If the report line is a detail line, create a detail style font of Times New Roman, 12-point Normal:

```
Dim PrintFont As Font

If ReportLines(ReportCounter).Length > 4 AndAlso _
    ReportLines(ReportCounter).Substring(0, 4) = "$HDG" Then
    ReportLines(ReportCounter) = ReportLines(ReportCounter).Substring(4)
    PrintFont = New Font("Tahoma", 18, FontStyle.Bold)
Else
    PrintFont = New Font("Times New Roman", 12)
End If
```

13. Now that you have the font and other settings, you can determine whether the line will fit in the printable area that's left. Call the `MeasureString` method of the `Graphics` object. This method accepts a number of parameters:

❑ **Text** — The actual string to measure

❑ **Font** — The font style that will be used to draw the font

❑ **LayoutArea** — A defined rectangle of area to which the text is restricted

❑ **Format** — The formatting rules to use when determining what fits

❑ **Characters** — The number of characters that fit into the area

❑ **Lines** — The number of lines that fit into the area.

It also returns a `SizeF` structure that defines the exact rectangle of space used by the text, given the parameters that were passed to the method:

```
Dim SizeNeeded As SizeF = e.Graphics.MeasureString(ReportLines(ReportCounter), _
    PrintFont, New SizeF(PrintAreaWidth, PrintAreaHeight), PrintingFormat, _
    NumberOfCharactersToPrint, NumberOfLinesFilled)
```

14. If `MeasureString` returns a `NumberOfLinesFilled` value of more than zero, then you know that the report line can be printed. Use a very similar function called `DrawString` to do the actual printing of the text:

```
e.Graphics.DrawString(ReportLines(ReportCounter), PrintFont, _
    Brushes.Black, PrintingArea, PrintingFormat)
```

15. Once you've printed the current line, you need to change the printable area for the next element in the `ReportLines` array. This uses the `SizeNeeded Height` property from the `MeasureString` method. To complete the logic, you need to increment the `NumberOfLinesPrinted` variable so that you know how many lines have been printed in this event:

```
PrintAreaHeight -= CType(SizeNeeded.Height, Integer)
TopMargin += CType(SizeNeeded.Height, Integer)
PrintingArea = New RectangleF(LeftMargin, TopMargin, PrintAreaWidth, _
    PrintAreaHeight)
NumberOfLinesPrinted += 1
```

16. Save your progress and run the application. Select the Print Preview command from the File menu to display the report. A sample is shown in Figure 11-3.
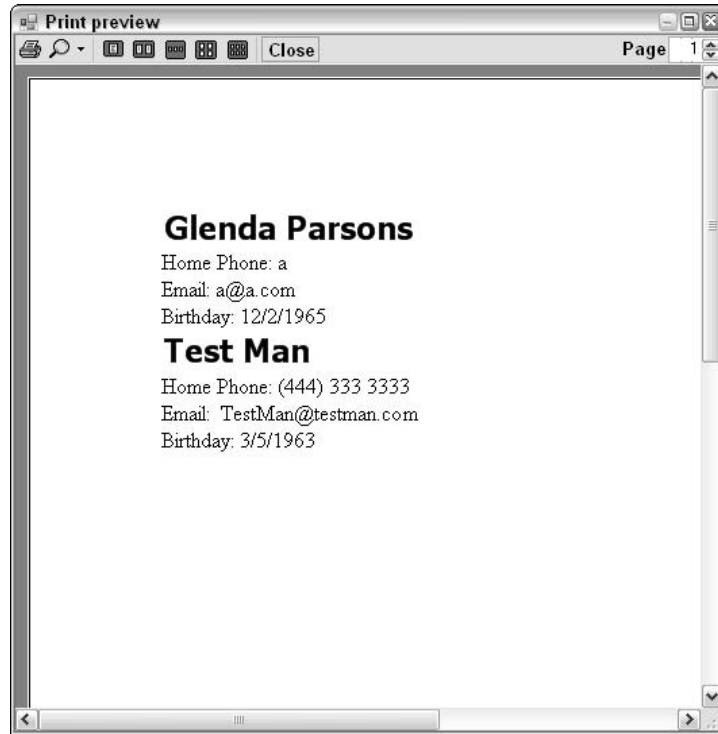
Figure 11-3

System Components

Many other components and features in Visual Basic Express help round out the application. Many of these components do not have a visible aspect to them, or if they do, rather than have a separate view-able control that is sited on the form, they display information in relation to another control.

Rather than go through each of these components in great theoretical detail, the following provides a short list of the top four components (besides the ones you've already learned about, of course) that you might find useful in your applications:

❑ **ErrorProvider** — The `ErrorProvider` control sits invisibly in your system tray until you tell it to give feedback to the user about a particular control on the form that is in error. You specify the text that is to be displayed in a tool tip and the kind of icon that should be displayed next to the control.

   In addition, you can specify how fast and long the icon should blink to attract the user's attention and where it should be positioned in relation to the control.

❑ **FileSystemWatcher** — This clever component enables you to monitor a folder or individual file for changes. When a change has occurred, it raises an event that you can then trap with an event handler routine. It distinguishes between additions, deletions, and updates to files.

❑ **HelpProvider** — The `HelpProvider` extends other controls, adding additional properties to each control on the form or user control design surface. At design time, you can access these properties through the Properties window.

   The properties identify how the application should respond to a request for help when the particular fields are being displayed. You can link to a compiled help file or a Web page, or simply display a tool tip containing the help information.

❑ **ImageList** — If your application uses many icons or images, you might benefit from compiling them all into a single `ImageList`. Once they're loaded into this control, you can retrieve each image as needed from the `Images` collection.

   This control is also used for many other controls that use a series of images, such as the `TreeView`.

In addition to these are a number of system-related components not normally used in most basic applications. Items such as the Windows system message queues, performance counters, and Active Directory entries can all be accessed via components available to you in Visual Basic Express.

You should also keep an eye out for additional properties on the standard controls that make your application function better. For example, the `TextBox` control enables you to specify some AutoComplete options to help your users enter the information you're after.

Finally, the .NET Framework is full of classes that help you implement functionality into your application without you needing to worry about how it's being done beneath the hood. For example, the `System.Net.Mail` namespace has a number of classes and methods that enable you to send e-mail messages from your program. For a longer discussion on the types of classes and objects that are available to Visual Basic Express, refer to Appendix B, which covers the .NET Framework.

In the next Try It Out, you'll use several of these components and see how the AutoComplete properties work in the `TextBox` control to add some helpful functionality to the `PersonalDetails` control in your Personal Organizer application. You'll also create a function to send e-mail messages to selected people in the `PersonList` control so you can see how the `System.Net.Mail` namespace works.

Try It Out   Using System Components

1. Return to Visual Basic Express and your Personal Organizer project. The first thing to do is add some help and validation to the `PersonalDetails` control so users know what is expected of them, so open `PersonalDetails.vb` in DesignvView.

2. Add a `HelperProvider` component to the form and name it `helpPersonalDetails`. This extends each visible component in the control with additional properties that are accessible through the Properties window (see Figure 11-4).

3. Select `txtFirstName` and scroll to the new properties. Set the following properties:

   ❑ **HelpString** — Enter the first name of the person here

   ❑ **ShowHelp** — `True`

4. Set the same properties on each control so that when the user has focus on that particular control and presses F1, a helpful tool tip will be displayed. You can even include help information on buttons.
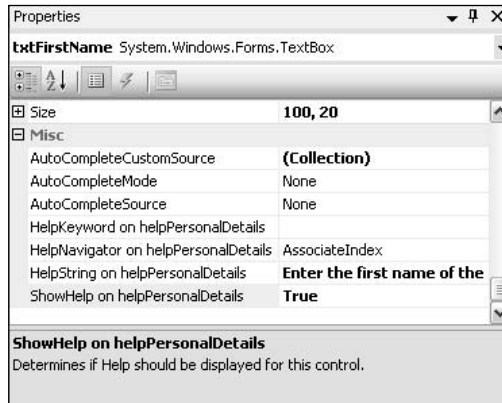
Figure 11-4

5. Add an `ErrorProvider` component to the form and name it `errorPersonalDetails`. If you like, you can change the icon to something you prefer over the default red exclamation mark, but for this Try It Out, it is left as the default. To allow space for the error icon for the last name (you're going to make it a required field), reduce the width of `txtLastName` slightly. To keep the design consistent, you should also reduce the width of the other fields so they all align along the right-hand side.

6. Add an event handler routine for the `Validating` event for the `txtFirstName` control. First you need to set up the error icon alignment and padding so that Visual Basic Express can position the icon correctly. After that, check the `Text` property of the `TextBox`, and if it's empty, call the `SetError` method, passing in the control that is in error (`txtFirstName`) and the error text. Note that you'll have to reset the error text to empty if you want the error to be cleared:

```
Private Sub txtFirstName_Validating(ByVal sender As Object, _
  ByVal e As System.ComponentModel.CancelEventArgs) Handles txtFirstName.Validating
    With errorPersonalDetails
        .SetIconAlignment(Me.txtFirstName, ErrorIconAlignment.MiddleRight)
        .SetIconPadding(Me.txtFirstName, 2)
        If txtFirstName.Text = vbNullString Then
            .SetError(Me.txtFirstName, "First Name is required.")
        Else
            .SetError(Me.txtFirstName, "")
        End If
    End With
End Sub
```

Repeat this process for `txtLastName`, making sure you're referencing the correct object in the `SetError` methods:

```
Private Sub txtLastName_Validating(ByVal sender As Object, _
  ByVal e As System.ComponentModel.CancelEventArgs) Handles txtLastName.Validating
    With errorPersonalDetails
        .SetIconAlignment(Me.txtLastName, ErrorIconAlignment.MiddleRight)
        .SetIconPadding(Me.txtLastName, 2)
        If txtLastName.Text = vbNullString Then
            .SetError(Me.txtLastName, "Last Name is required.")
```

```
        Else
            .SetError(Me.txtLastName, "")
        End If
    End With
End Sub
```

7. Because you have made both the first and last names required fields with the `ErrorProvider`, you should also check to make sure they're valid before the intended functionality in the Save button's Click event routine executes. Locate the `ButtonClickedHandler` routine you created previously and change the code for the Save button so it validates the fields first, and only if the fields are valid does it continue.

    If the fields are found to be invalid, then it displays a message and positions the cursor on the first field that is in error:

```
Private Sub ButtonClickedHandler(ByVal sender As System.Object, _
ByVal e As System.EventArgs)

    Dim btnSender As Button = CType(sender, Button)
    If btnSender.Name = "btnSave" Then
        If Me.ValidateChildren() = True Then
            RaiseEvent ButtonClicked(1)
        Else
            MessageBox.Show("Please enter the first and last names")
            If txtFirstName.Text = vbNullString Then
                txtFirstName.Focus()
            Else
                txtLastName.Focus()
            End If
        End If
    ElseIf btnSender.Name = "btnCancel" Then
        RaiseEvent ButtonClicked(2)
    End If
End Sub
```

8. The other helpful feature you'll implement is a selected items list for the Favorites `TextBox`. Rather than let users guess what they should enter in this field, you can set the `AutoComplete` properties so that users get some visual cues as they enter information into this field. In Design view, set the `AutoCompleteSource` property to `CustomSource` and the `AutoCompleteMode` property to `SuggestAppend`.

9. This will tell Visual Basic Express to look for an associated custom-built list of text items to suggest to users as they type. It will display the list (`Suggest`) and append the first item that matches what the user has typed so far in the `TextBox` (`Append`). Click the ellipsis button on the `AutoCompleteCustomSource` property and create a list of strings that Visual Basic Express can use as suggestions (see Figure 11-5).

10. Run the application and add a new person. Notice how the errors are indicated by the `ErrorProvider` when the text fields do not match what's required; in addition, note that the Favorites `TextBox` contains suggested items as you type (see Figure 11-6).

Figure 11-5

11. To finish the Personal Organizer project for this chapter, you'll add e-mail capabilities to the `PersonList` form. Stop the application and add a new Windows Form to the project via the Project ⇨ Add Windows Form menu command. Name the new form `POMessage` and set the following properties:

   ❑ **FormBorderStyle** — `FixedDialog`

   ❑ **Text** — Send Email
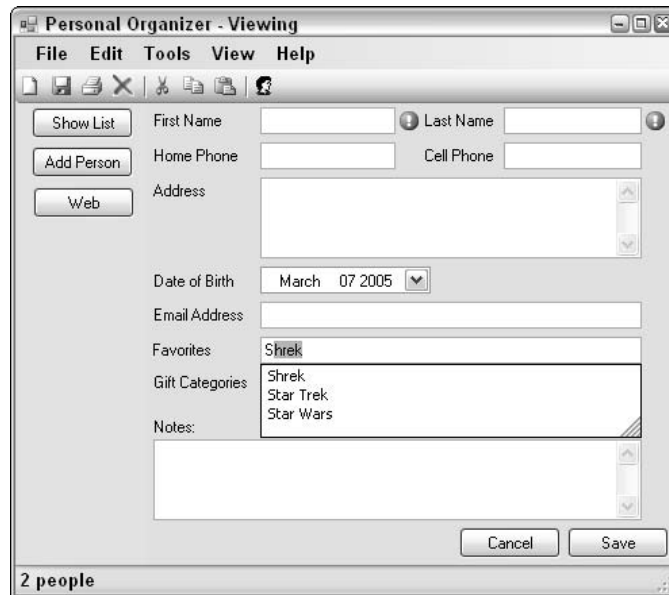
Figure 11-6

12. Add four `Labels` and four `TextBoxes` to the form along with two `Buttons` and lay them out as shown in Figure 11-7. Name the `TextBoxes` according to the content they will have and set the `ReadOnly` property of the From and To `TextBoxes` to `True`, as this information will be populated from the `PersonList` form.
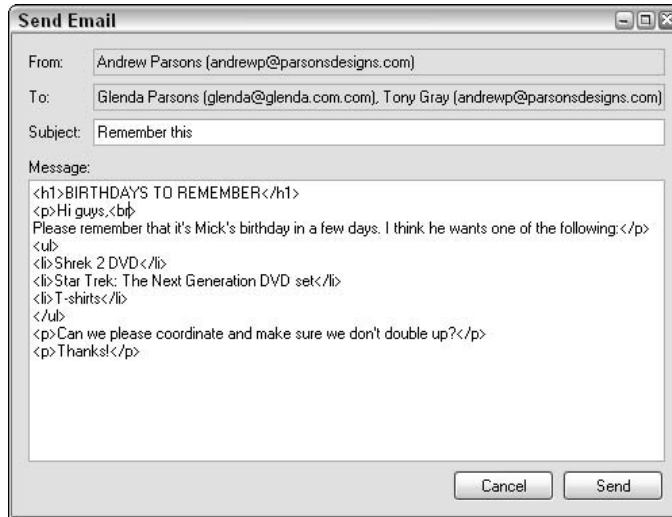
Figure 11-7

13. Switch to code view and, because you'll be using a lot of classes within the `System.Net.Mail` namespace, add an `Imports` statement at the top of the class to shortcut the e-mail–related objects:

```
Imports System.Net.Mail
```

14. Create two properties for the sender e-mail address and the recipient e-mail list. Note that because you can send an e-mail message to multiple people, you must use the `MailAddressCollection` object to store the list of addresses. In the `Set` clause for each of the properties, set the `Text` property of the corresponding `TextBox` so the user knows what information is being used:

```
Private mFromAddress As MailAddress
Private mToAddresses As MailAddressCollection
Public Property FromAddress() As MailAddress
    Get
        Return mFromAddress
    End Get
    Set(ByVal value As MailAddress)
        mFromAddress = value
        txtFrom.Text = mFromAddress.DisplayName & " (" & mFromAddress.Address & ")"
    End Set
End Property
Public Property ToAddresses() As MailAddressCollection
    Get
        Return mToAddresses
    End Get
    Set(ByVal value As MailAddressCollection)
        mToAddresses = value
        For Each ToAddress As MailAddress In mToAddresses
            txtTo.Text &= ToAddress.DisplayName & " (" & ToAddress.Address & "), "
        Next
```

```
            txtTo.Text = txtTo.Text.Remove(txtTo.Text.Length - 2, 2)
        End Set
    End Property
```

15. Add an event handler routine for the Cancel button's `Click` event to close the form (use `Me.Close`), and then create another event handler routine for the Send button's `Click` event. You'll need to create a new `MailMessage` object and then populate its properties. Setting the `IsBodyHtml` property to `True` enables the message to include formatted HTML if the user desires.

Once the e-mail message has been created, complete with `From`, `To`, `Subject`, and `Body` properties all set, you must create a new instance of the `SmtpClient` object that is used to send e-mail via the Simple Mail Transfer Protocol (SMTP), which almost all Internet providers use for e-mail services. The only property you usually need to set is the `Host` property. Make this the same as what you use in your regular e-mail program; and once it is set, you simply call the `Send` method to send the e-mail message you created:

```
Private Sub btnSend_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnSend.Click
    Dim POMessage As New Net.Mail.MailMessage()
    With POMessage
        .From = mFromAddress
        .To.Clear()
        For Each ToAddress As MailAddress In mToAddresses
            .To.Add(ToAddress)
        Next
        .Subject = txtSubject.Text
        .Body = txtMessageBody.Text
        .IsBodyHtml = True
    End With

    Dim MyMailServer As New SmtpClient()
    With MyMailServer
        .Host = "smtp.yourhost.here.com"
        .Send(POMessage)
    End With
    MessageBox.Show("Message sent")
    Me.Close()
End Sub
```

Please note that if you are going to give this program to someone else, they might not have access to the same mail server as you, so you might need to allow the `Host` property to be configured, as opposed to hardcoding it as shown in this Try It Out.

16. The `Email Form` is now ready; all you need to do is show it with the e-mail addresses of the people selected in the `PersonList` control. Open the `PersonList` control in Design view and add a third button underneath the other two. Name it `btnSendEmail` and change its `Text` property to `Send Email`.

17. Double-click the new button to automatically create a `Click` event handler routine. First check whether the `SelectedItems` collection of the `Listbox` contains any items. If it does, then you should create a new `MailAddress` object containing the e-mail information about the sender (again, this is hardcoded in this Try It Out, but you could make this configurable in your application if you're giving it to other people) and create a new `MailAddressCollection` to store each of the people selected:

```
Private Sub btnSendEmail_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnSendEmail.Click
  If lstPersons.SelectedItems.Count > 0 Then
    Dim FromAddress As New System.Net.Mail.MailAddress("MyEmail@email.com", _
      "This is me")
    Dim ToAddresses As New System.Net.Mail.MailAddressCollection
  End If
End Sub
```

18. Retrieve the contents of the `Person` table from the database and compare each row to the `SelectedItems` collection, much like you did for the Delete Selected button. This time, instead of deleting the record when you find a match, create a new `MailAddress` object with the `EmailAddress` from the database and the `DisplayName` from the `Person` object and then add it to the `MailAddressCollection`:

```
Dim PersonListAdapter As New _PO_DataDataSetTableAdapters.PersonTableAdapter
Dim PersonListTable As New _PO_DataDataSet.PersonDataTable
PersonListAdapter.Fill(PersonListTable)

For Each CurrentPersonRow As _PO_DataDataSet.PersonRow In PersonListTable.Rows
    For Each objPerson As Person In lstPersons.SelectedItems
      If CurrentPersonRow.ID = objPerson.ID Then
          If CurrentPersonRow.EmailAddress.Trim <> vbNullString Then
              Dim ToAddress As New System.Net.Mail.MailAddress( _
                CurrentPersonRow.EmailAddress, objPerson.DisplayName)
              ToAddresses.Add(ToAddress)
          End If
          Exit For
      End If
    Next
Next
```

If the `ToAddresses` collection has any e-mail address objects, then create a new instance of the `POMessage` form, set the `FromAddress` and `ToAddresses` properties, and then show it. The `POMessage` form does the rest of the work:

```
If ToAddresses.Count > 0 Then
    Dim frmSendEmail As New POMessage
    With frmSendEmail
        .FromAddress = FromAddress
        .ToAddresses = ToAddresses
        .ShowDialog()
    End With
End If
```

19. Go ahead and run the application and display the person list. Select a couple of the entries and then click the Send Email button to display the `POMessage` form. Enter a subject line and some text in the body and click Send. If you enter HTML tags as part of the text, the e-mail message will be correctly formatted when the recipients receive it (see Figure 11-8).
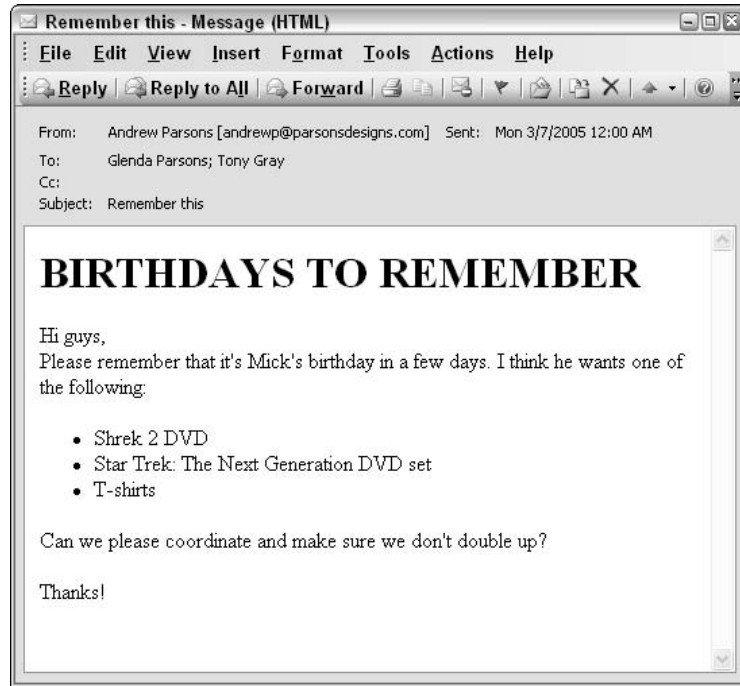
Figure 11-8

Summary

Using the techniques discussed in this chapter, you can begin to add the finishing touches to your applications. When you are building a program, the basics of a good user interface and efficient coding to access database information might be enough to get the job done, but it's the little things that separate the amateur from the professional.

The little things begin with making sure you provide information to the user as much as possible without stopping them from working — status bars, error messages, and help information all assist with this. Notification icons can convey information to the user without your application needing to have focus, and the printing capabilities of Visual Basic Express are detailed enough that you can produce pretty much anything you need.

In this chapter, you learned to do the following:

❑ Use timers and notification icons to send information to the user

❑ Print information either to paper or to a print preview dialog

❑ Display contextual error indicators and help information.

Exercises

1. Customize the printing code so that it prints the list of people only if the `Person List` control is showing. Add another report to display information about the currently selected person if individual details are shown.

2. Add two elements to the `StatusStrip` at the bottom of the `PersonalOrganizer`'s main form, a `StatusLabel` and a `ProgressBar`. Keep the `StatusLabel` up to date with the number of people currently in the database for the current user and use the progress bar to indicate how much of the report has been generated when it is processing the person list.

12

Using XML

When Microsoft first designed .NET, they realized that they needed to use as many open standards for the different components as possible. As a result, even when they created a new language, C#, they put it through the standards process to have it certified. However, the most important aspects of their program to follow the open standard were the bits that could interface with other applications and environments.

For those, Microsoft turned to a technology called *Extensible Markup Language*, or *XML*. XML is used to format the communication documents created to talk to other programs. Web services like those you accessed in Chapter 9 use XML to format the request sent to the web service and to store the response. SQL databases can be easily exported to XML, and numerous other parts of .NET also use XML to format the data. Because Visual Basic Express is based on .NET, it has the capability to use all of these XML components, as you'll see over the next several pages.

In this chapter, you learn about the following:

❑　What XML is and how you can use it in your programs

❑　How databases can export and import their data via XML

❑　The XML objects available in Visual Basic Express

So What Is XML?

As stated in the introduction, XML stands for Extensible Markup Language (some people write this as eXtensible Markup Language to highlight where the *X* came from). A *markup language* is a way to format data so that it contains information that describes what the data is for and how it should be used. Each part of the document is marked with *tags* that contain attributes identifying specific properties about the data enclosed in the tags.

XML is not the only markup language that you might encounter. In fact, the entire web is based on another markup language—Hypertext Markup Language (HTML), which looks very similar to XML. Consider the following two files:

```
<HTML>                                <config version="1.0" time="12.20">
 <HEAD>                                <Values>
  <TITLE>My Web Page</TITLE>            <Setting>Value</Setting>
 </HEAD>                                <Setting>123</Setting>
 <BODY>                                </Values>
  <H1>A Heading</H1>                   <State>
  <P>This is normal text.</P>           <User Login="true">Andrew</User>
  <A HREF="link">My link</A>           </State>
 </BODY>                              </config>
</HTML>
```

The one on the left is a simple web page written in HTML, including the title, a heading, a paragraph, and a hyperlink. The file on the right is an XML file containing a number of properties for a log file. Both contain a series of values enclosed in matching tags to identify the type of data that is represented. The big difference between the two is that HTML is a highly specialized markup language aimed at a particular purpose—to describe the format of a web page. XML, on the other hand, is a generic language designed to describe any kind of data.

XML has no predefined tags like those in HTML. Instead, most XML files are defined by a definition document of some kind. There are two kinds of definition files: Document Type Definition (DTD) and XML Schema Documents (XSDs). To keep this discussion brief, the focus here will be on XSD definitions because they are usually used when using XML in .NET.

Your XML file does not require an XSD to accompany it, but if it contains more than a couple of values and you're depending on the XML contents following a set structure so that you can read it in your application, you are better off using an XSD to keep the data in order.

This is because most XML processing systems, including the one that comes with Visual Basic Express, can validate the XML data against the XSD and produce errors that you can check if the data is not valid.

The makeup of an XML file is straightforward. Each matching pair of tags is called an *element* or a *node*. Therefore, in the earlier sample XML file, you have a config node, which contains a Values node and a State node. The Values node in turn contains two Setting nodes. The information between the opening and closing tag is known as the *value*. The two Setting nodes have the values Value and 123, respectively. Finally, within the opening tag of an element can be a number of properties that belong to the node; these are known as *attributes*. The User node has an attribute of Login with a value of True. The whole thing is called an *XML document*.

Besides this simple structure, you have to follow some basic rules when creating an XML file. When you are using the classes and methods in Visual Basic Express, it does most of the work for you, but you can still produce invalid XML data if you don't follow these guidelines.

First, you must have only one *root element* that contains the rest of the XML document within its opening and closing tags. While you could define multiple nodes at the top level and read them using custom-built programming logic, the XML standard specifies that there be only one.

You must include the closing tag in the pair. HTML and other markup languages sometimes allow you to omit the closing tag, and implicitly assume them, but XML is stricter than that. If the XML node doesn't contain any data, you can shortcut the opening and closing tag by closing the node off in the first tag with a single slash (/). For example, the following two lines are considered identical by an XML processor:

```
<myNode></myNode>
<myNode />
```

Unlike Visual Basic Express variables and class names, XML tags are case sensitive. Therefore, if your opening tag is called `<MyTaGrUlEs>`, you must close the pair with exactly the same case — `<mytagrules>` won't cut it.

When attributes are defined within a XML tag, the values must be enclosed in quotation marks, even when the values are numeric or single words. HTML allows you to omit the quotation marks in these simple-value cases.

Extensible Means Just That

One great advantage of XML is that you can extend the data definition without breaking your application. This is because the application can still find the nodes it used prior to the data change. For example, the original XML definition used by your application is as follows:

```
<config version="1.0" time="12.20">
  <Values>
    <Setting>Value</Setting>
    <Setting>123</Setting>
  </Values>
  <State>
    <User Login="true">Andrew</User>
  </State>
</config>
```

Your program uses the `Setting` nodes and the `User` node to display some information on a form. Now, the other program that created the XML file is extended and includes additional information:

```
<config version="1.0" time="12.20">
  <Values>
    <Setting>Value</Setting>
    <Setting>123</Setting>
  </Values>
  <State>
    <User Login="true">Andrew</User>
    <File Overwrite="true">C:\Temp\MyLog.txt</File>
  </State>
</config>
```

Your code would still be able to access the `Setting` nodes and the `User` node without needing to know anything about the new data being stored in the file. The same thing applies to additional attributes in a node.

When referring to nodes, XML uses a family-oriented nomenclature. This enables you to easily determine how nodes relate to each other. The node that owns another is known as the *parent element* of the other node, while the one that is owned is the *child element* of the first. Nodes that are on the same level within a single parent element are called *siblings*, or sometimes *sister elements*.

To illustrate this, in the case of the sample XML that you've been looking at, the `Setting` nodes are child elements of the `Values` node, and the `State` node is the parent element of the `User` and `File` nodes. `User` and `File` are siblings of each other but are not siblings of the `Setting` nodes.

XML Attributes

Each XML element can have its own attributes. Again, these attributes can be controlled by a definition file so that only allowed property names and values can be included, but because you usually own the definition file as well, you can dictate which attributes you want to have defined.

The first line of the sample XML file defines the root element. It has a name of `config` and two attributes — `version` and `time`. As mentioned earlier, every attribute value must be enclosed in quotation marks. XML allows you to use either single or double quotes, so both `version="1.0"` and `version='1.0'` are deemed acceptable.

Usually attributes are used by the program to determine what to do with the data stored within the XML element. The `User` node has a value of `Andrew` and an attribute of `Login` with a value of `true`. The application could use this information to determine that the user involved in the process was named Andrew, and that he was logged into the system at the time. The `Login` attribute wasn't necessary to identify the User, but provided additional information that the program could use.

There is no hard-and-fast rule about when to store information in an attribute, when to use a child element, or when to include the data in the value component of the element. The `User` node could be rewritten as follows:

```
<User>
  <Login>true</Login>
  <Name>Andrew</Name>
</User>
```

It could even have been defined with what is known as *mixed content* (whereby the element has a value and child elements) like so:

```
<User>Andrew
  <Login>true</Login>
</User>
```

Validating Data

An XML Schema Document, known as a XSD, is a definition file used to determine whether the XML data is *valid*. You can have an XML file that is *well formed*, a term used to indicate that all nodes have their opening and closing tags, attributes are properly defined, and so on, that is still not valid. A valid XML document is one that conforms to a data definition — either a DTD or an XSD.

Each element within the XML must be defined in the schema; otherwise, the XML document is considered invalid. The XML file that's been used as an example could be defined with the following XSD:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="config" type="configType"/>
  <xs:complexType name="configType">
    <xs:sequence>
      <xs:element name="Values" type="ValuesType" maxOccurs="1"/>
      <xs:element name="State" type="StateType" maxOccurs="1"/>
    </xs:sequence>
    <xs:attribute name="version" type="xs:string" use="required"/>
    <xs:attribute name="time" type="xs:string" use="required"/>
  </xs:complexType>

  <xs:complexType name="ValuesType">
    <xs:sequence>
      <xs:element name="Setting" type="xs:string" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="StateType" mixed="true">
    <xs:sequence>
      <xs:element name="User" type="xs:string" />
    </xs:sequence>
    <xs:attribute name="Login" type="xs:boolean" use="optional"/>
  </xs:complexType>
</xs:schema>
```

You might have noticed that the XSD itself looks like XML, and that's because it is. XSD files must conform to their own data definition layout specified in the standard for XML schema. In fact, this sample XSD file contains the location of the namespace that defines its own structure — `http://www.w3.org/2001/XMLSchema`.

When you look through this schema, each element can be seen as an `xs:element` node that has attributes describing its use and type. For example, the `config` node has a type of `configType`, which is then defined in the following lines in the file.

While this book isn't aimed at teaching you XML, the previous discussion should serve to help you get a basic understanding of how it works so you can look at the way Visual Basic Express uses XML and takes advantage of it. If you need to know more, you can find plenty of resources for writing XML and XSDs, including Beginning XML, *3rd Edition, by David Hunter et al. (Wiley, 2004).*

Databases and XML

One feature of Visual Basic Express is its capability to export information stored in a SQL Server database to an XML file. This can then be accessed by other applications that do not have access to your database. You can also populate a database table from XML files, too.

Before you look at the main XML objects found in Visual Basic Express, these capabilities to convert SQL Server data to and from XML should help you understand how an XML file might be used in your own applications.

The `DataTable` class has two methods — `ReadXml` and `WriteXml` — both with multiple definitions:

❑ `ReadXml` is the simplest because it just needs to know where to get the data from and then works out the rest. The different definitions of `ReadXml` simply take different parameters to indicate the data source.

   The syntax of `ReadXml` is `MyTable.ReadXml(DataSource)`, where `DataSource` can be a filename, an `IOStream`, an `XMLReader` or a `TextReader`. The filename is the most basic and easiest to use. If you try to read XML data that does not meet the `DataTable`'s own definition, an exception is raised. Otherwise, the `DataTable` contents are replaced with the information stored in the XML file.

❑ The `WriteXml` method of the `DataTable` object has an overwhelming number of overloaded definitions. Overwhelming, that is, until you realize they are just variations on a theme. In fact, you have only a few options, but each can be used in conjunction with a different set of other parameters. The first parameter defines the type of output object that will be written to. This is similar to the parameter of `ReadXml` — `IOStream`, `TextWriter`, `XMLWriter`, or filename. In addition to this are two optional parameters, a `Hierarchy` Boolean value and a `WriteMode` value.

   The `Hierarchy` flag dictates whether the `WriteXml` command includes all child tables or just the main table that the `DataTable` object contains. This could be useful if you have a single `DataTable` object with a collection of tables stored within it.

   The `WriteMode` tells the `WriteXml` what information to include with the actual data of the table. When Visual Basic Express creates the XML file, it can include the data as is — this is, the default behavior. However, you can also specify that it should include an XSD along with the data so that any application reading the XML knows how to validate it and what each element is supposed to contain.

   Finally, you can specify a `WriteMode` of `DiffGram`. This tells `WriteXml` to write only the parts of each row in the table that have changed. This can be useful for logging database changes because it excludes any records of information that have not changed since the last database update.

To confirm the information just discussed, the following Try It Out adds export and import functionality to the Personal Organizer application using XML data files.

Try It Out　　Exporting and Importing XML

1. Start Visual Basic Express and open the Personal Organizer application project you've been working on. If you don't have an up-to-date version of the project, you can find one in the `Code\Chapter 12\Personal Organizer Start` folder of the code you downloaded from `www.wrox.com`.

   You have two functions to implement: exporting the data from the database into an XML file and importing an XML file back into the database. The first feature is quite straightforward to implement, with only one gotcha to be aware of, but importing has a number of other issues that you'll see in a moment.

2. Open the `GeneralFunctions.vb` module and create a new function called `ExportPOData` that returns a `Boolean` to indicate success or failure. Give it parameters of a `UserID Integer` and `ExportDataLocation` as a `String`. Add a statement to return `True` at the end of the function:

```
Public Function ExportPOData(ByVal UserID As Integer, _
    ByVal ExportDataLocation As String) As Boolean

    Return True
End Function
```

3. The function accepts only one filename — the location for storing the `Person` data — but you want to store the `POUser` table as well. Therefore, create an additional filename from the parameter by changing the file extension:

```
Public Function ExportPOData(ByVal UserID As Integer, _
    ByVal ExportDataLocation As String) As Boolean
    Dim POUserLocation As String
    POUserLocation = ExportDataLocation.Remove(ExportDataLocation.Length - 3, 3) &
"pou"
    Return True
End Function
```

4. Before you can do the export, you should determine whether the files exist already, and, if so, delete them. The `My.Computer.FileSystem` object works well here:

```
        With My.Computer.FileSystem
            If .FileExists(ExportDataLocation) Then .DeleteFile(ExportDataLocation)
            If .FileExists(POUserLocation) Then .DeleteFile(POUserLocation)
        End With
```

5. Now you're ready for the export functionality. To get the data ready, you need to create a `DataAdapter` and a `DataTable` and then use the `Fill` method to populate the `DataTable`. You learned how to do this in Chapter 7. Once the table contains data, the only additional command required is the `WriteXml` method on the `DataTable` object. Therefore, to export the contents of the `Person` table, you could write the following code:

```
        Dim GetPersonAdapter As New _PO_DataDataSetTableAdapters.PersonTableAdapter
        Dim GetPersonTable As New _PO_DataDataSet.PersonDataTable
        GetPersonAdapter.Fill(GetPersonTable)
        GetPersonTable.WriteXml(ExportDataLocation)
```

This version of the `WriteXml` method has a flaw, however. Because it doesn't include any definition information about the data stored in the XML file, any fields in the table that do not contain values will not be included in the XML. This might be okay if you want to send the file to some other application, but because you want to be able to import it directly into the database tables in your own application, you'll get errors about missing fields.

`WriteXml` has a number of different versions that enable you to include additional information — including the schema definition of the database table. This is the XSD structure you saw earlier in this chapter. To include the schema, alter the `WriteXml` call to include an additional parameter of `XmlWriteMode.WriteSchema`. When you've done this for both the `Person` and `POUser` tables, your `ExportPOData` function is complete:

```
Public Function ExportPOData(ByVal UserID As Integer, _
    ByVal ExportDataLocation As String) As Boolean
    Dim POUserLocation As String
    POUserLocation = ExportDataLocation.Remove(ExportDataLocation.Length - 3, 3) &
"pou"
    With My.Computer.FileSystem
        If .FileExists(ExportDataLocation) Then .DeleteFile(ExportDataLocation)
        If .FileExists(POUserLocation) Then .DeleteFile(POUserLocation)
    End With
    Dim GetPersonAdapter As New _PO_DataDataSetTableAdapters.PersonTableAdapter
    Dim GetPersonTable As New _PO_DataDataSet.PersonDataTable
    GetPersonAdapter.Fill(GetPersonTable)
    GetPersonTable.WriteXml(ExportDataLocation, XmlWriteMode.WriteSchema)

    Dim GetUserAdapter As New _PO_DataDataSetTableAdapters.POUserTableAdapter
    Dim GetUserTable As New _PO_DataDataSet.POUserDataTable
    GetUserAdapter.Fill(GetUserTable)
    GetUserTable.WriteXml(POUserLocation, XmlWriteMode.WriteSchema)
    Return True
End Function
```

6. To enable users to run this function, open the `MainForm` in Design view. Add a `SaveFileDialog` to the form and name it `ExportDataLocationDialog`. Change the `FileName` property to `POData.per` so it defaults to an appropriate name for the `Person` table.

7. Add an event handler routine to the Tools ⇨ Export Data menu item by double-clicking it and add the following code:

```
Private Sub exportToolStripMenuItem_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles exportToolStripMenuItem.Click
    With ExportDataLocationDialog
        If .ShowDialog = Windows.Forms.DialogResult.OK Then
            If ExportPOData(mCurrentUserID, .FileName) = False Then
                MessageBox.Show("Export Failed!")
            End If
        End If
    End With
End Sub
```

This will show the File Save dialog, and if the user correctly selects a filename and clicks Save, will call the `ExportPOData` function you just created. Go ahead and run the application. Select Tools ⇨ Export Data and choose a location for the files to be stored. After it has been completed, locate the files that were created and take a look at the contents. Figure 12-1 shows some sample output. Note how the schema defining what fields belong to a record is defined at the beginning of the file and is then followed by `POUser` nodes for each `POUser` row in the table.

8. Creating an import function is actually significantly more difficult because there are two database tables with a relationship defined between them. The `Person` table stores the unique ID for the `POUser` with which it is associated. However, when importing the data for the tables, it's necessary to delete what's currently in the table and create an entire set of new rows. This results in the `POUser` rows all having new ID values. If the `Person` table is then imported, it fails because the `POUser` rows the XML is referencing no longer exist.
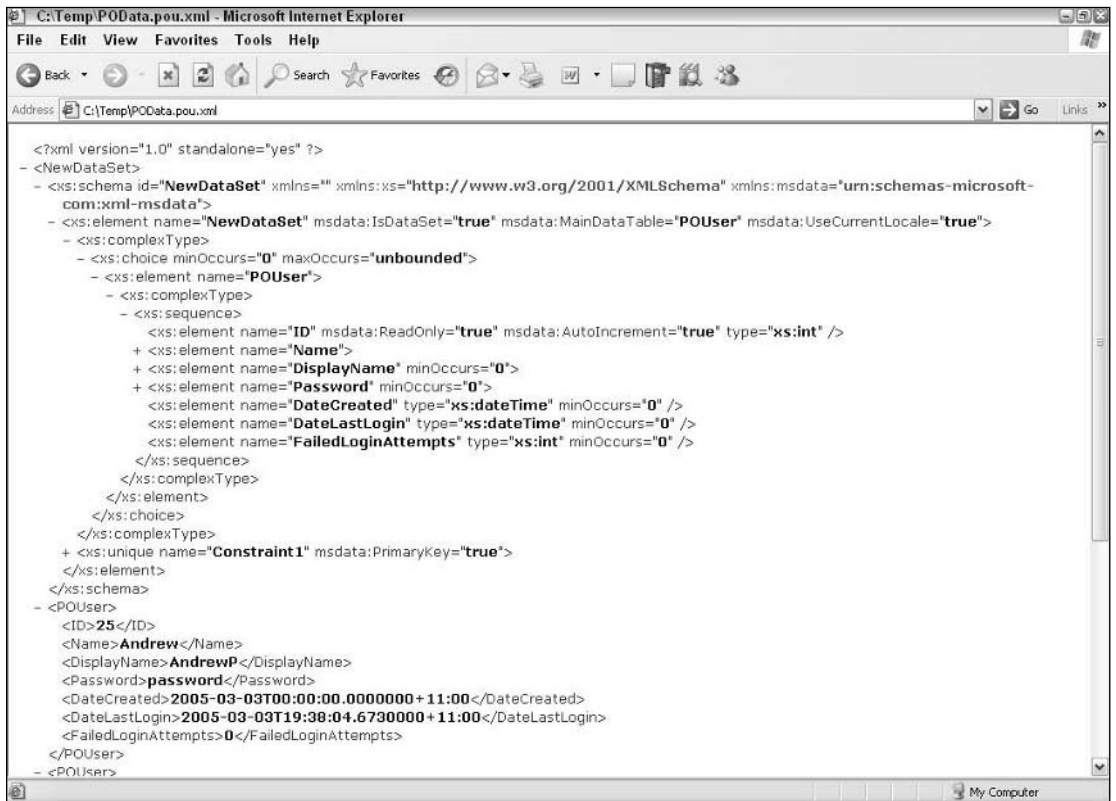
Figure 12-1

Instead, you need to first read the POUser XML import file and store the original POUser details in a collection. Then you can delete the contents of the current POUser table in the database and create new rows from the XML file. When this has updated the database, you then read through the new table and extract the new ID values and store them in the collection, too.

This enables you to create the Person rows — as you read each Person row, extract the old POUserID value and find it in the collection you built. Then you can access the new POUser row by the corresponding new POUserID value in the collection.

One last thing you'll need to do is reassign the CurrentUserID, because deleting and recreating the tables causes a new ID to be assigned to the currently logged on user.

9. Create the ImportPOData function, but this time define the return value as an Integer:

```
Public Function ImportPOData(ByVal UserID As Integer, _
    ByVal ImportDataLocation As String) As Integer

End Function
```

249

10. You need to retrieve the `Name` property of the currently logged on user so you can find that person again after the data has been recreated. To do this, create a new function that reverses the order of the `GetUserID` function you created and used in Chapter 8:

```
Public Function GetUserName(ByVal ID As Integer) As String
    Dim CheckUserAdapter As New _PO_DataDataSetTableAdapters.POUserTableAdapter
    Dim CheckUserTable As New _PO_DataDataSet.POUserDataTable

    CheckUserAdapter.Fill(CheckUserTable)
    Dim CheckUserDataView As DataView = CheckUserTable.DefaultView
    CheckUserDataView.RowFilter = "ID = " + ID.ToString

    With CheckUserDataView
        If .Count > 0 Then
            Return .Item(0).Item("Name").ToString
        Else
            Return vbNullString
        End If
    End With
End Function
```

11. Return to the `ImportPOData` function and store the name by calling the new function:

```
Public Function ImportPOData(ByVal UserID As Integer, _
    ByVal ImportDataLocation As String) As Integer
    Dim CurrentUserName As String = GetUserName(UserID)

End Function
```

12. Just as with the `ExportPOData` function, you need to create the `POUser` XML filename. At this point, you should make sure both files exist; if they don't, then return –1 to indicate there was a problem in the function:

```
Public Function ImportPOData(ByVal UserID As Integer, _
    ByVal ImportDataLocation As String) As Integer
    Dim CurrentUserName As String = GetUserName(UserID)
    Dim POUserLocation As String
    POUserLocation = ImportDataLocation.Remove(ImportDataLocation.Length - 3, _
        3) & "pou"
    With My.Computer.FileSystem
        If .FileExists(ImportDataLocation) = False Then Return -1
        If .FileExists(POUserLocation) = False Then Return -1
    End With
End Function
```

13. As outlined in step 8, you need to first build a collection that stores the original ID values for each `POUser` row. Create a small private class at the bottom of the `GeneralFunctions.vb` module to use in the collection:

```
Private Class ImportDataUserInfo
    Public OriginalID As Integer
    Public NewID As Integer
    Public Name As String
End Class
```

14. Reading the XML file is actually quite easy: Create a new `DataTable` object and use the `ReadXml` method to bring the data into the table. Because the XML you exported contains the schema, the `ReadXml` method understands how to translate the data to the database table:

```
Dim UserTable As New _PO_DataDataSet.POUserDataTable
UserTable.ReadXml(POUserLocation)
```

15. Iterate through each `POUserRow` and store the `ID` and `Name` column values in a `Collection`:

```
Dim UserCollection As New Collection
For Each MyRow As _PO_DataDataSet.POUserRow In UserTable.Select()
    Dim CurrentUserInfo As New ImportDataUserInfo
    CurrentUserInfo.OriginalID = MyRow.ID
    CurrentUserInfo.Name = MyRow.Name
    UserCollection.Add(CurrentUserInfo)
Next
```

16. Now you can delete the data from the two tables. First fill the `DataTable` from the database and then delete each row one by one. When they're gone, call the `Update` method of the `DataAdapter` object to update the database:

```
Dim PersonAdapter As New _PO_DataDataSetTableAdapters.PersonTableAdapter
Dim PersonTable As New _PO_DataDataSet.PersonDataTable
PersonAdapter.Fill(PersonTable)
For Each MyRow As _PO_DataDataSet.PersonRow In PersonTable.Select()
    MyRow.Delete()
Next
PersonAdapter.Update(PersonTable)

Dim UserAdapter As New _PO_DataDataSetTableAdapters.POUserTableAdapter
UserAdapter.Fill(UserTable)
For Each MyRow As _PO_DataDataSet.POUserRow In UserTable.Select()
    MyRow.Delete()
Next
UserAdapter.Update(UserTable)
```

17. When the two tables have been cleared out, the `POUser` table can be created directly from the XML file:

```
 UserTable.ReadXml(POUserLocation)
 UserAdapter.Update(UserTable)
```

18. The `UserCollection` array now needs to be updated with the new ID values that were created by the previous two statements. Iterate through all the rows of the table and find each one in the `UserCollection` array. When found, update the `NewId` property:

```
For Each MyRow As _PO_DataDataSet.POUserRow In UserTable.Select()
    For Each CurrentUserInfo As ImportDataUserInfo In UserCollection
        If CurrentUserInfo.Name = MyRow.Name Then
            CurrentUserInfo.NewID = MyRow.ID
            Exit For
        End If
    Next
Next
```

251

19. Importing the `Person` table is done differently. Like the `AddPerson` function, you need to include the `POUser` row to which the `Person` row belongs. First read the XML file into a separate table so you can process the information before adding it to the database:

```
Dim ImportPersonTable As New _PO_DataDataSet.PersonDataTable
ImportPersonTable.ReadXml(ImportDataLocation)
```

20. Iterate through the rows of this table, and for each one, look through the `UserCollection` array for a matching `OriginalID` value. Once this is found, store the `NewID` value in a temporary variable and exit the loop:

```
For Each MyRow As _PO_DataDataSet.PersonRow In ImportPersonTable.Select()
    With MyRow
        Dim NewPOUserID As Integer
        For Each CurrentUserInfo As ImportDataUserInfo In UserCollection
            If .POUserID = CurrentUserInfo.OriginalID Then
                NewPOUserID = CurrentUserInfo.NewID
                Exit For
            End If
        Next
        ... add the row here.
    End With
Next
```

21. With the new ID, you can retrieve the correct row from the `POUser` table by using the `Select` method. Use this `POUser` row as a parameter in the `AddPersonRow` method of the `PersonTable`, along with the fields in the imported `Row` object. Once you've finished processing all the rows that have been read from the XML file, call the `Update` method of the `Adapter` to send the changes to the database:

```
For Each MyRow As _PO_DataDataSet.PersonRow In ImportPersonTable.Select()
    With MyRow
        Dim NewPOUserID As Integer
        For Each CurrentUserInfo As ImportDataUserInfo In UserCollection
            If .POUserID = CurrentUserInfo.OriginalID Then
                NewPOUserID = CurrentUserInfo.NewID
                Exit For
            End If
        Next
        Dim POUserRows() As _PO_DataDataSet.POUserRow = CType(UserTable.Select( _
            "ID = " + NewPOUserID.ToString), _PO_DataDataSet.POUserRow())

        PersonTable.AddPersonRow(POUserRows(0), .NameFirst, .NameLast, _
            .PhoneHome, .PhoneCell, .Address, .EmailAddress, .DateOfBirth, _
            .Favorites, .GiftCategories, .Notes)
    End With
Next
PersonAdapter.Update(PersonTable)
```

22. The final step is to find the new ID for the currently logged on user. You can do this by iterating through the `UserCollection` array looking for the `CurrentUserName` you saved at the beginning of the function. When you find it, simply return the `NewID` value:

```
    For Each CurrentUserInfo As ImportDataUserInfo In UserCollection
        If CurrentUserInfo.Name = CurrentUserName Then
            Return CurrentUserInfo.NewID
        End If
    Next
    Return -1
```

23. Return to the `MainForm` Design view, add an `OpenFileDialog`, and name it `ImportDataLocationDialog`. Add the following code to the Tools ➪ Import Data menu item's `Click` event handler:

```
Private Sub importToolStripMenuItem_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles importToolStripMenuItem.Click
    With ImportDataLocationDialog
        If .ShowDialog = Windows.Forms.DialogResult.OK Then
            Dim TempUserID As Integer = ImportPOData(mCurrentUserID, .FileName)
            If TempUserID = -1 Then
                MessageBox.Show("Could not find the current user in the new " & _
                    "data. Program ending!")
                End
            Else
                mCurrentUserID = TempUserID
            End If
        End If
    End With
    End Sub
```

24. This will show the Open File dialog window, enabling users to select the `Person` data file to be imported. Once they click Open, the `ImportPOData` function is called; if successful, it will return the new ID value for the currently logged on user, which then updates the module-level variable for future functions. Run the application and change some data in your `Person` tables, and then import the data you exported in step 7.

The System.Xml Namespace

Now that you have a handle on how XML can be used in your application with only a few simple function calls, it's time to take a look at how extensive the XML support is in .NET. Most XML classes can be found in the `System.Xml` namespace. By default, your Visual Basic Express projects do not have access to this set of classes, so you need to add a reference to it first.

The core object you will most likely use in your applications is the `XmlDocument` class. This class represents an entire XML file. As discussed earlier in this chapter, each XML file has a single root element that contains the entire information set — the `XmlDocument` object represents that root element.

To create a new `XmlDocument`, you use the following command:

```
Dim myXmlDocument As New System.Xml.XmlDocument()
```

Once you have an `XmlDocument` object, you can begin to process the individual elements within the XML. If you want to read XML from a location, you need to load it into the `XmlDocument` object using either `Load` or `LoadXml`. The `Load` method takes three different types of input streams: an `IOStream`, a `TextReader`, or a filename. The `LoadXml` method accepts a string variable that it expects to contain XML:

```
Dim myXml As String = "<config> " & _
                   "   <Values>" & _
                   "      <Setting>Value</Setting>" & _
                   "      <Setting>123</Setting>" & _
                   "   </Values>" & _
                   "   <State>" & _
                   "      <User Login='true'>Andrew</User>" & _
                   "   </State>" & _
                   "</config>"
myXmlDocument.LoadXml(myXml)
```

Once you have the XML in the `XmlDocument` object, you can retrieve a string representation at any time using the `ToString` method. This can then be used to write back to a file using any of the methods you prefer. Alternatively, you can use the `WriteTo` and `WriteContentTo` functions to write the contents of the `XmlDocument` elements to an `XmlWriter` object.

Each element within the XML file is represented by an `XmlNode` object. The main `XmlDocument` object has a property called `ChildNodes` that returns the root node. This node has its own `ChildNodes` collection that returns the child elements belonging to it, and so on down the hierarchy. The simplest way to get to the `User` node in the sample would be the following line of code:

```
Dim myUserNode as XmlNode = myXmlDocument.ChildNodes(0).ChildNodes(1).ChildNodes(0)
```

The first `ChildNodes` object returns the `config` node, the second returns the `State` node, and the third returns the `User` node. Once you have the element you need, you can access its attributes through an `Attributes` collection, and the value stored between the opening and closing tags via the `InnerText` property.

Attributes can be retrieved by their name if you know them or accessed via their index in the collection. The following line of code displays the name of the node, the text within the opening and closing tags, and the `Login` attribute:

```
MessageBox.Show("Node = " & myUserNode.Name & ", Value = " & _
    myUserNode.InnerText & ", Login Attribute = " & _
    myUserNode.Attributes("Login").ToString)
```

If you need a specific child element of a node you're working with, you can use the `SelectSingleNode` method. If more than one node matches the criteria, Visual Basic Express throws an exception that you must trap. Otherwise, the `SelectSingleNode` method returns either `Nothing` (indicating the node wasn't present) or an `XmlNode` object with the child node:

```
Dim myValuesNode As XmlNode = myXmlDocument. SelectSingleNode("config/Values")
```

Alternatively, if you are trying to retrieve a collection of nodes that are all of the same type, you can use the `SelectNodes` function. Rather than return an `XmlNode` object, this function returns an `XmlNodeList` collection that contains all of the nodes that met the criteria. To retrieve the `Setting` nodes from the `Values` element and display the value for each, you could use this code:

```
Dim mySettings As XmlNodeList = myValuesNode.SelectNodes("Setting")
For Each mySettingNode As XmlNode In mySettings
    MessageBox.Show(mySettingNode.InnerText)
Next
```

Inserting `XmlNodes` into an existing `XmlDocument` can be done through the `CreateElement` method exposed by the `XmlDocument` object and the `AppendChild` method of the `XmlNode` class. First you need to create the new `XmlNode` object using `CreateElement`:

```
Dim myNewSetting As XmlNode = myXmlDocument.CreateElement("Setting")
```

Once you have the node, you can set its attributes through the `Attributes` collection, and the value with the `InnerText` property. Then you add it to the node that should be its parent:

```
myNewSetting.InnerText = "NewData"
myValuesNode.AppendChild(myNewSetting)
```

You can also use the `InsertBefore` and `InsertAfter` methods to insert the new node into the `ChildNodes` collection in a specific location.

Alternatively, creating `XmlNodes` within a document can be done using an `XmlWriter`. If the node is at the bottom of the hierarchy and does not contain any other elements, use the `WriteElementString` function. If the element contains other nodes, you need to use the `WriteStartElement` and `WriteEndElement` methods to create the opening and closing tags. The following code snippet writes out the first half of the sample XML `config` file:

```
Dim MyNavigator As XPath.XPathNavigator = myXmlDocument.CreateNavigator()
Dim MyWriter As XmlWriter = MyNavigator.PrependChild()
MyWriter.WriteStartElement("config")
MyWriter.WriteStartElement("Values")
MyWriter.WriteElementString("Setting","Value")
MyWriter.WriteElementString("Setting","123")
MyWriter.WriteEndElement()
MyWriter.WriteEndElement()
```

> **Speaking of code snippets, Visual Basic Express comes with a number of useful snippets relating to XML. From reading an XML file using an** `XmlReader` **to inserting** `XmlNode` **objects into an existing** `XmlDocument` **to finding an individual node, the code snippet library is an excellent resource for those situations when you just can't think of what you need. It even has an excellent serialization example to automatically convert a class into XML form and write it out to a file.**

The next Try It Out ties together a lot of the concepts you've learned up to this point to create a wizard form that you can add to any application that needs its own custom-built step-by-step wizard. The wizard takes an XML configuration file and builds the pages dynamically, including images, controls, and text. When the user clicks the Finish button, it then compiles the values chosen into an XML document and returns it to the calling program. The types of functionality found in this Try It Out include the following:

❑ Adding controls to a form, docking them into place, using auto alignment, and setting properties of the form itself

❑ Defining regions within your code to organize it into logical areas that are easy to manage

❑ Using Imports to shortcut variable definitions

❑ Using XML to read and create documents and to search for individual nodes

❑ Dynamically altering the properties of controls at runtime, including the form

❑ Creating internal structures (`Class` and `Enum`) to support the rest of the code

❑ Creating controls dynamically, adding them to the form, and then deleting them when they're done

Try It Out   Creating a Wizard Form

1. Start Visual Basic Express and create a new Windows Application project. This project will be used as a testing ground for your wizard form, as well as where you design the wizard itself. Call the application `WizardControl`.

2. Most wizards follow the same pattern — a series of pages, or steps, that users navigate through until they arrive at the last one and click the Finish button. Normally, you have several buttons at the bottom of the form for navigation, a picture on the left-hand side, and information describing the current page.

Rather than hardcode each of the pages for a specific wizard, your form is going to dynamically build the page for each step as needed, creating the controls and placing them on the form as well as setting all the text and visual clues. The information regarding what goes where will be controlled through an XML file.

How It Works — The User Interface

Add a new form to the project, naming it `WizardBase.vb`, and set the following properties:

❑ **Name** — `WizardBase`

❑ **FormBorderStyle** — `FixedDialog`

❑ **Size** — `426,300`

Setting the form to a fixed size enables you to control how each wizard that uses the form appears.

3. Add to the form three `Panel` objects that you'll use to control the layout of the form. The first `Panel` will contain the navigation buttons. `Dock` it to the bottom of the form and set the `Height` to `30` pixels to provide just enough room for the buttons.

The second `Panel` should have its `Dock` property set to `Left` and its name changed to `pnlGraphic`. This area will be used to store the image associated with the wizard's steps. To provide a logical size for the graphic images, set its `Width` property to `120`. In addition, set

the `BackgroundImageLayout` property to `Stretch` so that any images loaded stretch to the available area. The last panel should have its `Dock` property set to `Fill` to take up the remaining space in the form.

4. Add five buttons to the bottom panel and evenly space them out. Use the built-in visual alignment cues that Visual Basic Express provides so the buttons all line up and are at the optimum distance from the edges of the form.

Set the Text property of the buttons to `Cancel`, `Start`, `< Previous`, `Next >`, and `Finish`. Change the names of the button controls to correspond to these captions.

5. Believe it or not, you're almost done creating the user interface. The only thing left to do is add three elements to the main area to contain the current step information. Add a `Label`, a `TextBox`, and another `Panel` control to the panel taking up the main area of the form.

6. The `Label` will be used to display the heading of the current step. Change its `Font` properties so it's a lot larger and bolder than normal text. Set its `Name` property to `lblHeading` so you can change it in code later.

7. The `TextBox` will contain the detailed description of what the user should do in the current step. Because this could be lengthy, a `TextBox` is used to display a few lines at a time. It should also be blended in the form so it doesn't draw away attention from the actual settings that the user is supposed to be changing. Set the following properties:

❑ **BorderStyle** — `None`
❑ **ScrollBars** — `Vertical`
❑ **Multiline** — `True`
❑ **ReadOnly** — `True`
❑ **Name** — `txtDescription`

8. The `Panel` control should be resized so it takes up the remaining space in the form and named `pnlControls` so that the program knows where to add the controls at runtime. Because you don't know if the space will be enough for any given page in a wizard, set the `AutoScroll` property to `True`. If the wizard dynamically adds more controls than can fit in the visible area, scrollbars will automatically be added to the panel so the user can get to them all. When you're done, the user interface should look like the one shown in Figure 12-2. Save the project so you don't lose the changes to your user interface.
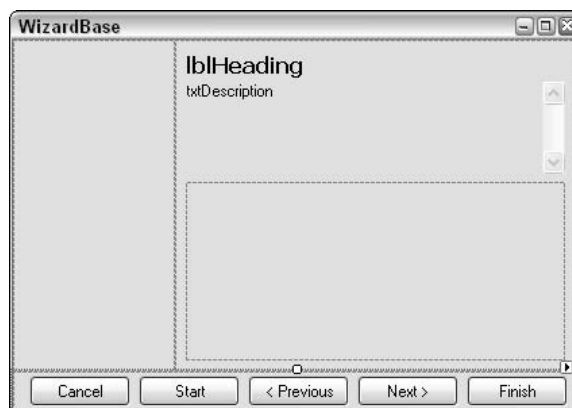
Figure 12-2

How It Works — The Data Definition

9. Before you can write the code, you need to understand how the data is presented to the form. Whenever an application needs a wizard, it will pass over a string containing XML-formatted information. The `WizardBase` form can process this XML to determine what the wizard is called, how many pages it has, and what information should be stored on a page.

Breaking the information down, a wizard typically needs the following information:

❑ **Name** — To identify the wizard internally

❑ **Title** — Displayed at the top of the form to inform users about the wizard's purpose

❑ **Graphic** — An image that can be displayed in the left-hand pane of the wizard

❑ **Finish flag** — A Boolean value that indicates whether users must navigate through all the pages before the Finish button is enabled or whether they can click Finish at any time

Within the wizard are a number of pages, or steps. Each step needs its own information:

❑ **Name** — To identify the step internally

❑ **Heading** — The text to be displayed in `lblHeading`

❑ **Description** — The information text to be displayed in `txtDescription`

❑ **Graphic** — An optional image that can be used to override the main wizard graphic for individual steps

A step has components with which users interact. As some steps might be informational only, the collection of components might not exist for a particular step, but each component that is defined needs a certain amount of information:

❑ **Name** — To identify the component internally

❑ **Caption** — Displayed next to the control so users knows the particular component's purpose

❑ **Value** — The value for the component

❑ **Control Type** — An identifier telling `WizardBase` what kind of control should be employed for this component

Rather than allow any kind of component in the wizard and potentially have a nightmare on your hands trying to manage the myriad of options in the code, you can restrict it to only a few. Generally, wizards need one of only four different types of component:

❑ A **CheckBox** to indicate a Boolean value — use a value of `CB`

❑ A **TextBox** to allow text settings — use a value of `TB`

❑ A collection of **RadioButtons** to select from a small number of options — use a value of `RB`

❑ A **ComboBox** to enable users to select from multiple options without taking up space on the form — use a value of `CM`

Except for the `ComboBox` control, all of the preceding elements can be controlled by the previously mentioned settings. That control needs a list of allowable values that is used to populate its list. The allowable values need only the display value and an indicator of which one is to be selected by default.

How It Works — Translating to XML

10. Using this information, you can create a sample XML file that defines the various values and attributes for each component, as shown here:

```
<Wizard Name="W" Title="T" GlobalGraphic="FN" AllowFinishBeforeLastStep="False">
    <Step Name="Intro">
        <Heading>Introduction</Heading>
        <Description>Description goes here</Description>
        <Graphic>Filename</Graphic>
        <Component Name="Name1" ControlType="CB" Caption="MyCap1">Value</Component>
        <Component Name="Name2" ControlType="CM" Caption="MyCap2">
          <AllowedValue Name="Value1" Selected="True">Value1</AllowedValue>
          <AllowedValue Name="Value2">Value2</AllowedValue>
        </Component>
    </Step>
</Wizard>
```

How It Works — Defining Supporting Structures

11. Now that you know the contents of the XML that specifies how the wizard is to be displayed, return to your project and open the `WizardBase` form in code view. Before you begin creating the logic, it makes sense to build some supporting structures to make dealing with individual steps and components more logical. Create a `Region` in the code called `Supporting Structures` to contain the classes and types you will write:

```
#Region "Supporting Structures"

#End Region
```

12. The first thing to do is create an `Enum` that contains only the allowed control types for the components:

```
#Region "Supporting Structures"
    Private Enum AllowedControlTypes As Integer
        CheckBox = 1
        ComboBox = 2
        RadioButton = 3
        TextArea = 4
    End Enum
#End Region
```

If you want to support other object types, you will need to add them to this `Enum`.

13. To store the information about a particular step, create a private `WizardStep` class within the `WizardBase` code. Making it private hides it from public use and enables you to do things that you would normally not do. Because you are in control of when this class is used, rather than define complete `Property Get` and `Set` statements for each attribute of a step, you can just define public variables:

```
    Private Class WizardStep
        Public Number As Integer
        Public Name As String
        Public Heading As String
        Public Description As String
```

```
            Public Graphic As Image
            Public Components() As WizardComponent
        End Class
```

You might note that these variables all equate to the different components of a step that was identified earlier. The `Components` object is defined as an array of `WizardComponent` classes, which you create next.

14. Create another private class for each component of a step. The `ControlType` can be defined with a type of `AllowedControlTypes`, the `Enum` you created in step 12. The `AllowedValues` array stores the information for `ComboBox` controls and is set to `Nothing` for the other control types:

```
    Private Class WizardComponent
        Public ComponentControlType As AllowedControlTypes
        Public ComponentName As String
        Public ComponentCaption As String
        Public ComponentValue As String
        Public ComponentAllowedValues() As String
    End Class
```

15. While you could create yet another class for the wizard itself, only a few properties are required, and because the `WizardBase` form handles only one wizard at a time, you can just store these as module-level variables:

```
#Region "Properties"
    Private mFinishBeforeLastStepAllowed As Boolean = True
    Private mWizardFormTitle As String
    Private mGlobalGraphic As Boolean
    Private mGlobalGraphicFileName As String
    Private mGlobalGraphicImage As Image
#End Region
```

Notice that the `GlobalGraphic` property has three objects associated with it—a string to store the file location of the image to use, an `Image` object to store the actual image, and a `Boolean` flag to indicate whether a global graphic image is defined in the wizard.

16. You need to expose two properties: a `Definition` string that the application can use to pass over the wizard definition in XML, and a `SettingValues` string that is used by the `WizardBase` to return the values the user has chosen. The `SettingValues` property can be read-only:

```
#Region "Properties"
    Private mFinishBeforeLastStepAllowed As Boolean = True
    Private mWizardFormTitle As String
    Private mGlobalGraphic As Boolean
    Private mGlobalGraphicFileName As String
    Private mGlobalGraphicImage As Image

    Private mWizardDefinition As String
    Private mWizardSettings As String
    Public Property WizardDefinition() As String
        Get
            Return mWizardDefinition
        End Get
        Set(ByVal value As String)
```

```
                mWizardDefinition = value
           End Set
      End Property
      Public ReadOnly Property WizardSettingValues() As String
           Get
                Return mWizardSettings
           End Get
      End Property
  #End Region
```

17. You need a few more properties and module-level variables. A read-only `Cancelled` property helps the application determine whether the user canceled the wizard instead of finishing it properly. In addition, because the controls are to be added dynamically in each step, keeping track of a standard control height is handy. Do it once when the form is loaded and then keep track of the value. This could easily be a constant, but to cater to different user systems that have a variety of control settings in their system setup, you should calculate the height.

Finally, several variables to keep track of the steps in the wizard will be needed. You need to know how many steps there are and what step is currently being displayed, and you need to define an array of `WizardStep` objects to store all the step information for the wizard. Add all of this to the `Properties` region:

```
      Private mControlHeight As Integer
      Private mNumberOfSteps As Integer
      Private mCurrentStep As Integer
      Private mSteps() As WizardStep
      Private mCancelled As Boolean = False
      Public ReadOnly Property Cancelled() As Boolean
           Get
                Return mCancelled
           End Get
      End Property
```

How It Works — Object Initialization

18. Now that the stage is set, you can start writing the code that drives the wizard. The first thing to do is write any setup or initialization code that is required when the form first loads. Create an event handler routine for the form's `Load` event and add the following code:

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

    Dim tempTB As New TextBox
    mControlHeight = tempTB.Height + 5

    ImportDefinition()

    mCurrentStep = 1
    Me.Text = mWizardFormTitle + " - Step " + mCurrentStep.ToString + " of " + _
        mNumberOfSteps.ToString
    SetForm(mCurrentStep)

End Sub
```

The first two lines create a `TextBox` control to determine the default height. The height of the control (plus a buffer so the dynamically created controls aren't right up against each other) is stored in the module-level variable you created earlier.

The next line calls the `ImportDefinition` subroutine that you'll define next. This routine extracts all the information `WizardBase` needs from the XML that was passed over to it.

The `CurrentStep` variable is set to the first step, and the text of the form itself is set to the wizard title, followed by the progress the user has made through the wizard. You could also encapsulate all of this programming logic into a separate subroutine called `InitializeWizardSettings`. This would enable the code to be called from multiple locations — not just when the form loads.

19. The `ImportDefinition` is where the XML data is first processed. To take advantage of the XML namespace available within Visual Basic Express, you need to first convert the string containing the XML to an actual XML document object:

```
Private Sub ImportDefinition()
    Dim xmlWizard As New XmlDocument()
    xmlWizard.LoadXml(mWizardDefinition)
End Sub
```

If you get errors while defining the `XmlDocument`, you need to first add a reference to `System.Xml` and then use the `Imports` statement at the top of the module to import that namespace. As discussed in Chapter 11, this enables you to create objects without needing to fully define their name (the alternative would be to define `xmlWizard` as a `System.Xml.XmlDocument` object).

20. You can use the `SelectSingleNode` method of the `XmlDocument` class to extract the `Wizard` node and its children (as discussed earlier in this chapter). This is useful if the XML string passed to the `WizardBase` form contains other information that's not relevant:

```
Private Sub ImportDefinition()
    Dim xmlWizard As New XmlDocument()
    xmlWizard.LoadXml(mWizardDefinition)

    Dim WizardXML As Xml.XmlNode
    WizardXML = xmlWizard.SelectSingleNode("Wizard")

End Sub
```

21. All of the information about the wizard can be found in the `Attributes` collection, so write the following loop to iterate through the list and extract the information you want for each of the `Wizard` variables:

```
Private Sub ImportDefinition()
    Dim xmlWizard As New XmlDocument()
    xmlWizard.LoadXml(mWizardDefinition)

    Dim WizardXML As Xml.XmlNode
    WizardXML = xmlWizard.SelectSingleNode("Wizard")

    For Each WizardAttribute As XmlAttribute In WizardXML.Attributes
        Select Case WizardAttribute.Name
            Case "Title"
                mWizardFormTitle = WizardAttribute.Value
            Case "GlobalGraphic"
```

```
                    mGlobalGraphicFileName = WizardAttribute.Value
                    mGlobalGraphicImage = Image.FromFile(mGlobalGraphicFileName)
                    pnlGraphic.BackgroundImage = mGlobalGraphicImage
                    mGlobalGraphic = True
                Case "AllowFinishBeforeLastStep"
                    If WizardAttribute.Value.ToLower = "true" Then
                        mFinishBeforeLastStepAllowed = True
                    Else
                        mFinishBeforeLastStepAllowed = False
                    End If
            End Select
        Next
```

```
    End Sub
```

22. Notice that the `GlobalGraphic` attribute is used to set all three module-level variables — if the `GlobalGraphic` attribute is never found, then the `mGlobalGraphic` Boolean variable defaults to `False`. To finish this routine, you need to create the `Steps` array. You'll write a new function in a moment that extracts `Step` information, so call that at the end of the `ImportDefinition` routine and assign the returned object to the module-level array of `WizardSteps`:

```
        mSteps = GetSteps(WizardXML)
```

23. As mentioned in the last step, you now need to create a function that extracts the information about the steps in a wizard from the XML. First define the function and accept an `XmlNode` object as a parameter. Make the return value an array of `WizardStep` objects:

```
    Private Function GetSteps(ByVal WizardXml As Xml.XmlNode) As WizardStep()

    End Function
```

24. Establish just how many steps there are for this wizard definition. To do that, you can use the `SelectNodes` method of the `XmlNode` class. This works just like the `SelectNodes` method for the `XmlDocument` class and returns a special collection object called an `XmlNodeList`, containing all nodes that met the particular search criteria. Because the function accepts the `Wizard` node as a parameter, the criteria to pass to the `SelectNodes` function is simply the name of the child node — `Step` — like so:

```
Private Function GetSteps(ByVal WizardXml As Xml.XmlNode) As WizardStep()

    Dim StepsList As Xml.XmlNodeList
    StepsList = WizardXml.SelectNodes("Step")

End Function
```

25. Once you have this collection of nodes, you can determine the number of steps and create an array of `WizardStep` objects to populate. This array is then returned after you process each `Step` node:

```
Private Function GetSteps(ByVal WizardXml As Xml.XmlNode) As WizardStep()

    Dim StepsList As Xml.XmlNodeList
    StepsList = WizardXml.SelectNodes("Step")
```

```
        mNumberOfSteps = StepsList.Count
        Dim StepArray(mNumberOfSteps) As WizardStep

        ... processing the nodes will go here

        Return StepArray
    End Function
```

26. You can use the `For Each` loop to process each `XmlNode` object in the `StepsList` collection you just created. As you process each new node, increment a local variable by 1 to keep track of the current step you are processing, define the array element as a new `WizardStep` object, and set the `Number` property to the local variable:

```
Private Function GetSteps(ByVal WizardXml As Xml.XmlNode) As WizardStep()

        Dim StepsList As Xml.XmlNodeList
        StepsList = WizardXml.SelectNodes("Step")
        mNumberOfSteps = StepsList.Count

        Dim StepArray(mNumberOfSteps) As WizardStep

        Dim CurrentStep As Integer = 0
        For Each StepXml As Xml.XmlNode In StepsList
            CurrentStep += 1
            StepArray(CurrentStep) = New WizardStep
            StepArray(CurrentStep).Number = CurrentStep
        Next
        Return StepArray
    End Function
```

27. The information for the `WizardStep` class is in two parts. The first is the name of the step and is found as an `Attribute` of the node. Because you're interested in only one attribute and you know its name, you can refer to it directly in the `Attributes` collection like so:

```
StepArray(CurrentStep).Name = StepXml.Attributes("Name").Value
```

28. The `Heading` and `Description` properties are found in individual children nodes of the `Step`. Again, you can use the `SelectSingleNode` method to retrieve them directly. Even better, because you're interested only in the content of the node, you don't even need to create an `XmlNode` object — extract the information using the `InnerText` property:

```
StepArray(CurrentStep).Heading = StepXml.SelectSingleNode("Heading").InnerText
StepArray(CurrentStep).Description = _
    StepXml.SelectSingleNode("Description").InnerText
```

29. The `Graphic` property of a step is optional. You first need to try to find it, and only if it's found can you then load the image:

```
Dim GraphicNode As XmlNode = StepXml.SelectSingleNode("Graphic")
If GraphicNode IsNot Nothing Then
    StepArray(CurrentStep).Graphic = Image.FromFile(GraphicNode.InnerText)
End If
```

30. The final property of the `WizardStep` object is the `Components` array. Much like the `GetSteps` function, you'll create a separate function called `GetComponents` that returns an array of `WizardComponent` objects, so assign the return value of that function to the `Components` property. The final `GetSteps` function should look like this:

```
Private Function GetSteps(ByVal WizardXml As Xml.XmlNode) As WizardStep()

    Dim StepsList As Xml.XmlNodeList
    StepsList = WizardXml.SelectNodes("Step")
    mNumberOfSteps = StepsList.Count

    Dim StepArray(mNumberOfSteps) As WizardStep

    Dim CurrentStep As Integer = 0
    For Each StepXml As Xml.XmlNode In StepsList
        CurrentStep += 1
        StepArray(CurrentStep) = New WizardStep
        StepArray(CurrentStep).Number = CurrentStep
        StepArray(CurrentStep).Name = StepXml.Attributes("Name").Value
        StepArray(CurrentStep).Heading = _
            StepXml.SelectSingleNode("Heading").InnerText
        StepArray(CurrentStep).Description = _
            StepXml.SelectSingleNode("Description").InnerText
        Dim GraphicNode As XmlNode = StepXml.SelectSingleNode("Graphic")
        If GraphicNode IsNot Nothing Then
            StepArray(CurrentStep).Graphic = Image.FromFile(GraphicNode.InnerText)
        End If
        StepArray(CurrentStep).Components = GetComponents(StepXml)
    Next
    Return StepArray
End Function
```

31. The last routine that processes the XML is the `GetComponents` function. This accepts an `XmlNode` object as a parameter and returns an array of `WizardComponent` objects. You extract the `Component` nodes in the same way you did the `Step` nodes in the `GetSteps` function; using the `SelectNodes` method. Because a step can have no `Components`, you first need to check whether the `SelectNodes` method returned a list of nodes. If not, then simply return `Nothing`.

32. If there is a list of nodes, then declare an array of `WizardComponent` objects and return that array after processing the list:

```
Private Function GetComponents(ByVal StepXml As Xml.XmlNode) As WizardComponent()
  Dim ComponentsList As Xml.XmlNodeList
  ComponentsList = StepXml.SelectNodes("Component")
  If ComponentsList Is Nothing Then
    Return Nothing
  Else
    Dim CurrentComponents(ComponentsList.Count) As WizardComponent

    ... process the Component nodes here

    Return CurrentComponents
  End If
End Function
```

33. Define a local variable to keep track of which component you are processing and use `For Each` to loop through the `ComponentsList` collection. At the beginning of each iteration of the loop, increment the local variable and create a new `WizardComponent` object:

```
Dim CurrentComponentCounter As Integer = 0
For Each ComponentXml As Xml.XmlNode In ComponentsList
  CurrentComponentCounter += 1
  CurrentComponents(CurrentComponentCounter) = New WizardComponent
Next
```

34. Set three main properties in the `WizardComponent` class: `ControlType`, `Name`, and `Caption`. All four component types use these attributes, so iterate through the `Attributes` collection of each `Component` node to extract this information. The `ControlType` attribute needs to be translated to the internal `Enum` that the `ComponentControlType` property uses:

```
Dim CurrentComponentCounter As Integer = 0
For Each ComponentXml As Xml.XmlNode In ComponentsList
  CurrentComponentCounter += 1
  CurrentComponents(CurrentComponentCounter) = New WizardComponent
    With CurrentComponents(CurrentComponentCounter)
      For Each ComponentAttribute As XmlAttribute In ComponentXml.Attributes
        Select Case ComponentAttribute.Name
          Case "ControlType"
            Select Case ComponentAttribute.Value
              Case "RB"
                .ComponentControlType = AllowedControlTypes.RadioButton
              Case "TB"
                .ComponentControlType = AllowedControlTypes.TextArea
              Case "CB"
                .ComponentControlType = AllowedControlTypes.CheckBox
              Case "CM"
                .ComponentControlType = AllowedControlTypes.ComboBox
            End Select
          Case "Name"
            .ComponentName = ComponentAttribute.Value
          Case "Caption"
            .ComponentCaption = ComponentAttribute.Value
        End Select
      Next
    End With
Next
```

35. The `ComboBox` components have additional information and use a different technique to determine the selected (or displayed) value. After extracting the information from the `Attributes` collection, you'll know what `ComponentControlType` the item is, so check whether it's a `ComboBox`. If it's not a `ComboBox`, you can simply set the `ComponentValue` property to the `InnerText` property of the `Component` node:

```
If .ComponentControlType = AllowedControlTypes.ComboBox Then
   ... process AllowedValues here
Else
  .ComponentValue = ComponentXml.InnerText
End If
```

Component nodes that are defined as a `ComboBox` have a collection of `AllowedValue` nodes. You can use the same `SelectNodes` method to grab the list of `AllowedValues` nodes to work on. If the `SelectNodes` method returns a collection, loop through each node in the list extracting the `InnerText` property for the value to be used in the `ComboBox` list:

```
If .ComponentControlType = AllowedControlTypes.ComboBox Then
  Dim AllowedValuesList As Xml.XmlNodeList
  AllowedValuesList = ComponentXml.SelectNodes("AllowedValue")
  If AllowedValuesList IsNot Nothing Then
    Dim sValues(AllowedValuesList.Count) As String
    Dim AllowedCounter As Integer = 0
    For Each AllowedValueXml As Xml.XmlNode In AllowedValuesList
      AllowedCounter += 1
      sValues(AllowedCounter) = AllowedValueXml.InnerText
    Next
    .ComponentAllowedValues = sValues
  End If
Else
  .ComponentValue = ComponentXml.InnerText
End If
```

You also need to determine which entry in the `AllowedValues` list is selected by default. Add the following lines of code to find the `Selected` attribute; if it's found, check for a value of `True`:

```
If .ComponentControlType = AllowedControlTypes.ComboBox Then
  Dim AllowedValuesList As Xml.XmlNodeList
  AllowedValuesList = ComponentXml.SelectNodes("AllowedValue")
  If AllowedValuesList IsNot Nothing Then
    Dim sValues(AllowedValuesList.Count) As String
    Dim AllowedCounter As Integer = 0
    For Each AllowedValueXml As Xml.XmlNode In AllowedValuesList
      AllowedCounter += 1
      sValues(AllowedCounter) = AllowedValueXml.InnerText
      Dim AllowAtt As XmlAttribute = AllowedValueXml.Attributes("Selected")
      If AllowAtt IsNot Nothing Then
        If AllowAtt.Value.ToLower = "true" Then
          .ComponentValue = AllowedValueXml.InnerText
        End If
      End If
    Next
    .ComponentAllowedValues = sValues
  End If
Else
  .ComponentValue = ComponentXml.InnerText
End If
```

When checking the `Selected` attribute, you'll have to compare a string representation of a Boolean value. In this case, you're looking for `True`, but because the XML file could contain any variation of capitalization (for example, *TRUE*, *True*, *true*, or even *TrUE*), you first have to convert it to some sort of common denominator. Fortunately, String variables have a built-in function called `ToLower` that converts all the text to lowercase; you can use that in this situation. For the record, they also have a `ToUpper` function that converts the string to all uppercase characters.

How It Works — Runtime Form Customization

36. The next task for this application is to create the routines that customize the form for each step. You saw the `SetForm` routine being called in the form's `Load` event handler in step 18. That subroutine enables and disables the navigation buttons depending on what step the user is up to in the wizard. It also sets the Heading and Description areas, the form's title bar text, and loads the image for the step if there is one. The final and most important part of `SetForm` is to dynamically create the components for the step so the user can interact with the wizard.

Define the `SetForm` subroutine so that it accepts a single parameter that indicates what step it should use. You could just interrogate the module-level variable that is keeping track of the current step, but doing it this way enables you to create a subroutine that can be called independently of that value:

```
Private Sub SetForm(ByVal CurrentStep As Integer)
End Sub
```

When the wizard is on step 1, it doesn't make sense to have the Start and Previous buttons enabled, so disable them. If the wizard has only one step, the Next button should also be disabled and the Finish button should be enabled because the first step is also the last step:

```
Private Sub SetForm(ByVal CurrentStep As Integer)
    If CurrentStep = 1 Then
        btnStart.Enabled = False
        btnPrevious.Enabled = False
        If mNumberOfSteps > 1 Then
            btnNext.Enabled = True
            btnFinish.Enabled = mFinishBeforeLastStepAllowed
        Else
            btnNext.Enabled = False
            btnFinish.Enabled = True
        End If
    End If
End Sub
```

If the current step is the last step, then disable the Next button and enable the Finish button; and if the wizard has more than one step, enable the Previous and Start buttons. Finally, if the step is neither the first step nor the last step, enable all of the buttons, remembering to allow the Finish button to be enabled only if the flag is set to allow the user to finish the wizard before navigating to the final step:

```
Private Sub SetForm(ByVal CurrentStep As Integer)
    If CurrentStep = 1 Then
        btnStart.Enabled = False
        btnPrevious.Enabled = False
        If mNumberOfSteps > 1 Then
            btnNext.Enabled = True
            btnFinish.Enabled = mFinishBeforeLastStepAllowed
        Else
            btnNext.Enabled = False
            btnFinish.Enabled = True
        End If
    ElseIf CurrentStep = mNumberOfSteps Then
        btnNext.Enabled = False
        btnFinish.Enabled = True
```

```
            If mNumberOfSteps > 1 Then
                btnPrevious.Enabled = True
                btnStart.Enabled = True
            End If
        Else
            btnNext.Enabled = True
            btnPrevious.Enabled = True
            btnStart.Enabled = True
            btnFinish.Enabled = mFinishBeforeLastStepAllowed
        End If
    End Sub
```

37. You need to set the text shown in the form's title bar and assign the appropriate values to the `Heading` label and the `Description` text box. Add these lines directly after the button state logic:

```
Me.Text = mWizardFormTitle + " - Step " + mCurrentStep.ToString + " of " + _
    mNumberOfSteps.ToString
lblHeading.Text = mSteps(CurrentStep).Heading
txtDescription.Text = mSteps(CurrentStep).Description
```

38. The only other part of the form that needs customizing besides the dynamically created components is the graphic on the left. Check whether the current step's `Graphic` object has an image loaded into it. If so, set the `BackgroundImage` property of the `pnlGraphic` control to that image. Otherwise, set it to the global graphic. Note that if no global graphic is defined, this simply resets the background image of the panel to blank:

```
        If mSteps(CurrentStep).Graphic Is Nothing Then
            pnlGraphic.BackgroundImage = mGlobalGraphicImage
        Else
            pnlGraphic.BackgroundImage = mSteps(CurrentStep).Graphic
        End If
```

39. The last part of the form that is customized based on the step being shown are the controls that are dynamically created and added to the `pnlControls` object you added to the main part of the form. Rather than do all the individual control work in the `SetForm` subroutine, you create four additional subroutines for the four control types — `AddCheckBox`, `AddRadioButton`, `AddTextArea`, and `AddComboBox`.

This means you need to iterate only through the `Components` array for the current `WizardStep` object and call the appropriate routine for each component. To cater to steps that do not have any `Components`, such as an introductory page, ensure that the `Components` object actually refers to something first:

```
        If mSteps(CurrentStep).Components IsNot Nothing Then
            With mSteps(CurrentStep)
                For MyCounter As Integer = 1 To .Components.GetUpperBound(0)
                    Dim ThisControlTop = mControlHeight * (MyCounter - 1)
                    Select Case .Components(MyCounter).ComponentControlType
                        Case AllowedControlTypes.CheckBox
                            AddCheckBox(.Components(MyCounter), ThisControlTop)
                        Case AllowedControlTypes.ComboBox
                            AddComboBox(.Components(MyCounter), ThisControlTop)
                        Case AllowedControlTypes.RadioButton
                            AddRadioButton(.Components(MyCounter), ThisControlTop)
```

```
                     Case AllowedControlTypes.TextArea
                         AddTextArea(.Components(MyCounter), ThisControlTop)
                 End Select
             Next
        End With
    End If
```

Each of the four `Add` subroutines accept two parameters: a `WizardComponent` object that contains all of the properties necessary to customize the control, and a value to set the top position of the control. The `Top` value is calculated based on the module-level variable you set in step 18 and is multiplied by the control's position in the array.

40. All of the `Add` routines follow a similar pattern, but because each control type is different, there are some variations as to how to set values or what controls are needed. The easiest one to create is the `CheckBox`. You set its `Name`, `Text`, and `Checked` properties from values found in the `WizardComponent` object and some position and size properties so that it is in the correct spot on the form.

41. The `Name` of each `CheckBox` is prefixed with a `CB` so that it's easy to determine each control's type when you're saving the values entered by the user. You could use a special piece of functionality called *reflection* to look at the object and determine its type, but it's just as easy to do it this way. When you've set all of the required properties, you add it to the `Controls` collection of the `pnlControls` object. The `AddCheckBox` routine appears as follows:

```
Private Sub AddCheckBox(ByVal ThisWizardComponent As WizardComponent, _
    ByVal ThisControlTop As Integer)
    Dim newCB As New CheckBox
    With newCB
        .Name = "CB" + ThisWizardComponent.ComponentName
        .Text = ThisWizardComponent.ComponentCaption
        If ThisWizardComponent.ComponentValue = "True" Then
            .Checked = True
        Else
            .Checked = False
        End If
        .Left = 0
        .Top = ThisControlTop
        .Width = pnlControls.Width
    End With
    pnlControls.Controls.Add(newCB)

End Sub
```

42. Adding `RadioButton` controls is almost exactly the same. The only difference is that you use a `RadioButton` control instead of a `CheckBox`, and you set only the `Checked` property if the `ComponentValue` is `Selected`. Otherwise, repeat the same code as previously shown.

43. A `TextBox` control is slightly different, which is why the subroutine is called `AddTextArea`. This is because you actually need two controls: a `Label` and a `TextBox`. The former is to tell the user what the latter is for.

You should note a couple of extra things in the following code. First, the label is prefixed with `LTB` to differentiate it from the other controls. Second, the size of the text shown in the label is calculated using the `MeasureString` method. This enables you to accurately position the `TextBox` so that it lines up against the `Label`. Finally, the `Label` control needs its `Top` value set

slightly lower than the `TextBox` so that the text in both aligns vertically. The final result is as follows:

```
Private Sub AddTextArea(ByVal ThisWizardComponent As WizardComponent, _
    ByVal ThisControlTop As Integer)
    Dim newLTB As New Label
    Dim newLTBTextSize As New System.Drawing.SizeF
    With newLTB
        .AutoSize = True
        .Name = "LTB" + ThisWizardComponent.ComponentName
        .Text = ThisWizardComponent.ComponentCaption
        .Left = 0
        .Top = ThisControlTop + 3
        newLTBTextSize = Me.CreateGraphics.MeasureString(.Text, .Font)
    End With
    Dim newTB As New TextBox
    With newTB
        .Name = "TB" + ThisWizardComponent.ComponentName
        .Text = ThisWizardComponent.ComponentValue
        .Left = newLTBTextSize.Width
        .Width = pnlControls.Width - .Left
        .Top = ThisControlTop
    End With
    pnlControls.Controls.Add(newLTB)
    pnlControls.Controls.Add(newTB)

End Sub
```

44. The last control type is the most complex — the `ComboBox`. Like the `TextBox`, it also requires a `Label`, but you also need to populate its `Items` collection with the `AllowedValues` you extracted from the XML definition.

Add the associated `Label` control as you did for the `TextBox` shown previously. The only additional code you need to add relates to the items in the `ComboBox` itself — you need to create a collection of items and set the `SelectedItem` property:

```
Private Sub AddComboBox(ByVal ThisWizardComponent As WizardComponent, _
    ByVal ThisControlTop As Integer)
    ...add a label similar to the TextBox one
    Dim newCM As New ComboBox
    With newCM
        .Name = "CM" + ThisWizardComponent.ComponentName
        .Text = ThisWizardComponent.ComponentValue
        If ThisWizardComponent.ComponentAllowedValues IsNot Nothing Then
            For ValueCounter As Integer = 1 To
ThisWizardComponent.ComponentAllowedValues.GetUpperBound(0)
                .Items.Add(ThisWizardComponent.ComponentAllowedValues(ValueCounter))
            Next
        End If
        .SelectedItem = .Text
        .Left = newLCMTextSize.Width
        .Width = pnlControls.Width - .Left
        .Top = ThisControlTop
    End With
    pnlControls.Controls.Add(newLCM)
    pnlControls.Controls.Add(newCM)
End Sub
```

How It Works — Control Event Handlers

45. You're just about done — believe me. The only parts of the `WizardBase` left to write code for are the event handler routines for the button clicks and several routines to store the values the user enters after completing a particular step.

The three navigation buttons — Next, Previous, and Start — all do the same thing but set the `WizardBase` to a different step in the sequence. The best way to approach this is to create a subroutine that accepts the step number to navigate to and have all of the three event handlers call it in turn.

Whenever users navigate from one step to another, you must first store the values they've entered in the current step. Once they've been saved, you can then clear the `Controls` collection of the `pnlControls` component to be ready for the next step, set the module-level variable to the requested step, and call the `SetForm` routine to set up the next step settings. The values are saved through the `StoreNewValues` subroutine you'll create in step 50. The following code defines this routine:

```
Private Sub NavigateToStep(StepNumber)
    StoreNewValues()
    pnlControls.Controls.Clear()
    mCurrentStep = StepNumber
    SetForm(mCurrentStep)
End Sub
```

Once you have this subroutine defined, the three `Click` event handlers are easy to create:

```
Private Sub btnNext_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnNext.Click
    NavigateToStep(mCurrentStep + 1)
End Sub
Private Sub btnPrevious_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnPrevious.Click
    NavigateToStep(mCurrentStep - 1)
End Sub
Private Sub btnStart_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnStart.Click
    NavigateToStep(1)
End Sub
```

46. There are two remaining buttons — Cancel and Finish. In Cancel's `Click` event, you should ask users if they're sure about canceling the wizard; and if they answer yes, set the `Cancelled` flag and close the form:

```
Private Sub btnCancel_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnCancel.Click
    If MessageBox.Show("Are you sure you want to cancel the wizard?", _
        "Confirm Cancel", MessageBoxButtons.YesNo, MessageBoxIcon.Question) = _
        Windows.Forms.DialogResult.Yes Then
        mCancelled = True
        Me.Close()
    End If
End Sub
```

47. The Finish button returns to the topic of this chapter—XML. Before closing the form and returning control to the application that called `WizardBase`, you need to create an `XmlDocument`, populate it with the values set by the user for every component in each step, and then save a `String` representation of it to return to the application.

First create the event handler routine and create a new `XmlDocument` object to do all your XML processing on. When the processing is finished, you need to assign the `InnerXml` value (a string representation of the XML in the `XmlDocument`) to the module-level `mWizardSettings` string that the application can retrieve through the read-only property, `WizardSettingValues`. Finally, close the form:

```
Private Sub btnFinish_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnFinish.Click
  Dim myXmlDocument As New XmlDocument

  mWizardSettings = myXmlDocument.InnerXml
  Me.Close()
End Sub
```

48. The `System.Xml` namespace comes with a whole raft of classes and associated methods designed to process XML easily, including the `XPath.XPathNavigator` object. Previous versions of .NET had limited functionality in this object, but the version of .NET that comes with Visual Basic Express has everything you need to add XML nodes to an `XmlDocument` object. To actually do the writing of the child nodes, you'll need an `XmlWriter` that is initialized with a new child node added by the `XPathNavigator`:

```
Private Sub btnFinish_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnFinish.Click
  Dim myXmlDocument As New XmlDocument
  Dim MyNavigator As XPath.XPathNavigator = myXmlDocument.CreateNavigator()
  Using MyWriter As XmlWriter = MyNavigator.PrependChild()

  End Using
  mWizardSettings = myXmlDocument.InnerXml
  Me.Close()
End Sub
```

Note the use of the `Using` statement. This tells Visual Basic Express to create the object temporarily and then dispose of it when the `End Using` statement is encountered. It's like combining the definition of an object and the use of a `With` statement, but with an added bonus that the object is destroyed when you're finished with it.

49. Write a loop to iterate through each `WizardStep` object in the `mSteps` array and then a subordinate loop to iterate through the individual components. Use the `WriteStartElement`, `WriteElementString`, and `WriteEndElement` methods of the `XmlWriter` object to add the values to the `XmlDocument` object, as shown here:

```
Private Sub btnFinish_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnFinish.Click
  Dim myXmlDocument As New XmlDocument
  Dim MyNavigator As XPath.XPathNavigator = myXmlDocument.CreateNavigator()
  Using MyWriter As XmlWriter = MyNavigator.PrependChild()
```

```
      MyWriter.WriteStartElement("WizardValues")
      For iStepCounter As Integer = 1 To mSteps.GetUpperBound(0)
        With mSteps(iStepCounter)
          If .Components IsNot Nothing Then
            MyWriter.WriteStartElement(.Name)
            For iComponentCounter As Integer = 1 To .Components.GetUpperBound(0)
              MyWriter.WriteElementString(.Components(iComponentCounter). _
                  ComponentName, .Components(iComponentCounter).ComponentValue)
            Next
            MyWriter.WriteEndElement()
          End If
        End With
      Next
      MyWriter.WriteEndElement()
    End Using
    mWizardSettings = myXmlDocument.InnerXml
    Me.Close()
  End Sub
```

How It Works — The Final Pieces

50. There are two subroutines left, both related to the storing of values as the user navigates away from a step. The first subroutine is called `StoreNewValues` and iterates through the `Controls` collection of the `pnlControls` component. For each control, the code finds the associated `WizardComponent` object by extracting the name from the `Control`'s name. It uses the `FindComponent` subroutine to do this:

```
Private Sub StoreNewValues()
  For Each CurrentControl As Control In pnlControls.Controls
    Dim myWizardComponentEntry As Integer = _
        FindComponent(CurrentControl.Name.Substring(2))
  Next
End Sub
```

51. If the corresponding `WizardComponent` is found (label controls won't be found, for example), then the `CurrentControl` object needs to be converted to the particular type of control that it is. Once the specific typed object has been created, then the most appropriate property is used to set the `ComponentValue` property in the `WizardComponent`, with the final code for this routine appearing as follows:

```
Private Sub StoreNewValues()
  For Each CurrentControl As Control In pnlControls.Controls
    Dim myWizardComponentEntry As Integer = _
        FindComponent(CurrentControl.Name.Substring(2))
    If myWizardComponentEntry > 0 Then
      With mSteps(mCurrentStep)
        Select Case CurrentControl.Name.Substring(0, 2)
          Case "CB"
            Dim myCB As CheckBox = CType(CurrentControl, CheckBox)
            .Components(myWizardComponentEntry).ComponentValue = _
              myCB.Checked.ToString.ToLower
          Case "CM"
            Dim myCM As ComboBox = CType(CurrentControl, ComboBox)
            If myCM.SelectedItem Is Nothing Then
```

```
                     .Components(myWizardComponentEntry).ComponentValue = myCM.Text
               Else
                  .Components(myWizardComponentEntry).ComponentValue = _
                   myCM.SelectedItem.ToString
               End If

            Case "RB"
               Dim myRB As RadioButton = CType(CurrentControl, RadioButton)
               If myRB.Checked = True Then
                  .Components(myWizardComponentEntry).ComponentValue = "Selected"
               Else
                  .Components(myWizardComponentEntry).ComponentValue = "NotSelected"
               End If

            Case "TB"
               Dim myTB As TextBox = CType(CurrentControl, TextBox)
               .Components(myWizardComponentEntry).ComponentValue = myTB.Text

         End Select
      End With
    End If
  Next
End Sub
```

52. The last subroutine is a simple one — `FindComponent`. All it does is iterate through the `WizardComponent` array for the current `WizardStep` object, looking for a `Component` whose names matches the parameter passed in. If it finds one, it returns the array position:

```
Private Function FindComponent(ByVal ComponentName As String) As Integer
    With mSteps(mCurrentStep)
        For ComponentCounter As Integer = 1 To .Components.GetUpperBound(0)
            If .Components(ComponentCounter).ComponentName = ComponentName Then
                Return ComponentCounter
            End If
        Next
    End With
    Return 0
End Function
```

53. Yes, you're done! Now all you need to do is test it. Open `Form1.vb` in Design view and add a `Button` to the form. In the `Click` event of the button, create an instance of the `WizardBase` form, set the `WizardDefinition` property, and show the form. Upon the return (that is, when the `WizardBase` form is closed), display a message box containing the values the user set in the wizard:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
        Dim frmMyExportWizard As New WizardBase
        Dim sUserExportSettings As String
        Dim sWizardDefinition As String = _
            My.Computer.FileSystem.ReadAllText("C:\MyWizard\WizardDefs.xml")
        With frmMyExportWizard
            .WizardDefinition = sWizardDefinition
            .ShowDialog()
```

275

```
            If Not .Cancelled Then sUserExportSettings = .WizardSettingValues
        End With
        frmMyExportWizard = Nothing
        MsgBox(sUserExportSettings)
    End Sub
```

54. As a sample file for the Wizard Definition, use the following XML. This XML defines a wizard you could use to customize the experience of the Export functionality you built into the Personal Organizer earlier in this chapter. The wizard has four steps with varying numbers of controls, and the last step specifies a custom image. This XML file (`WizardDefs.xml`) and associated images are available in the download for this book at www.wrox.com. If you don't have the images, create your own, or simply remove the graphic elements from the XML:

```xml
<Wizard Name="MyExportWizard" Title="Export Settings"
GlobalGraphic="C:\MyWizard\MyWizard.bmp" AllowFinishBeforeLastStep="True">
    <Step Name="Introduction">
        <Heading>Introduction</Heading>
        <Description>Welcome to the Export Settings Wizard</Description>
    </Step>
    <Step Name="ExportSettings1">
        <Heading>Export File Settings</Heading>
        <Description>Please choose the settings that best suit your
needs.</Description>
        <Component ControlType="TB" Name="Filename"
Caption="Filename:">C:\Temp\ExportData.xml</Component>
        <Component ControlType="CB" Name="OverwriteExisting" Caption="Overwrite
existing file?">False</Component>
    </Step>
    <Step Name="ExportSettings2">
        <Heading>Included Data</Heading>
        <Description>Select which kind of export you want to perform.</Description>
        <Component ControlType="RB" Name="Complete" Caption="All
information">Selected</Component>
        <Component ControlType="RB" Name="NamesOnly" Caption="Names
only">NotSelected</Component>
        <Component ControlType="RB" Name="NamesAndAddresses" Caption="Names and
addresses only">NotSelected</Component>
        <Component ControlType="CM" Name="DateFormat" Caption="Format of dates">
            <AllowedValue Name="YYYYMMDD" Selected="True">YYYY/MM/DD</AllowedValue>
            <AllowedValue Name="DDMMYYYY">DD/MM/YYYY</AllowedValue>
            <AllowedValue Name="MMDDYYYY">MM/DD/YYYY</AllowedValue>
        </Component>
    </Step>
    <Step Name="Completed">
        <Heading>Completed</Heading>
        <Description>The Export Settings Wizard is now completed. Click Finish to
close the wizard and have your data exported to the location
specified.</Description>
        <Graphic>C:\MyWizard\Completed.bmp</Graphic>
        <Component ControlType="CB" Name="SaveSettingsForFuture" Caption="Save
these settings for next time?">True</Component>
    </Step>
</Wizard>
```

55. Run the application and click the button on `Form1`. After a moment, during which the `WizardBase` form initializes all the classes based on the XML, it presents the wizard. You can change the settings and navigate through the wizard to your heart's content. Figure 12-3 shows how it looks in action.
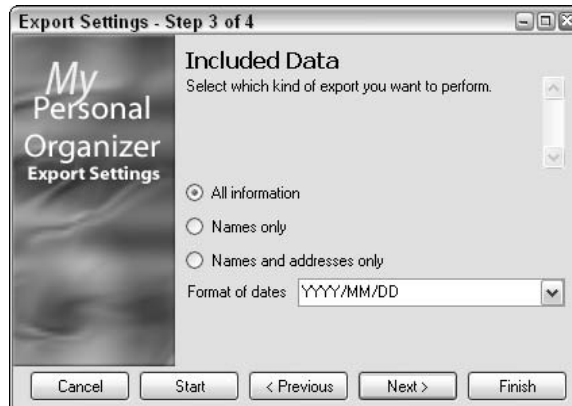
Figure 12-3

This `WizardBase` control can be used for all kinds of wizards. All you need to do is write an XML definition specifying the number of steps and what components should be shown in each step. The exercises at the end of this chapter will give you some additional ideas about what to do with it, but even they are just the tip of the iceberg.

Summary

XML is a powerful file format you can use to store pretty much anything. When it is used in conjunction with XSD, you can force data to comply with your own informational structure so that it won't hurt the internals of your application. Using the `System.Xml` namespace, you can create `XmlDocuments`, navigate them using `XPath`, and find specific parts of an XML file with `SelectSingleNode` and `SelectNodes`.

In this chapter, you learned to do the following:

❑ Harness the power of XML in different parts of your applications

❑ Transfer information to and from databases using the XML file format

❑ Use the `System.Xml` objects to build XML dynamically

In the next chapter, you will learn about security and encryption, two topics that work together to protect your application and its data from unwanted attacks.

Exercises

1. Add events to the Wizard form so the calling application knows when the user navigates between steps.

2. Add an optional attribute to the `TextArea` component in the Wizard form that enables you to insert a Browse for File dialog.

3. Create an XML Schema Document (XSD) to enforce the structure of the Wizard Definition XML file created in the last Try It Out.

13

Securing Your Program

While running your program on a local computer might work fine because you know what you're doing, an increasing number of applications can be executed over a network or even across the Internet. You'll see how you can deploy your own programs in the next chapter, but you first must understand the ramifications of the network boundaries your application must cross in order to be able to run successfully.

In this chapter, you learn about the following:

❑   Program security from both role and code viewpoints

❑   Encryption methods that can be used to protect your data

Program Security

Applications can be executed through a variety of means, some intentional, some not. If your program is something simple, such as a calculator, you might not care who runs it or what they do with it—after all, the functionality is generic and nonthreatening. However, if the application stores sensitive information, or calculates and updates important data, it might be a lot more important to control who has access to the information and functionality.

The decision about who can execute what can be considered from two different perspectives. First, you could decide that someone with a particular position, or *role*, has the authority to access the activities that your application can perform. The alternative is that the actual function itself, the *code*, is what drives access to the system and data.

These two different approaches to controlling access to the functionality and data in any given programming solution are both represented in the .NET Framework, and as a Visual Basic programmer you're able to harness both. Because security is actually quite a complex topic, this chapter introduces the concepts on which most programming security concerns are based and presents the theory behind the approaches with some small examples of how they work in code. For more advanced coverage of security for programs based on the .NET Framework (as Visual Basic Express applications are), take a look at the *Visual Basic .NET Code Security Handbook* by Eric Lippert (Wrox, 2002).

Role-Based Security

As mentioned earlier, sometimes you might want to control who has access to an application, or parts of its functionality, based on the role of the user. A manager might have the capability to approve a pay increase, but the secretary doesn't have that same capability. However, a secretary could be approved to order office supplies, whereas a tech support person could only check their status.

Using role-based security in your Visual Basic Express program enables you to specify these multiple levels of approval within your application's functionality. You could even allow different users access to the same functionality but with different limits based on their role—for example, the manager could withdraw $500 petty cash, the secretary could take $100, and the tech support person only $10.

To use role-based security, your application needs access to the information that Windows makes available about the current user. In fact, Windows allows an individual to access different applications under different user accounts, so it returns the information about the user account that is being used for the current application's process.

This information concerns what is known as an *identity*. An identity is usually based on a Windows account but it doesn't have to be—as long as Windows knows how to interrogate it and the authority it has, it can be represented as an identity.

The .NET Framework, on which Visual Basic Express is based, gives you access to the identity through a `Principal` object. This object is what you can use to determine a particular identity's access privileges. This is done by determining the roles to which the Principal object belongs.

Each role is a defined group of access privileges. For example, you might have a role of "CanPrint," which allows printing functionality, and another role of "CanWithdraw," which allows access to the petty cash account. When an identity is created, it is assigned certain roles, so a user who is a manager might have both the CanPrint and CanWithdraw roles, while the tech support person has access to only the CanPrint role (see Figure 13-1). Your program can check whether the `Principal` object associated with the current user running your application belongs to the specific role you need before continuing to allow access to the functionality.

Role-based security in the .NET Framework is performed through permissions in conjunction with the Principal, specifically `PrincipalPermission`, objects that can do the authorization checking for you. However, you can abbreviate the whole process of retrieving the Principal object and the roles to which it belongs by using Visual Basic Express and the `My` namespace. In Chapter 8, you saw a very limited example of this kind of checking of roles with the following example:

```
With My.User
    If .IsAuthenticated Then
        Me.Text = "Personal Organizer - logged in as " & .Name
        If .IsInRole("BUILTIN\Administrators") Then
            btnViewOptions.Visible = True
        Else
            btnViewOptions.Visible = False
        End If
    End If
End With
```
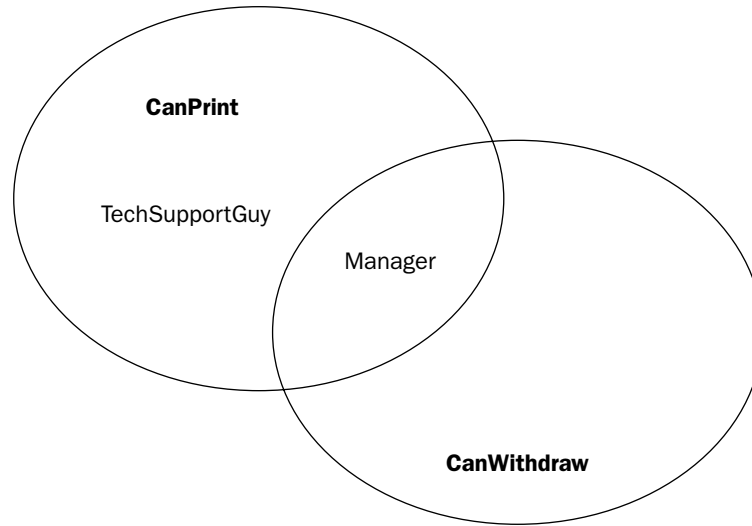
Figure 13-1

This code is using the shortcuts provided through the My namespace to directly access the current user. Because you most likely won't ever need to access the credentials about any other user during the execution of your application, this shortcut is immensely useful.

Breaking down the code reveals the following actions:

❑ **My.User** returns the Principal object related to the currently logged on user.

❑ **IsAuthenticated** is a Boolean flag that indicates whether the Principal is properly logged on or whether it needs authentication.

❑ **IsInRole** is another Boolean flag, but this one tells you whether the current Principal belongs to a specified role.

In this example, the IsInRole method explicitly states the role name. This is particularly useful for your own custom-built roles, but the built-in roles that Windows creates can be accessed through an enumeration of BuiltInRole:

❑ **AccountOperator** — Has responsibility for creating and maintaining user accounts

❑ **Administrator** — Complete and unrestricted access to the computer

❑ **BackupOperator** — Members of this role can perform backup operations.

❑ **Guest** — The most limited role group; guests can perform only a small group of actions

❑ **PowerUser** — Someone who has more responsibility than a regular user but not complete access, such as an Administrator

❑ **PrintOperator** — Allows control over printers

❑ **Replicator** — For network domain replication

❑　**SystemOperator**—Can operate the current computer

❑　**User**—The role to which normal users are assigned

A Closer Look at Identity and Principal

It's helpful to understand the differences between Identity and Principal objects so that when you write your programs, you can use the correct one. At their core, Identity objects represent individual users, and roles are the groups to which those identities can belong. A Principal object incorporates both of these concepts to return both the identity and role in which your application is interested.

Identity objects have a name and authentication type. As you will normally be creating standard Windows applications, you'll be most interested in the `WindowsIdentity` object, which represents an identity from a Windows authentication context. You'll be able to retrieve the identity's name and whether that user is authenticated when your application looks at the object.

When your code is actually executing, there is always a Principal object that identifies the security context under which it is running. The main class to use, if you're not going to use the shortcuts provided through the `My` namespace, is the `WindowsPrincipal` class, found in the `System.Security.Principal` namespace.

Using this class, you'll find that it has a property called `Identity` that returns the current identity object associated with this principal, and an `IsInRole` function that helps you identify the roles to which the identity belongs.

To illustrate how you can use roles in your code, the following Try It Out enables the state of buttons based on the roles to which the current user belongs.

Try It Out　Using Role-Based Security

1. Start Visual Basic Express and create a new Windows Forms application. Add two buttons to the form and label them Print and Withdraw.

2. Create an event handler routine for the `Load` event of the `Form` by double-clicking anywhere in the form. Add the following code:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    With My.User
        If .IsInRole(ApplicationServices.BuiltInRole.User) Then
            Button1.Enabled = True
        Else
            Button1.Enabled = False
        End If
        If .IsInRole(ApplicationServices.BuiltInRole.BackupOperator) Then
            Button2.Enabled = True
        Else
            Button2.Enabled = False
        End If
    End With
End Sub
```

3. Run the program. If the user belongs to the User role, the Print button will be enabled, and if the user belongs to the Administrator role, the Withdraw button will be enabled. Figure 13-2 shows the result if the user doesn't belong to the Administrator role.

Figure 13-2

Code-Based Security

Visual Basic Express programs have an alternative to limiting their functionality based on the role groups to which the user belongs — *code-based security*, sometimes called *code access security*. Because programs can be executed from almost anywhere, computer systems are often locked down so tightly against viruses and worms that normal applications have little hope of having the access to the system they need to do their work.

At the other end of the scale, if a user has membership in one or more of the more powerful roles such as the Administrator group, there's little stopping them from running any kind of program they want, which could indeed lead to unexpected activities that could destroy important information on the computer.

Besides the intentional malicious code found in programs like viruses, there's always the chance of an unintended bug in the code that could damage the computer system in some way. Windows by itself usually allows programs of any kind to run if they've been installed locally and blocks access to system functionality only if the application is running over the Internet via web scripts.

The .NET Framework comes with code access security procedures built right into the bodywork of the system. Every bit of code is signed with a particular category of access that it expects to have, so you can have total control over what you expect from each program. If you know your application doesn't ever access the file system, then you can set up the code access protocols for it to exclude that security requirement. Then, if a user is running your application and it has been compromised by a virus or worm that tries to access the file system, the attempt is foiled because the application doesn't have access.

This also benefits you as the designer of the application by protecting against unwanted results due to bugs in your code.

Having control over your program's access in this way enables system administrators to create a corresponding security policy that can associate the correct permissions with the code. They can even gather applications that require the same functionality into what's known as a *code group* and change the access on everything at once.

When the code executes, it can ask for the permission it needs and, if rejected, handle the exception gracefully. This enables you to write your program so that it performs certain functionality only when it has the right set of permissions, or ends completely if the necessary permission sets are not allowed.

You'll see more about code access security in Chapter 14 when dealing with deployment. As you'll see, when you configure your ClickOnce deployment project, you can specify what access your application expects to have and build the required code access permissions right into the installer.

Cryptography and Encryption

While application security provides control over who can access what functionality in your program, it doesn't necessarily protect your data from unauthorized access. Instead, you need to protect your information in some other way, which is where the science of cryptography comes in. *Cryptography* is the process used to secure data so that it cannot be changed or retrieved without the other person knowing how to extract the information.

This enables you to transfer your data over an insecure path such as the Internet and it can also be stored in a computer file so that anyone with access to your computer doesn't automatically have access to the information on it. Using cryptographic processes, you can *encrypt* data at one end, transfer it to the destination, and *decrypt* it at the other. Anyone looking at the data between the encryption and decryption phases is not be able to read it easily, if at all, without having the key to unlock the algorithms used to scramble the information.

You have a number of ways to encrypt data, and the option you choose is usually dependent on how the encrypted data is going to be used. If you're using it internally, then you don't need to publish a key to the data, and you can store the decryption routines within your program; however, if you need to send the data to someone else, then you need to give them the information they need to unscramble the contents.

All encryption processes fall into four broad categories, called *cryptographic primitives*. You must decide which of these primitives, or a combination of them, to use for your particular requirements. In order of usage, they are as follows:

❑ **Secret Key Encryption** — Also known as *symmetric cryptography*, secret key encryption involves the use of a single key that is used to both encrypt and decrypt the data. This key is used to transform the data itself and is normally kept private, hence the name.

❑ **Public Key Encryption** — An increasing number of applications need to transfer their data to another application that is only partially trusted. The sending program doesn't want to expose the information used to encrypt its data and so it uses a set of two keys — a private key for encrypting the data and a public key for decrypting the information. This is known as *asymmetric cryptography*.

❑ **Signing** — Rather than encrypt the data itself, you can choose to use a signing process that generates a unique digital signature for the information and the sender. It can be used as a verification process to ensure that it really was the expected person who sent the data.

❑ **Hashing** — Hashing data is a process that has been around for a very long time. It transforms the data into a fixed-length byte array that is unique and can be unhashed without the data being changed in any way.

Of these four categories, everything ultimately boils down to either secret or public key encryption, so these two methods of cryptography are worthy of a closer look.

Secret Key Cryptography

Probably the most common way of protecting sensitive data is to use secret key encryption. A single secret key value is used to both encrypt and decrypt the information. This means that anyone with the secret key value can extract the information, so it's important that you carefully consider where to store the secret key in this situation.

Using a secret key, a symmetric cryptographic provider such as Rijndael, TripleDES, or RC2 encrypts the data one block at a time. Doing this enables them to run extremely fast, as the blocks used are typically quite small—usually less than 32 bytes each.

As each block is encrypted, it uses a special process called *cipher block chaining* (*CBC*) to chain the data together. The CBC uses the secret key in combination with another special value called the *Initialization Vector* (usually abbreviated to *IV*) to do the actual transformation of the data to and from the encrypted form.

The Initialization Vector is used to ensure that duplicate blocks are encrypted into different forms, thus confusing the output even further. If the same IV value were used for every block being encrypted, the original content of two identical blocks would be encrypted into the same form. An unauthorized application could use this as a basis for determining common characteristics about your encrypted data and potentially determine the secret key's value.

The IV is used by the cipher block chaining process to link the information in a previous block into the encryption of the next block, thus producing different outputs for text that was originally the same. The IV is also used to perform a similar process on the first block, so depending on the rest of the data, even common first block content will be different.

Visual Basic Express can use any of the secret key encryption algorithms that the .NET Framework provides, of which there are four: `DESCryptoServiceProvider`, `RC2CryptoServiceProvider`, `RijndaelManaged`, and `TripleDESCryptoServiceProvider`. You'll use this last encryption method in the Try It Out at the end of this section to encrypt and decrypt the password string in the Personal Organizer application.

The problem with secret key encryption is that the two sides of the cryptographic equation must have the same key and IV. If the two processes are in separate applications, and have to communicate these values to each other somehow, there is a chance that the secret key values can be intercepted. That's why there is an alternative—public key encryption.

Public Key Cryptography

Public key encryption uses two keys to do the cryptographic transformations. The two keys work hand in hand to encrypt and decrypt data. You have a *private key* that is known only to yourself and other authorized users, but the *public key* can be made public so that anyone can access it.

The public key is related to the private key through mathematical equations—what the equations are depends on the particular encryption provider you use—and data that is encrypted with the public key can be decrypted only with the private key, while data transformed by the private key can be used only by those who have the public key in their possession.

Typically, you would use public key encryption if you were dealing with another party that is not part of your internal organization. In this case, too many factors in communicating the private key to the other party could be broken down, so the public key alternative is much better — only you can create the data using the private key, so when the other application tries to decrypt it using your public key, it is successful only if it was sent by you. However, that's not the best way to use this kind of cryptography.

The trick to public key encryption is that both parties have their own pair of private and public keys. Therefore, Person A gives Person B his public key, while Person B gives Person A her public key. When they want to send information to each other, they use the other person's public key, knowing that it can be decrypted only by the private key held by that person (see Figure 13-3).
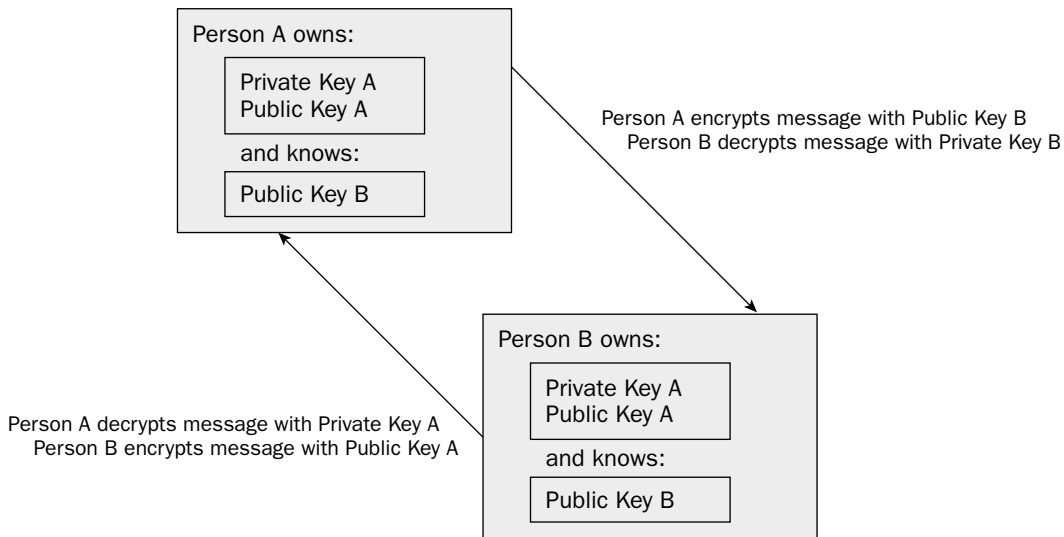
Figure 13-3

Visual Basic Express has access to two types of public key encryption through the `DSACryptoService Provider` class and the `RSACryptoServiceProvider` class.

Because encryption is quite complex to understand, the following Try It Out walks you through the process of creating encryption and decryption routines for the Personal Organizer application. You'll use these to encrypt the password of the user when it's stored in the database, but the general techniques discussed here can be applied to most other situations that warrant encryption.

Try It Out   Encrypting a Password

1.   Start Visual Basic Express and open the Personal Organizer application you've been working on throughout the book. If you haven't completed all of the exercises, you can find an up-to-date version of the project in the `Code\Chapter 13\Personal Organizer Start` folder of the downloaded code you can find at `www.wrox.com`.

2.   Open the `GeneralFunctions.vb` module. This is where you'll create the `EncryptString` and `DecryptString` functions. Normally, you would store the keys that define the encryption

elsewhere so they cannot be decompiled out of your program, but for this sample, store the Initialization Vector and the secret key values in the application itself so it's easier to see what's going on.

3. Because you are using several IO- and Security-related functions, add two new `Imports` statements at the top of the code module. In addition, define the Initialization Vector at this point as an array of `Bytes`. These values can be any kind of hexadecimal values — the sample here works fine if you don't want to create your own:

```
Imports System.Data
Imports System.IO
Imports System.Security.Cryptography

Module GeneralFunctions
    Private myDESIV() As Byte = {&H12, &H34, &H66, &H79, &H91, &HAB, &HCD, &HEF}
```

4. Create a new function called `EncryptString`. Have it accept two string parameters for the text to be encrypted and the encryption key to use and a return value of a string that contains the encrypted text. Because encryption can sometimes cause errors if everything isn't just right, wrap the entire process in a `Try` block:

```
Public Function EncryptString(ByVal PlainTextString As String, _
  ByVal EncryptionKey As String) As String
  Try

  Catch exCryptoError As Exception
    Return exCryptoError.Message
  End Try
End Function
```

When you initially create this function, Visual Basic Express displays a warning indicator underneath the End Function *statement. This is because it has recognized that under some conditions, the function does not return a string value to the calling code, which could potentially cause errors. This warning will be displayed until all possible paths through the code return a value.*

5. Check the encryption key parameter. Because you are going to use TripleDES as the encryption algorithm, you need a key of 24 bytes, so if the string is anything less than that, exit the function with an error. Otherwise, convert the string to an array of `Bytes` to use in the cryptography functions:

```
Public Function EncryptString(ByVal PlainTextString As String, _
  ByVal EncryptionKey As String) As String
  Try
    Dim DESKey() As Byte = {}
    If EncryptionKey.Length = 0 Then
      Return "Error - Key must be supplied"
    Else
      DESKey = System.Text.Encoding.UTF8.GetBytes(EncryptionKey.Substring(0, 24))
    End If
    ... the code to perform the encryption will go here
  Catch exCryptoError As Exception
    Return exCryptoError.Message
  End Try
End Function
```

You'll notice that the conversion of the string to a `Byte` array uses the `System.Text.Encoding` namespace to convert the string contents. This Try It Out uses UTF8 as the text format, but you could use Unicode instead. Either way, the aim is convert the string to a fixed array of byte values, and you need to use the `GetBytes` function to do this.

6. This encryption function is going to use the TripleDES encryption algorithm. TripleDES stands for Triple Data Encryption Standard, a common encryption standard. To use the encryption, you first must define an instance of the appropriate `Provider` object, which you pass into a `CryptoStream` object to perform the actual encryption. Define the TripleDES provider directly after the `End If` and before the `Catch` statement:

```
Dim CSPSym As New TripleDESCryptoServiceProvider
```

7. You also need to convert the text that is to be encrypted into another array of byte values, because all encryption methods use byte arrays to do the processing. You can use the same `GetBytes` method immediately after the declaration of `CSPSym`:

```
Dim inputByteArray() As Byte = _
    System.Text.Encoding.UTF8.GetBytes(PlainTextString)
```

8. When you pass the bytes to be encrypted into the cryptography functionality, you need something to store the output. You can use any kind of `Stream` object for this purpose, and if you were going to be writing a significant amount of data, you could write it to a file, or even an XML document. However, because you're going to encrypt only the password, and do everything internally within the program, you can use a simple `MemoryStream` to keep the output.

A `MemoryStream` object is, as you might guess, an object that stores the information in memory and knows nothing about file structures or writing to disk. It can be found in the `System.IO` namespace but because you used an `Imports` statement for that namespace, you can define it like so:

```
Dim EncryptMemoryStream As New MemoryStream
```

9. To complete the setup, you need to create a `CryptoStream` that does the encryption transformation. The `CryptoStream` object needs a stream that contains the data to be encrypted (and after the encryption has occurred, the output), the type of cryptography function to be performed on the stream, and the mode, to indicate whether you are encrypting the data (Write mode) or decrypting the data (Read mode):

```
Dim EncryptCryptoStream As New CryptoStream(EncryptMemoryStream, _
    CSPSym.CreateEncryptor(DESKey, myDESIV), CryptoStreamMode.Write)
```

The second parameter of this object's instantiation is created by calling the `CreateEncryptor` method of the `TripleDESCryptoServiceProvider` object you defined earlier, passing in the secret key and initialization vector information. This is the core of the encryption process. Without a correct key or vector, the encryption does not work as expected.

10. You can now use the `CryptoStream` object in much the same way as you would any other stream object. Call the `Write` method to pass in the plaintext. Because you're encrypting a simple string, you can do this in one pass, specifying the entire length of the byte array to be written all at once. Because you're writing this to memory, you'll need to tell Visual Basic Express that you've finished writing to the `CryptoStream` by calling `FlushFinalBlock`:

```
EncryptCryptoStream.Write(inputByteArray, 0, inputByteArray.Length)
EncryptCryptoStream.FlushFinalBlock()
```

11. Your original plaintext has now been encrypted, and you can return it to the calling code. However, because the string could contain unprintable characters and you might choose to store this encrypted string in a file that might not accept extended character sets, you should first convert it to base 64. This is particularly useful if the ultimate endpoint for the encrypted string is an XML file.

```
Return Convert.ToBase64String(EncryptMemoryStream.ToArray())
```

The final function should look like this:

```
Public Function EncryptString(ByVal PlainTextString As String, _
  ByVal EncryptionKey As String) As String
  Try
    Dim DESKey() As Byte = {}
    If EncryptionKey.Length = 0 Then
      Return "Error - Key must be supplied"
    Else
      DESKey = System.Text.Encoding.UTF8.GetBytes(EncryptionKey.Substring(0, 24))
    End If

    Dim CSPSym As New TripleDESCryptoServiceProvider
    Dim inputByteArray() As Byte = _
        System.Text.Encoding.UTF8.GetBytes(PlainTextString)

    Dim EncryptMemoryStream As New MemoryStream
    Dim EncryptCryptoStream As New CryptoStream(EncryptMemoryStream, _
        CSPSym.CreateEncryptor(DESKey, myDESIV), CryptoStreamMode.Write)
    EncryptCryptoStream.Write(inputByteArray, 0, inputByteArray.Length)
    EncryptCryptoStream.FlushFinalBlock()

    Return Convert.ToBase64String(EncryptMemoryStream.ToArray())

  Catch exCryptoError As Exception
    Return exCryptoError.Message
  End Try
End Function
```

12. You can now create the `DecryptString` function that takes the encrypted string and processes it back into plaintext. The function is almost identical to `EncryptString` except that it first converts from a base-64 string into a byte array and to return a readable UTF8 string upon return. The only other difference is in the creation of the `CryptoStream` object, where you need to call the `CreateDecryptor` method to specify what kind of transformation should be performed. The full function appears as follows (with the lines that differ highlighted):

```
Public Function DecryptString(ByVal EncryptedString As String, _
  ByVal EncryptionKey As String) As String
  Try
    Dim DESKey() As Byte = {}
    Dim inputByteArray(EncryptedString.Length) As Byte

    If EncryptionKey.Length = 0 Then
      Return "Error - Key must be supplied"
    Else
      DESKey = System.Text.Encoding.UTF8.GetBytes(EncryptionKey.Substring(0, 24))
    End If
```

```
      Dim CSPSym As New TripleDESCryptoServiceProvider

      inputByteArray = Convert.FromBase64String(EncryptedString)

      Dim DecryptMemoryStream As New MemoryStream
      Dim DecryptCryptoStream As New CryptoStream(DecryptMemoryStream, _
          CSPSym.CreateDecryptor(DESKey, myDESIV), CryptoStreamMode.Write)

      DecryptCryptoStream.Write(inputByteArray, 0, inputByteArray.Length)
      DecryptCryptoStream.FlushFinalBlock()

      Return System.Text.Encoding.UTF8.GetString(DecryptMemoryStream.ToArray())

    Catch exCryptoError As Exception
      Return exCryptoError.Message
    End Try
  End Function
```

13. For this Try It Out, you change the `UserPasswordMatches` and `CreateUser` functions to call the `EncryptString` or `DecryptString` methods to get the appropriately formatted string. As mentioned earlier, you would normally keep the secret key elsewhere in the code, but for this example, you keep it in the functions themselves.

14. Locate the `UserPasswordMatches` function in `GeneralFunctions.vb`. Previously, you simply compared the `Password` field in the database to the password the user entered, but now you use the `DecryptString` function to first convert the database password to plaintext. Locate the line where the comparison is performed. It will look like this:

```
            If .Item(0).Item("Password").ToString.Trim = Password Then
```

Replace this code with a call to `DecryptString`. You first need to define a string variable that contains a 24-character secret key. You should then check the return value of the function against the password value the user entered:

```
Dim SecretKey As String = "785&*(%HUYFteu27^5452ewe"
Dim DecryptedPassword As String = DecryptString( _
    .Item(0).Item("Password").ToString.Trim, SecretKey)
If DecryptedPassword = Password Then
```

15. Edit the `CreateUser` function so that it encrypts the password before storing it in the database. Locate the line of code that adds the new record to the `POUser` table (the `AddPOUserRow` function). Change it so that it passes over the encrypted password string instead. You need to define the same secret key (otherwise, the decryption in `UserPasswordMatches` won't work!) and call `EncryptString` to perform the transformation:

```
Dim SecretKey As String = "785&*(%HUYFteu27^5452ewe"
Dim EncryptedPassword As String = EncryptString(Password, SecretKey)

CreateUserTable.AddPOUserRow(UserName, UserName, EncryptedPassword, Now, Now, 0)
```

16. You can now run the program, but you'll most likely find that you cannot get past the login screen. This is because the `UserPasswordMatches` function is expecting the password fields in the database to be already encrypted, but you've got plaintext passwords in there.

To get past this, add the database to the Database Explorer and remove the row that contains your user information. Next time you start the program, it prompts you to create a password as a new user and subsequently encrypts the password into the database.

Summary

Securing your program and data is essential in today's computing environment. You need to tell your users what kind of access your application needs so that it can execute correctly, and you also need to protect your data from external factors that could retrieve it for unwanted uses. With careful application of role- and code-based security mechanisms, you can ensure that your program runs with the required permissions and that unauthorized users are not able to access it. Encryption algorithms exposed by the .NET Framework can be used in Visual Basic Express to scramble your data.

In this chapter, you learned to do the following:

❑ Analyze your program for appropriate security mechanisms and choose role- or code-based security for any given application

❑ Encrypt your sensitive data so that it cannot be retrieved by unwanted parties

Exercise

1. Although decrypting the password from the database might work for comparing it to the string the user has entered, it's not as secure as it could be. Change the logic so that the `UserPasswordMatches` function encrypts the entered string and compares it to the already encrypted database field to ensure that the fields match.

14

Getting It Out There

All of the information you've learned so far has helped you create some great applications, but there's a slight problem — they're all still sitting on your own computer. If you want someone else to be able to run the program, you need to be able to get it to them.

Deployment of Visual Basic Express programs is very straightforward. In fact, you could simply copy the application file to another computer and chances are good it will run without a problem if the computer keeps current with the latest Windows Updates. But Visual Basic Express comes with additional tools to build a proper installation program for your projects, including ClickOnce deployment.

In this chapter, you learn about the following:

❑   Installing your programs to another computer

❑   Using ClickOnce to deploy your application via the web

❑   Creating additional settings to enable your applications to automatically update

Installing the "Hard" Way

Visual Basic Express programs are ready to be run as soon as you've built them. When Visual Basic Express compiles the project, it creates an application file along with the necessary configuration files (if needed at all) in either the Debug or Release subfolders of the project's bin directory. (This is dependent on your project settings and the main options page in Visual Basic Express.) The options for building the project can be found by selecting Projects and Solutions ⇨ Build and Run from the Options dialog of Visual Basic Express, which is visible only when you have the Show All Settings option checked.

To enable it to run on another computer, all you need to do is copy these files to a location on the destination computer and run the main executable. If you have an application that is more compli-cated and requires additional files, you just need to include these extra files when you do the copy process.

Visual Basic Express programs depend on the .NET Framework version 2.0. However, if you try to run an application on a computer system that does not have the correct version of the Framework installed, it will end cleanly with a simple message informing the user that the appropriate version must be installed. Also included with the message is the version information so the user can find and install it properly.

If you don't believe it's this simple, create a standard Windows Forms application, put a button on it, and use the `MessageBox` command to display "Hello World." Build the project and run the application to ensure that it works as you expect. Then, locate the `.exe` file in the `bin\Debug` folder in the project directory, copy it to another computer via disk or network, and run the application on the destination computer.

If the computer has the correct version of the .NET Framework installed, you will be able to run the application without error (see Figure 14-1), and clicking the button will produce the expected message dialog box. Otherwise, you'll get an error message telling you to install the proper version of the .NET Framework. You can even e-mail the application to someone and they can run it immediately.

Figure 14-1

The problem with this method is that for more complex projects, you run the risk of missing an important file, and if you use more advanced techniques such as web services or database access, you might not even realize that the file you need is not present. Fortunately, Microsoft anticipated this and included a new deployment technology with Visual Basic Express to ease the process of installation—ClickOnce.

Just ClickOnce

While copying the files you need using normal Windows methods might sound straightforward, ClickOnce deployment makes it even easier. Using ClickOnce, you can create a setup package, complete with web page, that enables people to download and run your application over the network or Internet. You can even have the application accessible only from the website on which you store it, so if the user is not logged on, they won't be able to run it at all.

ClickOnce does all the hard work for you, including monitoring for updates, ensuring that the user has the correct version of the software, and automatically updating it if need be. In addition, ClickOnce ensures that each application is self-contained and therefore not affected by another application's installation.

Previous installation options used another technology known as Windows Installer. Windows Installer did indeed help automate the deployment process but it had some issues that tended to make the end user experience more cumbersome than it should have been. The top two problems with Windows Installer were the updating process and security concerns:

❑ When Windows Installer applications were installed, any time an update was applied, the application had to be completely reinstalled. The best option was to ship a new update installer that applied changes right across the application so that the new files were integrated with the old files. ClickOnce can apply any changes to the application automatically; and by default, only updated parts of the program will be reinstalled through the process.

❑ To install an application using Windows Installer, the user had to be an administrator or have administrator privileges, even if the application itself didn't need them. Using ClickOnce, you can specify the level of security access the application requires, thus enabling users without administrator privileges to control the installation.

ClickOnce capitalizes on previous advances made in technology that enabled applications to run over the network or web, and optionally enables you to deploy your program in such a way that it doesn't require any files at all to be installed on the user's computer. Doing this requires that the user have a constant connection to the server that hosts the application files, but it means that any updates to the project are automatically flowed through to the end users the next time they run the application, without any installation process being required at all.

Alternatively, publishing your ClickOnce application to a CD or normal file location enables you to distribute the program in more traditional ways to the users. In this situation, you can include an autorun file so that the CD automatically starts the setup procedure when inserted into the user's CD drive.

To illustrate the simplicity of deploying your application using ClickOnce, the next Try It Out walks through the creation of a simple application and the deployment of the application to a website. It shows you how easy it is to install, run, and uninstall your Visual Basic Express applications.

Try It Out   Using ClickOnce

1. Start Visual Basic Express and create a new Windows Application project. Name it `ClickOnceTestApp` so you can find it later. Make sure you save the project as well.

2. Open the My Project page and click the Publish tab to view the ClickOnce deployment options. Click the Updates button to display the update options for this project. Make sure the checkbox for "The application should check for updates" is selected, as shown in Figure 14-2, and click OK to save the setting.

3. Publish the application without making any changes to it. To use ClickOnce deployment, you can either right-click the project in the Solution Explorer and choose Publish, or run the Build ⇨ Publish ClickOnceTestApp menu command.

4. After a moment, the Publish Wizard starts. First you must choose the location for the installation files. By default, Visual Basic Express chooses a local web server location, but you can override this to send the installation directly to a remote FTP site or network location, or even to the normal file structure of your computer.

    If you choose to create the installation on the local file system, the wizard will also prompt you to specify how users will ultimately install the application so it knows what supporting files it needs to include. If you choose anything else, such as the default web server location, it will assume the appropriate setup (in this case, a web setup).

Figure 14-2

Leave the installation location as the default and click Next. At this point, you need to choose whether the application runs over the network or Internet or whether it is installed on the local machine so the user can run it without being connected. This latter option is the default, so click Next to continue.

5. A summary page is displayed reminding you of your options and what happens next. Click Finish to close the Publish Wizard and commence the building process. Visual Basic Express first recompiles the application project and then assembles all the necessary files into a `setup.exe` ready for installation.

6. Once it's done, it copies that file, along with all the required files to enable the setup process to work, to the specified location. When this copy process is complete, it shows the default installation page ready for installation (see Figure 14-3). By default, it creates the page content based on your system and Visual Basic Express settings, but you can override these settings manually (you'll see how to do that later in this chapter).

7. Install the application by clicking the Install button. The ClickOnce deployment process first verifies that it has all the necessary application files (see Figure 14-4) and then launches the installation. The verification process is particularly important for subsequent installations because it is this process that can also check for updates.

Once the solution has been installed, the program is automatically started, and you see the blank form you created at the beginning of this Try It Out. A shortcut is also added to the Start menu so that the program can be run at a later date.

8. The application doesn't do much yet — in fact, it just sits there — so the next few steps show you how easy it is to update the application to do something. Stop the application from running and return to Visual Basic Express.
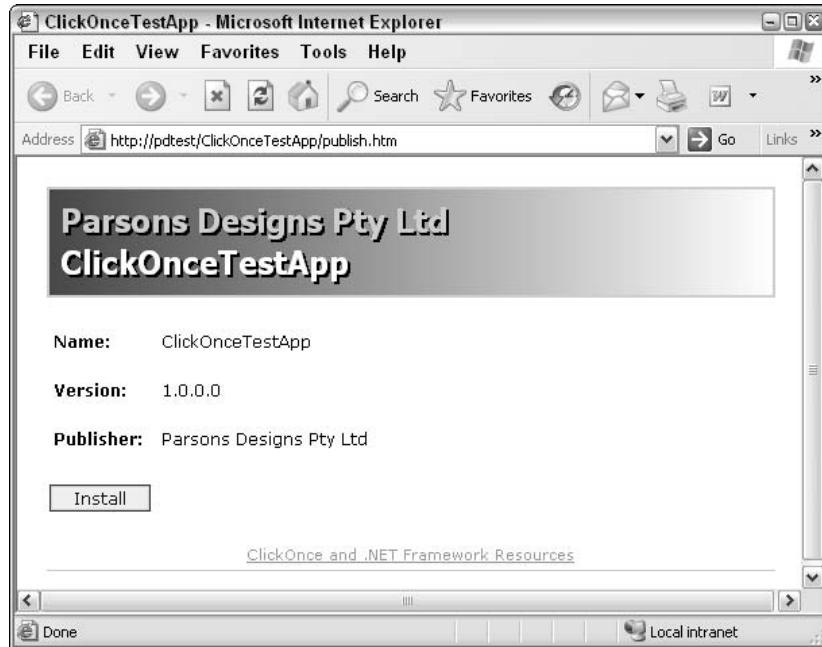
Figure 14-3
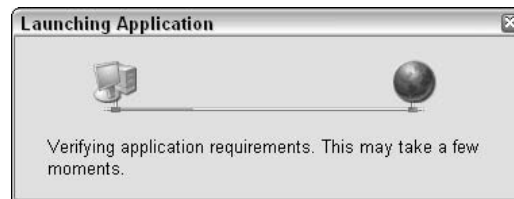
Figure 14-4

9. Add a button to the form and create an event handler for the button's `Click` event. Add a command to show users a message box when they click the button:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    MessageBox.Show("Hello World")
End Sub
```

10. Save the project and publish it again using the same default options. This time, when the installation web page is displayed, you should see that the version number has been incremented to indicate that there is a new version to install.

11. Rather than click the Install button to explicitly do the update, run the ClickOnceTestApp shortcut you find in the Start menu to run the application in the same way a user normally would. Because of the Updates setting you selected in step 2, when the application starts, it checks for any updates first (see Figure 14-5).
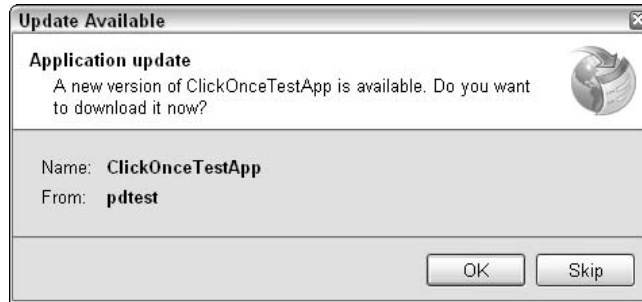
Figure 14-5

If you click Skip, the old version of the application without the button is executed, so click OK instead to update the application with the changes you made. ClickOnce automatically copies the changed files to the installation folder on the computer and runs the new version of the application.

12. Uninstalling a ClickOnce application is just as easy. Bring up the Add or Remove Programs dialog you find in the Control Panel and scroll through the list of installed programs until you find ClickOnceTestApp.

13. Select the entry and click the Change/Remove button. A simple installation dialog is displayed by your ClickOnce solution, enabling you to restore the application to a previous installation, or to remove the application entirely (see Figure 14-6).

14. Select "Restore the application to its previous state" and click OK. The installation process undoes the last set of changes to the application; and if you run the program again, you are presented with the form without a button.

15. Return to the Add or Remove Programs dialog and this time remove the application completely (the Restore option should no longer be available because no more updates are installed).
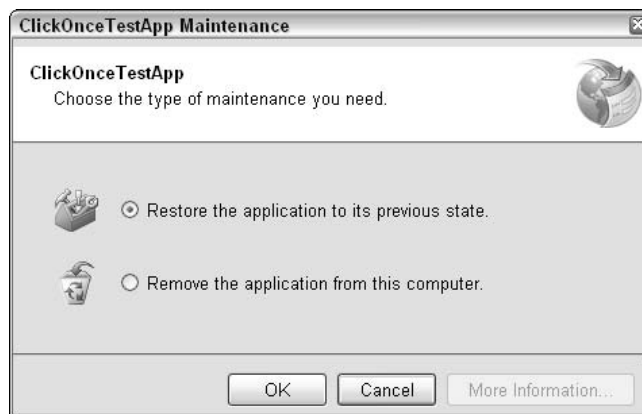
Figure 14-6

ClickOnce Options

Now that you've seen how easy it is to incorporate ClickOnce deployment into your solution, it's time to look at how to configure the installation settings to suit your own requirements. ClickOnce is so much a part of the Visual Basic Express development experience that it warrants three pages in the My Project settings form — general publishing settings, along with security and digital signing configuration options.

The main Publish tab is where the majority of the work is done (see Figure 14-7). You should first set the location for where the application is to be published. You'll find that the default setting sends it to a local website URL that includes the project's name. The ellipsis button enables you to change this location by browsing through the local file system (including any network drives or folders you're connected to) or the local web server.
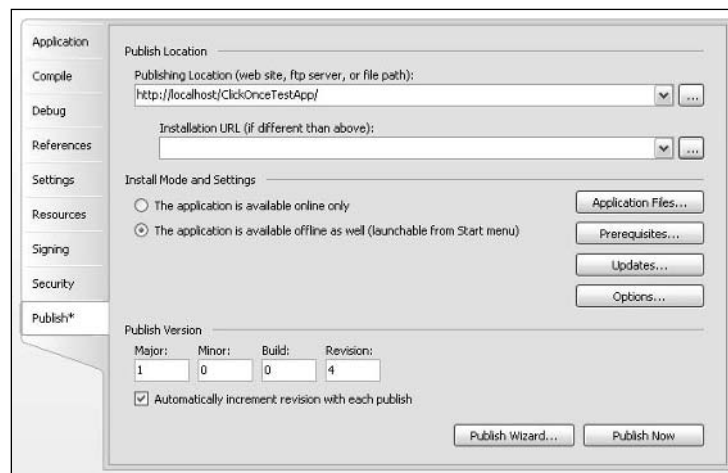
Figure 14-7

The other two options you can choose from are a remote FTP site and a remote website. The FTP option requires that you specify the FTP address and the settings needed to log onto the FTP server. Publishing directly to a remote website is possible only if the website has FrontPage Extensions installed, so if your site doesn't have FrontPage, you need to create the installation locally and then copy it using some other mechanism.

If you do choose to publish it locally, but intend for it to be then copied to another location — for example, on a remote website — you should then specify the Installation URL. This is used by the installation process to verify files and configuration options, so you need to include this if you are not going to be installing from the original publish location.

By default, your application is made available offline as well as online. This is the normal behavior for a Windows application because it enables the user to run the application without being connected to the Internet, but if you require total control over the version of software your users are running, then setting the application to be online only tells the deployment solution not to copy any of the application files to the local machine and instead to retrieve them as needed from the published location.

Visual Basic Express does a pretty good job of analyzing what files are required for a successful deployment, and you can double-check the file list by clicking the Application Files button. Each file defined in the application will be listed. Some project files may be hidden in the list if Visual Basic Express decided that they're not required, but you can check the Show All Files checkbox to display them.

The Application files dialog also enables you to include any files that are not part of the core application executable and define different download installation groups for them. This would enable your users to optionally install these additional components if they want them.

The Prerequisites dialog gives you the capability to control how system prerequisites are installed for your application (see Figure 14-8). As noted previously, all Visual Basic Express applications require the .NET Framework 2.0 to be installed on the computer first, so the prerequisite for that component is checked by default, but other components such as SQL Server Express are included only if you need them.

Once you've selected the components you want to include as part of your deployment process, you need to indicate the source from which users should retrieve the component installation packages. The default option is to use the component vendor's website — which in this case is Microsoft itself. Leaving this option selected means that if the user installs your application and the deployment determines that .NET Framework 2.0 (and any other marked prerequisites) is not installed, it downloads it from Microsoft's website.
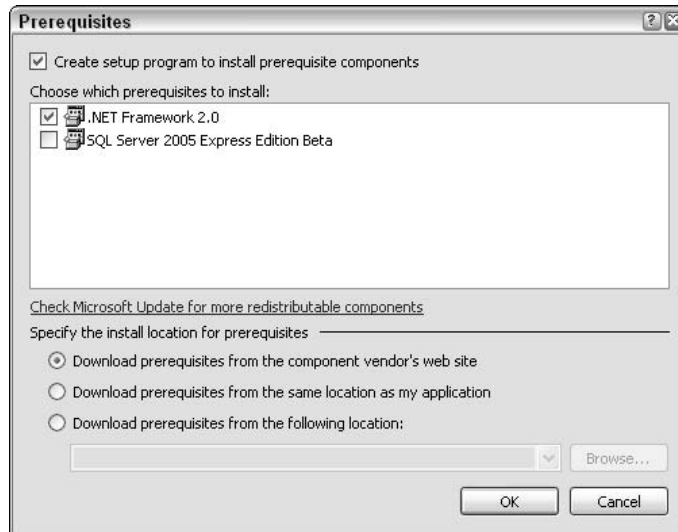
Figure 14-8

If you prefer, you can choose to include the setup packages for the prerequisites in your own deployment solution, or you can enter a different location where the installation can find the files.

You saw the Updates page in the previous Try It Out (refer to Figure 14-2), but the details weren't explained at that point. Previously, including the capability to automatically update your application once a user installed it on his or her system was a time-consuming and often costly process that included

subscription fees with specialized companies. These organizations (such as InstallShield) monitored your applications and, whenever an end user checked for updates, handled the updating process for you.

With Visual Basic Express, taking care of the update process is a matter of a couple of clicks to indicate that you are going to be doing updates and how the application should handle them. The obvious first option is to indicate that the application should check for updates. Without this checked, once the program is installed, it continues to run without checking for any changes that might have been made since the deployment.

If you need to ensure that the program is always run with the latest updates, select the "Before the application starts" option for update checking. Whenever the user runs the application, it checks the publish or update location for any updates made. If it finds an update, it is applied before the user can run the application. As you saw in the previous Try It Out, if the installation is available in offline mode, the user can choose to skip the update process.

Alternatively, the application can always start up with its current set of files and then check for updates once the application is running. This allows the update process to be performed in the background so it doesn't affect the startup sequence for the program. If updates were found, they are applied automatically the next time the user starts the application. You can control how often the update checking should be performed, from every time the application runs to a specified number of hours, days, or weeks.

If you have changed the application significantly, old versions might not be able to be updated automatically. Or you might decide that the old version should be left unchanged and only people with more recent builds installed are entitled to the latest update. You can specify a minimum required version for the application so that only more recent builds can find and accept this update, whereas old versions continue to run without the changes being installed.

The last set of options in the main Publish section of My Project deals with the installation itself (Figure 14-9). You can specify an installation language if it's different from the default that Visual Basic Express is using, along with the publisher's name (that's you!), and the product name. The product name setting is handy if you've used an unusual name for your project but want the program to be known as something else.

At this point, you can also specify a URL for users to go to for product support and the name of the web page that is built as part of a web deployment setup. Because this page is HTML, and you most likely will have modified it after the initial publishing process so it fits in with the style of your website, including additional links or information, you don't want the file to be generated every time the publish process takes place. You can disable this file generation by unchecking "Automatically generate deployment web page after every publish."

The other options found in this page can usually be left with their default values. If you don't want the application to automatically start after a successful installation, you can remove the check. CD installations can include the autorun.inf file, to automatically start the setup process when they're inserted into a CD drive; and when files are copied to a remote web server, you can tell Visual Basic Express to verify that the copy process was successful.
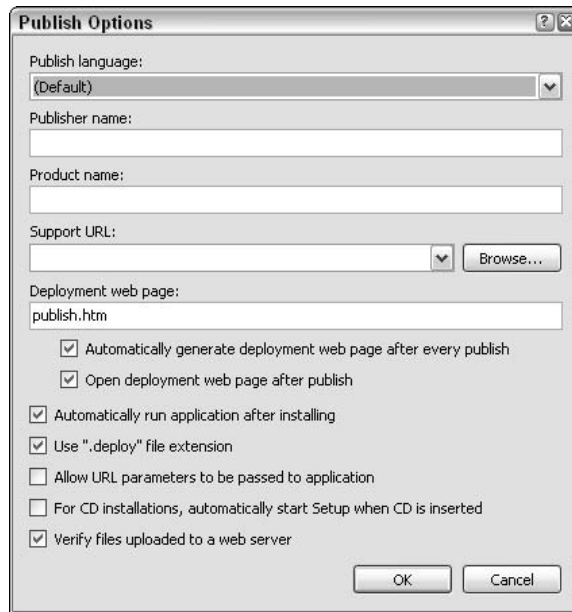
Figure 14-9

ClickOnce Has Security and Signing, Too

While all of these settings are enough for most application installations, you might find that you need additional options to enable your application to run correctly, and that's where the Security and Signing pages of My Project come into play.

When your application runs, it can perform only actions that it has been allowed to perform. If the program is installed locally on the normal file system, this means it can do pretty much anything; but if it's running over a network or from a website, it won't have access to many parts of the operating system.

The Security page (shown in Figure 14-10) allows you to enable ClickOnce Security options and specify how much security access the application needs to run. By default, ClickOnce security is not enabled, which means you must have full rights to run and install the application. Check the Enable ClickOnce Security Settings checkbox to gain access to the other settings.

You can specify that the application is a full trust program. This means the user must have installed it using administrator privileges and that it is running in a local context that allows it full access to the operating system.

However, if your program doesn't need access to everything, you can mark it as a partial trust application and then choose the permissions that you require. You should first choose the security zone from which the program is installed. By default, Visual Basic Express enables you to select Local Intranet (your normal home or office network), Internet (for website deployments), and Custom (which starts out with a blank slate of no permissions).

You should then scroll through the permission list and mark each one you require for inclusion if it differs from the Zone defaults. You can also exclude unnecessary permissions that belong to the selected zone.
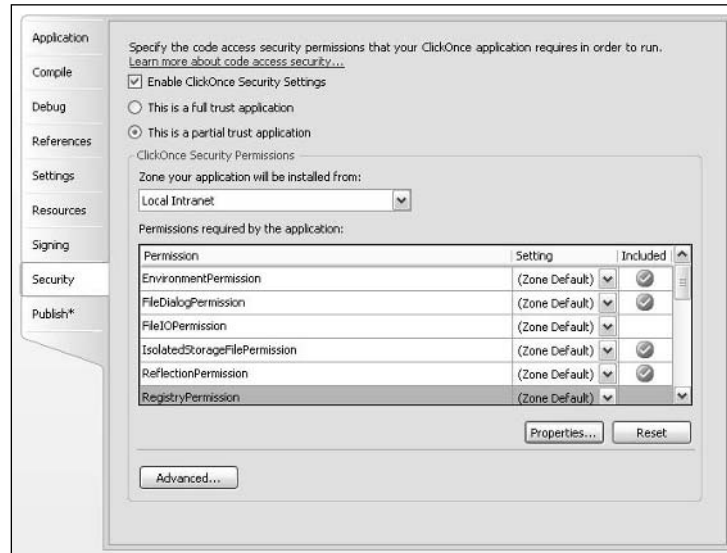
Figure 14-10

Every permission set has additional properties as well, enabling you to fine-tune exactly what your application needs to be able to do when it executes. For example, the `FileDialogPermission` set can be filtered so that only open or save dialogs can be shown, while the `SqlClientPermission` set can be customized to allow access only to SQL Servers using ADO.NET, and even to restrict access to applications that use blank passwords.

Using a digital signature, you can enable your application to be successfully deployed over the Internet without it being blocked as being unsecure. Visual Basic Express enables you to create temporary local digital signatures directly from the Signing page of My Project (see Figure 14-11).

If you have a real digital certificate, you can select it from the Certificate Store on your computer or from a physical file. Once you have selected the certificate you want to use, you can click the More Details button and get a window similar to what users see when they are examining the certificate upon download.

If you sign the assembly itself, you can protect it from hacking attempts, and Visual Basic Express can generate the strong name key file for you if you don't already have one. Whether you use the strong name in the certificate or create a new one, you can also password-protect the key file as well as add additional security to the signing process.

The default certificate Visual Basic Express creates for your application is not password protected, so this is an important consideration when you're creating your deployment solution.

In this last Try It Out, you will create the deployment project for the Personal Organizer application you've been building over the course of the book. You'll set the update options and select prerequisites and other settings so that the application can be successfully installed on another computer.
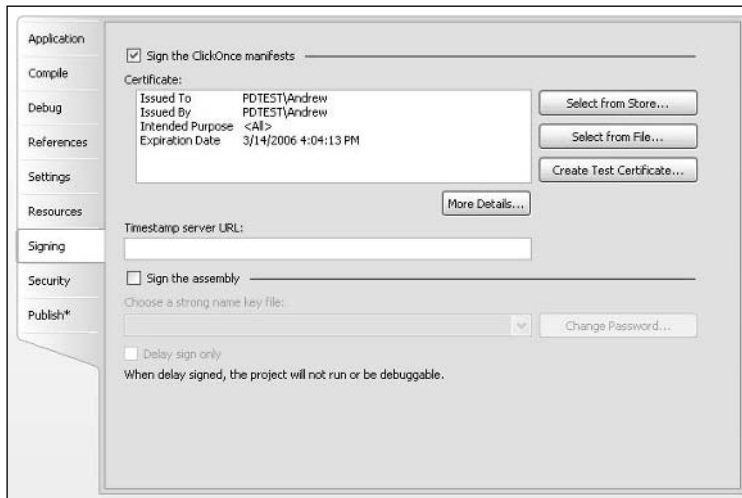
Figure 14-11

Try It Out   Advanced Settings in ClickOnce

1. Start Visual Basic Express and return to the Personal Organizer application that you've been working on. If you haven't completed all the exercises up to this point, you'll find an up-to-date project in the `Code\Chapter 14\Personal Organizer Start` folder, which you can use as a launching point for this Try It Out.

2. You can leave most of the settings for publishing to their default values, particularly in the Signing and Security pages, but you'll want to specify a couple of configuration options in the main Publish page. Open the My Project form and select the Publish page. Make sure the application can be run in offline mode so your users don't have to be connected to the installation server in order to be able to run the application.

    In addition, make sure the automatic increment of the publish version is checked so that each subsequent build of the deployment solution is identified with a new version number.

3. Click the Prerequisites button. When you connected the database to the project back in Chapter 7, you had the option to include it as a local file in the project. If you selected this option, you will find that the SQL Server 2005 Express prerequisite is already checked; otherwise, you need to select it to tell ClickOnce to include that requirement. Once you've verified that the prerequisites are selected, click OK to return to the main Publish options.

    A side effect of including the database as a local file is that it will also be installed and listed in the Application Files list. If your intention is that the database reside in a central location and the different installations all point to that file, then you can exclude it from the installation list.

4. Click the Updates button and ensure that the application checks for updates (and that it does so before every execution). This way, users always have the option to download and install the latest version if you update the product later.

5. The last thing to do is to set the text that will appear in the installation page. Click the Options button and change the publisher to Wrox's Starter Kit and the product name to My Personal Organizer. Leave all the other settings as is and click OK to save the changes.

6. Publish the project by selecting Build ⇨ Publish Personal Organizer. Because you've already selected all of the options you want, you don't have to go through all the steps of the wizard. Just click Finish to start the publish process.

7. Once the publishing has been completed, you are presented with a web page that should look like the one shown in Figure 14-12. Run the application via either the Install button or the Launch hyperlink (because you already have both .NET Framework 2.0 and SQL Server 2005 Express installed); and after ClickOnce has verified that you have all of the correct files, you should be presented with the familiar splash screen and login form you've been working with all along.

8. Congratulations! You've successfully deployed the Personal Organizer application.
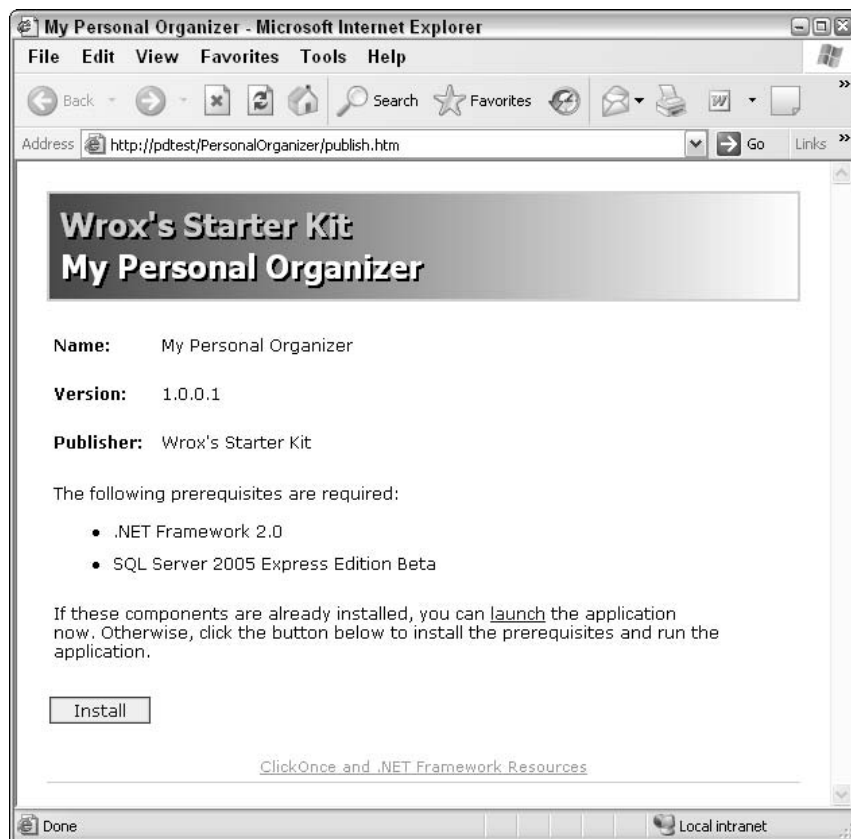
Figure 14-12

Summary

ClickOnce deployment makes the process of getting your program into the hands of your users incredibly easy. Without having to think about it, your project can be built into an installer, copied to a web location, and automatically publish updates so that end users always have the latest version, whether they're on a local PC or working on the program over the web.

In this chapter, you learned to do the following:

❑   Create an installer for your project to give it to other people

❑   Use the web to install and update your applications

❑   Create accompanying web pages so users know how to install your application

You've made it to the end of the book. By now you're familiar with the way Visual Basic Express works. You should have a solid understanding of the language, the user interface design mechanics and components, and the process of how to create and use databases. You also now know how to secure your application and get it out to your users.

Along the way, Visual Basic Express makes it easy for you at every step with wizards, aids, and help. You're now set to go ahead and make your own applications, easily but still including advanced techniques that professional developers will envy. Congratulations.

Exercise

1.   Update the Personal Organizer application to verify that updates work through the ClickOnce publishing process.

A

Need More? What's on the CD and Website

This book contains all of the information you need to get started with Visual Basic Express. From beginning to end, you can walk through the creation of a full-blown application that uses everything from simple text boxes and buttons to database connectivity, XML processing, web access, and more.

But having the information and theory isn't enough — you need Visual Basic Express itself if you want to put into practice any of the techniques you've learned throughout the pages of this book. Fortunately, Visual Basic Express is bundled with the book on the accompanying CD, along with SQL Server Express and a number of other development tools that might come in handy as you create your own applications. Here's a quick overview of the main applications you'll find on the CD:

❑ **Visual Basic 2005 Express Edition** — The main topic of this book, Visual Basic Express is a complete development environment that uses Visual Basic as the underlying language and couples it with the latest Integrated Development Environment produced by Microsoft. It is powered by the .NET Framework 2.0, which means it is completely up-to-date, with all of the new additions and enhancements made for programmers.

❑ **SQL Server Express** — SQL Server Express is a free version of SQL Server 2005 and can be installed on single PCs for database programming. It offers all of the performance of its bigger brother without the complexity of the enterprise features that SQL Server 2005 boasts.

❑ **MSDN Library** — An essential tool, the MSDN Library contains all of the documentation for Visual Basic Express. This includes a large section of "How Do I . . . ?" questions that answer commonly asked queries by walking you through examples in much the same way as this book does with its Try It Out sections. If you like this book's style, you'll feel immediately at home with this section of MSDN.

In addition, the MSDN Library comes with complete notes on the .NET Framework 2.0 and all of its classes and members.

❑ **Visual Web Developer 2005 Express Edition** — To do any kind of web development, you need Web Developer Express. This tool enables you to use the same Visual Basic code you've learned to use in this book to support applications that can run over the Internet.

There's more, too. The CD also contains other Microsoft-supplied utilities and development tools if you're still hungry for more information and aids.

On the Web, Too

In addition to the resources found on the CD, you'll find the complete code listings for all of the sample projects and exercises found in this book (assembled together into a downloadable package) at `www.wrox.com`.

All of the code is broken down into subfolders for each chapter, and then subfolders within the chapters for each exercise and Try It Out programming project. For the larger projects, you'll find multiple starting points so it's easier for you to get started on the exact project you're looking for. Whenever a chapter references the Personal Organizer application that is used as a basis for most of the book, you'll find at least two versions of the associated project — a starting version for which none of the chapter's code has been implemented and a complete version containing everything covered in that chapter.

Because the accompanying SQL Server database also grows as you progress through each chapter, you'll find an instance of it within each chapter's folder in a subfolder named Personal Organizer Database. In addition, every exercise has a separate project so you can examine the solution in detail.

Go to `www.wrox.com` and locate the page for this book by searching for the title or author name or go through the category listing. When you display the details page for *Wrox's Visual Basic 2005 Express Edition Starter Kit* by Andrew Parsons, you'll find a link labeled Download Code.

This link will take you to the download page, where you will find links for getting the complete code that accompanies this book, with options for HTTP and FTP downloads.

B

.NET — The Foundation

Visual Basic Express uses a technology known as .NET to give it the power and flexibility it exhibits during the development process. If you're unfamiliar with .NET in general and the .NET Framework in particular, this appendix should serve to introduce you to the main concepts of the technology.

The best place to start with Visual Basic Express is to actually examine just what Microsoft has done in the development arena and what all of the excitement concerning .NET is about. Understanding these two basic concepts will help immensely in your understanding of the total package that is Visual Basic Express.

Microsoft Visual Studio

Microsoft first released their development tools quite a few years ago. For example, MS-BASIC was first released for DOS. One glaring problem with their initial few releases was the lack of integration. BASIC programmers wrote in BASIC and called other BASIC modules, C programmers kept to themselves, and so on.

When Windows was released, there was a recognizable need to provide several development tools in one package. Visual Studio was created to do just that. Initially, Visual Studio was more of a marketing term than anything else. Each language still had its own Integrated Development Environment (IDE) with benefits and disadvantages.

Some languages (such as Visual Basic) were even quite inferior in the way they compiled code, and limited access to the more powerful parts of the Windows operating system in such a way as to render them toylike.

As each iteration of Visual Studio was released, the languages got closer in terms of performance, functionality, and ease of development. The different tools were also growing closer together to provide a more cohesive whole, but even the last version before .NET, Visual Studio 6.0, had completely different IDEs for the two primary languages, Visual Basic and Visual C++.

Therefore, to recap, the goal of Visual Studio was to provide developers with a cohesive set of tools with a variety of languages so that programmers could use their preferred language while using a common environment and set of functionality. This would result in the different programs being able to interact more fluidly and with a lot less headache compared to previous offerings for developers.

One of the problems with providing a tight, flawless coupling of the different languages and their IDEs was that the languages didn't lend themselves to being merged in such a fashion. Visual Basic didn't support full object-oriented practices, Visual C++ had a myriad of ways of defining an element for the graphical user interface (GUI), and the list went on.

Obviously, to achieve a true Visual Studio, where all languages are supported equally, with the same IDE and same set of functionality, drastic measures would need to be taken. In stepped .NET.

Microsoft .NET

Microsoft listened to the users of their development tools and went back to the drawing board. Their goal was to design the next generation not only for developers but also for the underlying structure and services that would provide a comprehensive whole for all users, without requiring anyone to learn multiple methods of achieving solutions.

One overriding force in their design was the Internet, and so it's no surprise that web standards and protocols were used in the basic design for this next set of technologies, all fitting under the banner of .NET.

It doesn't stop there, however. The huge growth of the Internet fostered increasingly demanding needs for proper distributed computing. When the last set of technologies were created, they were really developed for single PCs, or PCs on a local area network that was tightly controlled by specialized technicians such as network administrators.

With the Internet playing an increasingly large part in our computing lives, programmers found themselves with a requirement to provide solutions for users who weren't under the control of network administrators; and in fact, in the last couple of years, there are now more computers "out of control," so to speak, than there are those that are "in control."

Because of this, the state of a user's computer system is very hard to determine and really drives home the requirement of having standard communications protocols coupled with heavy-duty distributed technologies in place. The standard protocols have been introduced with the popularization of the Internet, with HTTP being the most easily identified one.

However, more recent standards, namely Extensible Markup Language (XML) and Simple Object Access Protocol (SOAP), have provided the way for the latest tools by giving developers a standard way of defining data and a standard method for transferring that data. This enables us to develop true web applications as opposed to web "presentations" that are available through the use of HTML and other similar presentation-level technologies.

> As a brief aside, Microsoft is so committed to following standards in order to open their development platform as much as possible that they submitted the .NET Framework to the European Computer Manufacturers Association (ECMA) board and had it ratified as a standard in its own right. As a result, other platforms besides Windows, such as the Mono project for Unix, now have a version of the Framework available.

In addition to this move, Microsoft also saw the need to improve the underlying componentization of software. Before .NET, much of the integration of software components was done with a technology called COM (the Component Object Model). When it was first introduced, COM promised the world to programmers, and it almost delivered.

It was intended to provide a watertight interconnection model between the different software components installed in Windows. Unfortunately, it turned out to be cumbersome, and rather than being rock-solid, it could be quite fragile at times. Due to the nature of registering the individual components into the Windows Registry, changing even one simple function could render the component incompatible with any other programs or components that relied on it.

In addition to this, COM based its usage of components on reference counting and so would only remove objects from memory when they were no longer referenced. A state called *DLL-Hell* was quickly coined by COM programmers, which refers to two objects connecting to each other but not being used anymore. Windows would not release their memory usage because it assumed they were still being referenced, when in reality the program that was using them could have been terminated long ago.

Because of this fragility, and the huge source of memory leaks that was DLL-Hell, something major needed to be done to the underlying method of creating and using components and the cleaning up of said components. This requirement went into the mix that was becoming the .NET Framework.

Many other factors were considered, but one last one that is worth mentioning here is the requirement to provide more robust enterprise-level servers. Security, transactional management, pooling of resources, and threads were all previously very hard to implement and fraught with dangers at every turn. .NET promised to change all that.

The result of all this planning and development was the holistic approach that became .NET. As David Lazar, the Microsoft Group Product Manager for Developer Tools says, "You only get the chance to hit the reset button once every 10 years or so," and .NET has certainly reset everyone's expectations.

The Microsoft .NET platform is made up of four main parts on top of the operating system itself. The central component is the .NET Framework, which provides the component infrastructure, language integration, and more that you'll see in a moment. Alongside this is the suite of .NET Enterprise Servers such as SQL Server, BizTalk Server, and others that fully integrate with the new .NET way of doing things.

The third part to the .NET platform, which goes hand-in-hand with these two, are the various .NET "building-block" services that Microsoft provides to make things even easier in this new world of .NET. The first and currently foremost service is Microsoft Passport, a now ubiquitous model of user identification and authentication that can be used by any developer.

On top of all of this lies the developer tools — first of which was Visual Studio .NET, a comprehensive set of developer tools that harnesses the power and ease-of-use found within the new .NET Framework and partner components. Visual Basic Express sits alongside Visual Studio 2005 in the grand scheme of .NET. All of these major components together form Microsoft .NET (see Figure B-1).

Besides the developer tools component, the other major part to this whole .NET conglomerate is the .NET Framework mentioned previously. When Microsoft began the process of creating the .NET vision, at its core was a new (and obviously better) way of supporting components and integrating the various programming languages. Why should it be completely different to code something in Visual Basic as compared to doing the same thing in C++?

Coupled with these desires came a need for easier development for more recent technologies, better security, and one thing that's long been a bane for programmers: a much easier deployment (or installation) model. Along came the .NET Framework, a collection of system-defined classes and objects that promised to do all of this and more.
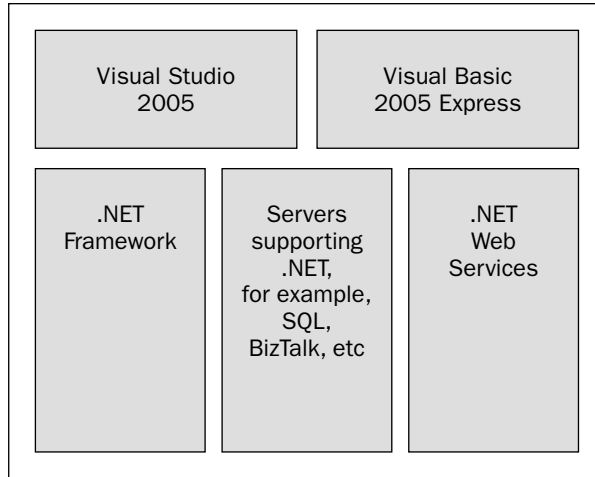
Figure B-1

Component Support

With the technology available before .NET came along, the only "proper" way of defining components was to use COM. However, as discussed previously, COM also introduced DLL-Hell, and a fragility that had almost every Windows developer cursing at least one time in their careers.

With the .NET Framework, it is now possible to produce components that can interrelate without these problems, and have side-by-side existence of multiple versions of a particular component. This in conjunction with garbage collection also means that DLL-Hell should be a thing of the past.

Language Integration

Until now, languages have been independent of each other. Create components in one, and you wouldn't be able to extend them in another. This, too, has been dealt with in .NET. All programs compile down to a common intermediate language, so it's irrelevant what language they were originally designed in. This common language is known as the MSIL — Microsoft Intermediate Language.

In fact, Microsoft provides disassemblers that anyone can use to look at the CIL code of a program. You can write the same program in two different languages and compare the CIL to prove that they do in fact end up being the same code.

This initially begs the question of why multiple languages should be supported, but it doesn't take too much analysis to understand. Sure, if you're just starting to develop, with .NET as your first environment, it wouldn't make much difference if you had support for one language or the more than 20 supported in Visual Studio 2005. Or if you're using the Express developer tools, you find little to distinguish between Visual Basic Express and Visual C# Express beyond personal preference.

However, if you're already a programmer, your livelihood depends on your existing knowledge. With the .NET Framework defined in such a way that it enables any language to be supported as long as it compiles down to CIL, your skills are still valuable.

This language integration goes further, including making common the types of variables that are usable across the languages — a 32-bit integer is now the same in Visual Basic as it is in C# or C++.

Coupled with this Common Type System (CTS) and the Common Intermediate Language (CIL) is the Common Language Runtime (CLR) and the Common Language Infrastructure (CLI). The CLR is kind of like the old Visual Basic Runtime (a closer analogy is probably the Java Virtual Machine). It is the beast that interprets and executes the CIL code generated by the .NET language compilers.

The CLI is a standardized part of the CLR (ratified by ECMA, the European standards body), aimed at providing a way for other platforms to provide their own CLRs and so have a truly cross-platform capability. As you can see, the .NET vision really aimed high — cross-platform and cross-language compatibilities are a dream for a lot of developers and to have both remotely possible at one go is mind-boggling.

Along with the CLR, the .NET Framework comes with a whole set of framework base classes (see Figure B-2). These classes define everything from Windows forms GUI objects and web controls to security, and everything in between. The .NET Framework aims to eliminate the need for directly using the Windows API by encapsulating everything in a framework base class and extending them to be more consistent with the way Microsoft wants you to code your solutions at the same time.
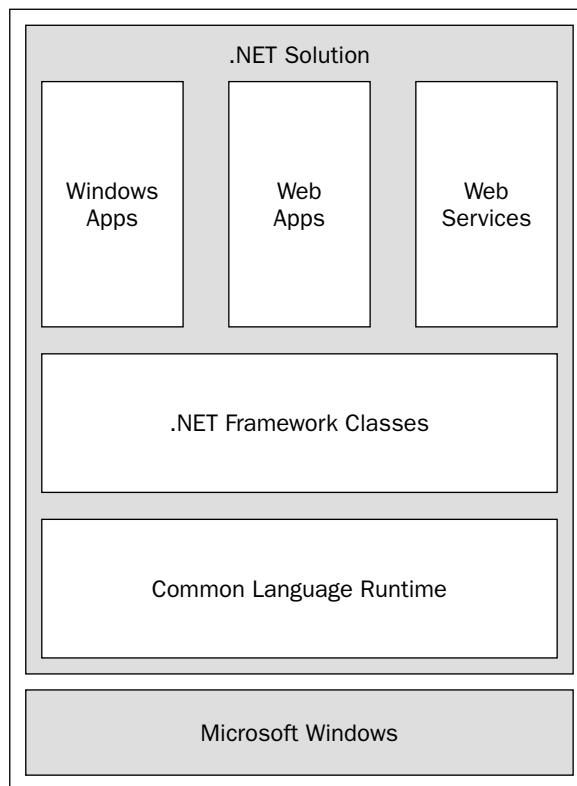
Figure B-2

Developer Tools

Once Microsoft had developed the underlying framework that would be common to all languages, they then turned to changing the languages themselves in subtle and (in the case of Visual Basic) some not so subtle ways.

In fact, there was some fairly major work done to provide languages that would work side-by-side without a problem and lend themselves to using the same IDE. Microsoft even went to the extreme of creating a new language, C# (pronounced "C-sharp"), which is now recognized as a fully standards-compliant language.

The Microsoft development team then went on to completely revolutionize the IDE, first for Visual Studio, and then for individual languages such as Visual Basic Express. They analyzed all the existing IDEs, from Visual Basic, InterDev, FoxPro, C++, and so on, and took the best parts of each as a basis. Added to the mix was an extra requirement for additional functionality that makes creating an application even easier for a developer.

As a result of all of this, Visual Basic Express provides a comprehensive toolset for the developer. It shares a common IDE with Visual Studio .NET, which ships with C#, Visual Basic, and Visual C++, but actually more than 20 languages can be "plugged" into the Visual Studio 2005 environment. Languages such as Perl, COBOL, RPG, and Java, and even less frequently used languages such as Eiffel, can all be integrated into the one IDE of Visual Studio 2005 and so can interoperate with Visual Basic Express.

The .NET Framework

The .NET Framework is a huge collection of classes complete with methods, properties, and events, like any of the classes you can create yourself. They serve many purposes, from being able to store information in collections or even simple data like string variables to encryption of data and encapsulation of web service methods.

The main part of the Framework is accessible through a central namespace called `System`. A secondary core namespace called `Microsoft` is used to expose Microsoft Windows–specific functionality, such as specific features of the Visual Basic language or Windows system calls and structures, but the discussion that follows deals with the `System` namespace.

Within `System` are many subordinate namespaces, which in turn have their own child namespaces, and so on, forming a great hierarchical tree that encapsulates all of the base functionality you need to create your Visual Basic Express applications. Each namespace has its own set of classes and structures, along with interfaces, delegates, and enumeration sets. For example, to convert an `Object` to a `DateTime` structure, you need the following members of `System`:

❑ **Convert** — A class used to convert a base data type to another data type

❑ **Object** — The generic class used as a base for all other class types

❑ **DateTime** — A structure that can contain an instant in time

❑ **DateTimeKind** — An enumeration that lists the way the date and time information can be stored in the structure

While the absolute fundamental classes and structures can be found in `System`, the majority of the .NET Framework can be found in the subordinate namespaces. You've actually used quite a few of these throughout the course of this book, in the Try It Outs and Exercises. Here are some of the more common second-tier namespaces that you might use in your programming:

❑ **System.Collections** — A series of classes that enable you to store arrays of like objects. The main `Collection` object is supported by specific classes that solve particular solutions, such as the `BitArray` collection that you could use in place of the bitwise operations in Chapter 7, and the `SortedList` that automatically sorts the entries you add by their key value.

❑ **System.Data** — You should already be familiar with this namespace as it contains all the classes necessary to process database information. `System.Data` has three main subcomponents: `System.Data.SqlClient` for SQL Server–based processing and `System.Data.OleDb` and `System.Data.Odbc` for other database types.

❑ **System.Drawing** — All graphical functionality can be sourced from this namespace, either directly when you use the `Graphics` object or indirectly when you set various properties on a user interface component.

❑ **System.IO** — `System.IO` contains the classes and methods needed to read and write all kinds of files. From flat files to memory streams, `System.IO` contains the functionality you require. It even has a subnamespace for the serial ports on a computer so you can write directly to the port.

❑ **System.Net** — You used `System.Net` in Chapter 11 to create and send e-mail to people in a list. `System.Net` also contains classes you can use to call out to a website and process the content of a web page (`System.Web.WebRequest` and `System.Web.WebResponse`) and to process information across a network.

❑ **System.Security** — In Chapter 13, you created additional functionality in your Personal Organizer application that encrypted the password string. This used the `System.Security.Cryptography` namespace, one of the subnamespaces of `System.Security`. `System.Security` also provides you with functionality to process security permission sets and authentication protocols.

❑ **System.Text** — Used to encode and decode from a variety of data formats, `System.Text` also contains the extremely valuable subnamespace `RegularExpressions`, which can be used to extract and find information in a string using regular expression technology.

❑ **System.Timers** — Relatively simple compared to the other namespaces, `System.Timers` nonetheless performs an important function by giving you the capability to create timers in your application for scheduled events.

❑ **System.Windows** — `System.Windows` is unusable by itself, but if you delve into its child namespace `System.Windows.Forms`, you can access all of the user interface controls and the form functionality. By default, your projects normally have a reference to `System.Windows.Forms` implicitly defined, so you won't see the full definition. However, if you ever use a line such as `Dim MyButton As Button`, you're referencing a class within this namespace.

❑ **System.Xml** — This namespace encapsulates all of the classes needed to process XML documents as discussed in Chapter 12. From reading and writing XML to processing individual nodes and their attributes, everything you need to deal with XML can be found here.

Most of these main namespaces are further subdivided. As an example, consider the `System.Drawing` namespace. It is used to do all kinds of graphical drawing, whether it is on a form or for printing purposes. Everything revolves around a base `Graphics` object, fully defined as `System.Drawing.Graphics`, but there are subordinate namespaces for specific actions.

`Drawing2D` enables you to control simple geometric shapes, `Imaging` provides a multitude of classes for advanced graphic techniques such as alpha blending or image encoding, `Printing` exposes the functionality you need to send information to a printer, and `Text` gives you the capability to "paint" text into a form or other graphics object without the need for a control to host the text.

As you can see, the .NET Framework forms an essential component of the Visual Basic Express development environment. Without the many classes and namespaces it provides, creating applications would be an incredibly difficult experience. If you're ever wondering how to do something, make sure you look through the .NET Framework classes first before you create your own—you might be pleasantly surprised.

C

Answers to Exercises

Chapter 1

Exercises

1. **Installing Visual Web Developer 2005 Express Edition:** To create applications that run on the Internet, you can still use Visual Basic 2005 as a language, but you will need to install Visual Web Developer 2005 Express Edition. The method for installing Web Developer Express is exactly the same as what has been outlined here, but it will install Web Developer instead of Visual Basic. If you have already installed Visual Basic Express, you'll find that the Web Developer installation process does not include options for MSDN or SQL Server, as it automatically detects that they are already present on your system.

2. **Customizing the Browser Application:** Extend your web browser program so you can both return to the previous web page you visited and navigate to the default home page of Internet Explorer. You'll need to use two more methods of the `WebBrowser` control — `GoHome` and `GoBack`.

Exercise 1 Solution

Installing Visual Web Developer 2005 Express Edition is performed in much the same way as Visual Basic 2005 Express. Locate the installation package on the CD that accompanies this book and start the `setup.exe` application.

If you have previously completed the installation of Visual Basic 2005 Express or any other product in the Visual Studio 2005 line, the installation automatically detects the presence of common components such as MSDN Library and SQL Server Express. Otherwise, these components are presented as optional components during the setup process.

Exercise 2 Solution

Add a new `Button` control to the form, next to `Button1`. Change the `Text` property to `Back`. Double-click on `Button2` and insert the following code:

```
    WebBrowser1.GoBack
```

Add another `Button` control to the form, next to the `Button` you inserted previously. Change the Text property to `Home`. Double-click on `Button3` and insert the following code:

```
    WebBrowser1.GoHome
```

Run the program again and browse to a website. Test your new buttons by navigating back pages and returning to the home page of the current user.

Chapter 2

Exercises

1. **Customize the DVD Collection application:** Re-open your `MyOrganizerMovies` project and change the images for the View DVDs and Search Online buttons. You'll need to set three properties for each in the Properties window — `NormalImage`, `HoverImage`, `PressedImage` — and you will need to edit the code so that the proper Resource objects are used.

2. Look up the documentation for an example of how to use the `BackgroundImage` property of a control.

Exercise 1 Solution

1. Open the `MainForm.vb` form in Design view. Select the View DVDs button and go to the Properties window.

2. Select the `NormalImage` property and click the ellipsis button to bring up the Select Resource dialog window. Choose DVD-normal and click OK.

3. Do the same for `HoverImage` and `PressedImage` but choose the DVD-hover and DVD-down resources, respectively.

4. Select the Search Online button and repeat steps 2 and 3.

5. Change to code view by right-clicking on the form and selecting View Code.

6. Find all occurrences of `My.Resources` and replace the current resource image with `DVD_normal`.

7. Build and run the application.

Exercise 2 Solution

1. Start the MSDN documentation by pressing F1.

2. Click the Search tab at the top of the window to show the Search dialog.

3. Type `backgroundimage` and click the Search button.

4. Select the `Control.BackGroundImage` search result (it's normally the first result) by clicking on the blue heading text.

5. Scroll down until you find the example in Visual Basic code.

Chapter 3

Exercise

1. Create a database that uses the Person and Pet tables defined at the beginning of this chapter. Make sure they are linked through a foreign key relationship so that each Pet record must be owned by a Person record.

Exercise 1 Solution

1. Start Visual Basic Express and create a new Windows Application project. Add a new SQL Database by selecting Project ➪ Add New Item, selecting SQL Database from the Add New Item dialog and clicking OK.

2. Add a new table via the Database Explorer and add the following columns:

Column Name	Data Type	Allow Nulls
ID	int	Unchecked
FirstName	nchar(35)	Unchecked
LastName	nchar(35)	Unchecked
DateofBirth	datetime	Checked
Address	nchar(255)	Checked
Notes	text	Checked

Set the ID column as the primary key by right-clicking and selecting Set Primary Key, and set the Is Identity property to Yes.

3. Save the table as Person. Add another table via the Database Explorer and add the following columns:

Column Name	Data Type	Allow Nulls
ID	int	Unchecked
PersonID	int	Unchecked
Name	nchar(20)	Unchecked
Type	nchar(50)	Checked
Breed	nchar(50)	Checked

Set the ID column as the primary key by right-clicking and selecting Set Primary Key, and set the Is Identity property to Yes.

4. Save this table as Pet. Click the Relationships button on the toolbar to bring up the Foreign Key Relationships window. Click Add to add a new foreign key and then click the ellipsis button next to the Tables and Columns Specification property.

Select the `Person` table for the Primary Key table and then select the ID column. In the Foreign Key table list, choose the `PersonID` column to bind the two together. Click OK to save the settings and then click Close to return to the table editing view.

5. Right-click on the `Person` table in the Database Explorer and select Show Table Data. Enter the details about a person and note the ID value that is assigned to it.

6. Right-click on the `Pet` table in the Database Explorer and select Show Table Data. Enter the details of a pet and ensure that the `PersonID` matches the value you jotted down in step 5. You will now be able to save the information. If you entered a value that was different from the ID automatically created in step 5, you will receive an error when you try to save the data.

Chapter 4

Exercises

1. **Anchor fields:** Set the `Anchor` properties on the Address and Notes `TextBox` controls so that they resize automatically when the form is resized.

2. **Adding the PersonList user control:** In the next chapter you'll need the `PersonList` user control to show the list of people in the database. Create a new user control with a `ListBox` and two `Button` controls. Remember to set the `Anchor` properties so that the fields are resized and positioned when the form's dimensions are changed (see Figure C-1).
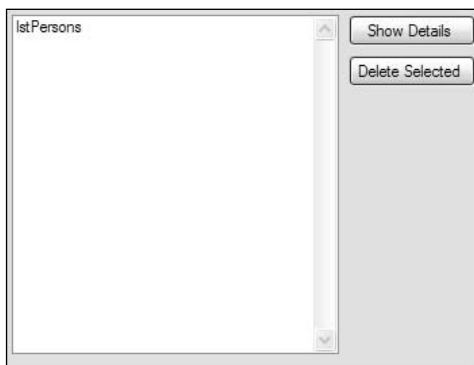
Figure C-1

Exercise 1 Solution

1. Select the Address `TextBox` and change its `Anchor` property to `Top`, `Left`, `Right`. This will ensure that the Address area is resized to the right as the form gets wider.

2. Select the Notes `TextBox` and change the `Anchor` property to `Top`, `Bottom`, `Left`, `Right`. Doing this will ensure that the Notes area takes up the remaining area of the control regardless of how big it is.

Exercise 2 Solution

1. Add a new user control by running the Project ➪ Add User Control command. Name the control `PersonList.vb` and click OK.

2. Add a `ListBox` control to the form and set the following properties:

- ❑ **Name** — `lstPersons`
- ❑ **ScrollAlwaysVisible** — `True`
- ❑ **SelectionMode** — `MultiSimple`
- ❑ **Anchor** — `Top, Bottom, Left Right`

3. Add two `Button` controls to the form and change their `Anchor` properties to `Top, Right` so they will always be aligned to the right-hand side of the control. Set the following properties:

- ❑ **Button #1 Name** — `btnShowDetails`
- ❑ **Button #1 Text** — `Show Details`
- ❑ **Button #2 Name** — `btnDeleteSelected`
- ❑ **Button #2 Text** — `Delete Selected`

4. Save the project.

Chapter 5

Exercises

1. Create an application that changes the color of the text in a `TextBox` control if numbers are present. To do this, you'll need to write a subroutine to handle the `TextChanged` event of a `TextBox` and set the `ForeColor` property if the condition is met.

2. Create an application that counts from 1 to 100 in increments specified by the user and displays the values in a `TextBox`.

3. Modify the application you created in Exercise 2 so that it ensures the increment is a number before it performs the loop.

Hint: Use the `IsNumeric` *built-in function to determine if a variable is numeric or not.*

Exercise 1 Solution

1. Create a new Windows Application project and add a `TextBox` to the form.

2. Double-click the `TextBox` to have a subroutine automatically generated for the `TextChanged` event, and add the following code:

```
Private Sub TextBox1_TextChanged(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles TextBox1.TextChanged
        If TextBox1.Text.Contains("1") Then
            TextBox1.ForeColor = Color.Red
        ElseIf TextBox1.Text.Contains("2") Then
            TextBox1.ForeColor = Color.Blue
        ElseIf TextBox1.Text.Contains("3") Then
            TextBox1.ForeColor = Color.Green
        ElseIf TextBox1.Text.Contains("4") Then
            TextBox1.ForeColor = Color.Yellow
        ElseIf TextBox1.Text.Contains("5") Then
            TextBox1.ForeColor = Color.Khaki
```

```
            ElseIf TextBox1.Text.Contains("6") Then
                TextBox1.ForeColor = Color.DarkGreen
            ElseIf TextBox1.Text.Contains("7") Then
                TextBox1.ForeColor = Color.Chocolate
            ElseIf TextBox1.Text.Contains("8") Then
                TextBox1.ForeColor = Color.Crimson
            ElseIf TextBox1.Text.Contains("9") Then
                TextBox1.ForeColor = Color.DarkGoldenrod
            ElseIf TextBox1.Text.Contains("0") Then
                TextBox1.ForeColor = Color.HotPink
            Else
                TextBox1.ForeColor = Color.Black
            End If
    End Sub
```

3. Run the application and enter a mix of characters and numbers and observe the color of the text change as it detects the numbers.

Exercise 2 Solution

1. Create a new Windows Application project. Add two `TextBox` controls and one button to the form. Set the following properties:

- ❑ **Button Name** — `btnGo`
- ❑ **Button Text** — `Go`
- ❑ **TextBox #1 Name** — `txtIncrement`
- ❑ **TextBox #2 Name** — `txtResults`
- ❑ **TextBox #2 MultiLine** — `True`
- ❑ **TextBox #2 Scrollbars** — `Vertical`

2. Double-click the button to generate code for the `Click` event and enter the following code:

```
        txtResults.Text = vbNullString

        For Counter As Integer = 1 To 100 Step CType(txtIncrement.Text, Integer)
            txtResults.Text &= vbCrLf & Counter.ToString
        Next
```

The first line resets the second `TextBox` to contain an empty string. Then a loop is defined that counts from 1 to 100 in increments of the value found in `txtIncrement`. Note that it's converting the value found in the `Text` property to an `Integer` so the compiler knows that the code is intentional.

The line inside the `For` loop concatenates each subsequent value to the existing contents of the `Text` property. The `&=` assignment operator is a shorthand form of saying `Variable1 = Variable1 & Variable2`. `vbCrLf` is a special constant that puts the following text on a new line.

Exercise 3 Solution

1. Return to the code for the `Click` event and add a condition to use the `IsNumeric` function so that the routine now looks like this:

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

    If IsNumeric(txtIncrement.Text) Then
        txtResults.Text = vbNullString

        For Counter As Integer = 1 To 100 Step CType(txtIncrement.Text, Integer)
            txtResults.Text &= vbCrLf & Counter.ToString
        Next
    Else
        MessageBox.Show("Sorry, the increment you entered is not valid.")
    End If

End Sub
```

2. Run the application, enter alphabetic characters in the Increment text box, and click the Go button.

Chapter 6

Exercises

1. Create an event handler for the New Person menu item that replicates the code you created for the New button on the `ToolStrip`.

2. Create an event in the `PersonalDetails` control that you can raise when the Save and Cancel buttons are clicked.

Exercise 1 Solution

1. Because the event signatures are the same for the `ToolStrip` New button and the New Person `MenuItem`, you can just change the `Handles` clause of the existing subroutine:

```
Private Sub newToolStripButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles newToolStripButton.Click, _
    newToolStripMenuItem.Click

    If objPersonalDetails IsNot Nothing Then
        objPersonalDetails.ResetFields()
        Me.Text = "Personal Organizer"
    End If
End Sub
```

Exercise 2 Solution

1. Define the event at the top of the `PersonalDetails.vb` code:

```
Public Event ButtonClicked(ByVal iButtonType As Integer)
```

2. Replace the `MessageBox` lines in the `ButtonClickHandler` routine to raise the event instead, including an identifier that tells the event handler routine which button was clicked:

```
Private Sub ButtonClickedHandler(ByVal sender As System.Object, _
    ByVal e As System.EventArgs)
```

```
                  Dim btnSender As Button = CType(sender, Button)
                  If btnSender.Name = "btnSave" Then
                      RaiseEvent ButtonClicked(1)
                  ElseIf btnSender.Name = "btnCancel" Then
                      RaiseEvent ButtonClicked(2)
                  End If
          End Sub
```

3. Change the definition of `objPersonalDetails` in the main form to include the `WithEvents` keyword:

```
      Private WithEvents objPersonalDetails As PersonalDetails
```

4. Create an event handler to intercept the event you created:

```
  Private Sub objPersonalDetails_ButtonClicked(ByVal iButtonType As Integer) _
      Handles objPersonalDetails.ButtonClicked
      MessageBox.Show("A button was clicked: " + iButtonType.ToString)
  End Sub
```

5. Run the application and click the Save and Cancel buttons to test the process.

Chapter 7

Exercise

1. Add four more routines to the `GeneralFunctions.vb` module to perform the following functions:

  a. Determine whether a specified user exists.

  b. Determine whether a user's password matches a given string.

  c. Create a new user record.

  d. Update a user record's Last Logged In value.

These functions are needed for the next chapter, so make sure you do them all!

Exercise 1 Solution

1. To determine whether a user exists, first retrieve the `POUser` table and then apply a `RowFilter` to a `DataView` copy of the table. If the `RowFilter` returns a row for the specified `UserName`, then return a `True` value to let the calling application know that it was found:

```
  Public Function UserExists(ByVal UserName As String) As Boolean
      Dim CheckUserAdapter As New _PO_DataDataSetTableAdapters.POUserTableAdapter
      Dim CheckUserTable As New _PO_DataDataSet.POUserDataTable

      CheckUserAdapter.Fill(CheckUserTable)
      Dim CheckUserDataView As DataView = CheckUserTable.DefaultView
      CheckUserDataView.RowFilter = "Name = '" + UserName + "'"

      With CheckUserDataView
          If .Count > 0 Then
```

```
                Return True
        Else
                Return False
        End If
    End With
End Function
```

2. This is a variation on the previous function, but this time it first finds the row in the `POUser` table and then, when found, compares the `Password` fields. If they match, it returns `True`; in all other cases, it returns `False`:

```
Public Function UserPasswordMatches(ByVal UserName As String, ByVal Password As
String) As Boolean
    Dim CheckUserAdapter As New _PO_DataDataSetTableAdapters.POUserTableAdapter
    Dim CheckUserTable As New _PO_DataDataSet.POUserDataTable

    CheckUserAdapter.Fill(CheckUserTable)

    Dim CheckUserDataView As DataView = CheckUserTable.DefaultView
    CheckUserDataView.RowFilter = "Name = '" + UserName + "'"
    With CheckUserDataView
        If .Count > 0 Then
            If .Table.Rows(0).Item("Password").ToString.Trim = Password Then
                Return True
            Else
                Return False
            End If
        Else
            Return False
        End If
    End With

End Function
```

3. Creating a new `POUser` record is straightforward because it does not require any foreign keys to be set up:

```
Public Function CreateUser(ByVal UserName As String, ByVal Password As String) As
Boolean
    If UserExists(UserName) Then Return False

    Dim CreateUserAdapter As New _PO_DataDataSetTableAdapters.POUserTableAdapter
    Dim CreateUserTable As New _PO_DataDataSet.POUserDataTable

    CreateUserAdapter.Fill(CreateUserTable)

    CreateUserTable.AddPOUserRow(UserName, UserName, Password, Now, Now, 0)
    CreateUserAdapter.Update(CreateUserTable)

End Function
```

4. Find the specified user first; then, in the `DataView`, you can edit the `DateLastLogin` field directly and then update the database through the `DataAdapter`:

```
Public Sub UpdateLastLogin(ByVal UserName As String)
    Dim UpdateUserAdapter As New _PO_DataDataSetTableAdapters.POUserTableAdapter
    Dim UpdateUserTable As New _PO_DataDataSet.POUserDataTable

    UpdateUserAdapter.Fill(UpdateUserTable)
    Dim UpdateUserDataView As DataView = UpdateUserTable.DefaultView
    UpdateUserDataView.RowFilter = "Name = '" + UserName + "'"
    With UpdateUserDataView
        If .Count > 0 Then
            .Table.Rows(0).Item("DateLastLogin") = Now
        End If
    End With
    UpdateUserAdapter.Update(UpdateUserTable)

End Sub
```

Chapter 8

Exercises

1. Use the code snippet library to draw a pie chart on a form. The pie chart snippet can be found by selecting Creating Windows Forms Applications ⇨ Drawing.

2. Create a class from two partial classes whereby one defines two variables and the other combines them together.

Exercise 1 Solution

1. Create a new Windows Forms application and add a button to the form.

2. In code view, right-click in the class and select the Insert Snippet command. Choose the Creating Windows Forms ⇨ Drawing category and then select Draw a Pie Chart to insert the routine. It requires a number of parameters that you will need to define variables for before calling it. Fortunately, the snippet command also includes an additional function called DrawPieChartHelper that provides a basis for these required values.

3. Add a Click event handler routine for the button and add the following code:

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

4. Modify the values in the DrawPieChartHelper so that you can display it in the normal form's size:

```
Public Sub DrawPieChartHelper()
    Dim percents() As Integer = {10, 20, 70}
    Dim colors() As Color = {Color.Red, Color.CadetBlue, Color.Khaki}
    Dim graphics As Graphics = Me.CreateGraphics
    Dim location As Point = New Point(70, 70)
    Dim size As Size = New Size(200, 200)
    DrawPieChart(percents, colors, graphics, location, size)
End Sub
```

5. Run the application and click the button to have a pie chart drawn on the form, as shown in Figure C-2.
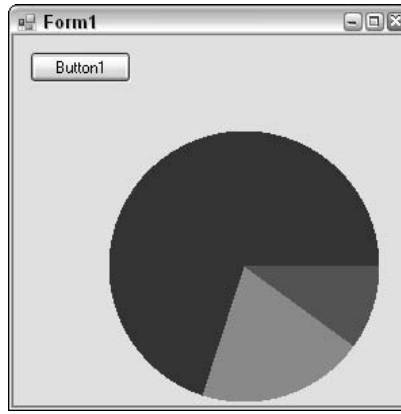
Figure C-2

Exercise 2 Solution

1. Create a new Windows Forms application and add a button to the form.

2. Add two class files to the application using Project ➪ Add Class. In the first class file, change the class name to `MyTest`, mark it as `Partial`, and insert the following code:

```
Partial Public Class MyTest
    Private mFirstNumber As Integer
    Private mSecondNumber As Integer

    Public Property FirstNumber() As Integer
        Get
            Return mFirstNumber
        End Get
        Set(ByVal value As Integer)
            mFirstNumber = value
        End Set
    End Property
    Public Property SecondNumber() As Integer
        Get
            Return mSecondNumber
        End Get
        Set(ByVal value As Integer)
            mSecondNumber = value
        End Set
    End Property
End Class
```

3. In the second class, change its name to `MyTest`, too, mark it as `Partial`, and insert the following code:

```
Partial Public Class MyTest
    Public Function AddNumbers() As Integer
        Return (mFirstNumber + mSecondNumber)
    End Function
End Class
```

4.   In the button's `Click` event handler, add the following code and run the application:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    Dim MyObject As New MyTest
    With MyObject
        .FirstNumber = 10
        .SecondNumber = 23
        MessageBox.Show(.AddNumbers.ToString)
    End With
End Sub
```

Chapter 9

Exercise

1.   In the Try It Out that added the Amazon web service to your Personal Organizer application, the `PersonalDetails` control can save the search results only when the `GetGiftIdea` form is closed. Change the program so that the `GetGiftIdea` form raises an event when the Save button is clicked, which the `PersonalDetails` control should handle and add the message to the Notes field. The Save button should also not close the `GetGiftIdea` form, so the user can perform multiple searches.

Exercise 1 Solution

1.   Because you will need to receive events from the `GetGiftIdea` form, you will need to change the definition of the `frmGetGiftIdeas` object so that it is accessible throughout the form's code. This means that you will need to declare it as a module-level variable. The `WithEvents` keyword is used to identify the object as one that can raise events that you wish to intercept:

```
Private WithEvents frmGetGiftIdeas As GetGiftIdeas
```

Remember to also change the Get Gift Ideas button's `Click` event so that the object is instantiated:

```
frmGetGiftIdeas = New GetGiftIdeas
```

2.   Open the `GetGiftIdea.vb` file in code view and modify the Save button's `Click` event handler as follows:

```
Private Sub btnSave_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnSave.Click
    Dim sGiftIdeasList As String = "Suggested gift ideas: "

    For iCounter As Integer = 0 To clbResults.CheckedItems.Count - 1
        If iCounter > 0 Then sGiftIdeasList += ", "
        sGiftIdeasList += clbResults.CheckedItems(iCounter).ToString
```

```
        Next

        RaiseEvent GiftIdeasSaveRequest(sGiftIdeasList)
    End Sub
```

3. Define the event at the top of the form's code:

```
    Public Event GiftIdeasSaveRequest(ByVal GiftIdeasList As String)
```

4. Return to the `PersonalDetails` control and add a routine to handle the
`GiftIdeasSaveRequest` event that adds the text included in the event to the Notes field:

```
Private Sub frmGetGiftIdeas_GiftIdeasSaveRequest(ByVal GiftIdeasList As String) _
    Handles frmGetGiftIdeas.GiftIdeasSaveRequest
    txtNotes.Text += GiftIdeasList
End Sub
```

5. Run the application to confirm that you can add multiple sets of search results to the Notes field
without closing the form.

Chapter 10

Exercise

1. Open the Personal Organizer project you worked on in Chapter 9 and debug through the call to
the Amazon web service. Try to determine how many items are returned from the call by looking
at the `ItemSearchResponse` object in the Quick Watch window before the `CheckedListBox` is
populated.

Exercise 1 Solution

1. Place a breakpoint on the first line of the event handler routine for the Search button's `Click`
event and run the application.

2. Step through the code by using either Step Into (F8) or Step Over (Shift+F8) actions until you
reach the `With awsItemSearchResponse` line.

3. Right-click `awsItemSearchResponse` and select Quick Watch. Expand the `Items` property to
examine the number of items returned.

Chapter 11

Exercises

1. Customize the printing code so that it prints the list of people only if the `Person List` control is
showing. Add another report to display information about the currently selected person if indi-
vidual details are shown.

2. Add two elements to the `StatusStrip` at the bottom of the `PersonalOrganizer`'s main form,
a `StatusLabel` and a `ProgressBar`. Keep the `StatusLabel` up to date with the number of
people currently in the database for the current user and use the progress bar to indicate how
much of the report has been generated when it is processing the person list.

Exercise 1 Solution

1. Create an additional `GenerateReport` function in the `GeneralFunctions.vb` module. This time, you need to include the ID of the person the user has selected. Use the following code for the routine:

```
Public Function GenerateReport(ByVal PersonID As Integer, _
    ByVal PersonID As Interger) As String
    Dim GetPersonAdapter As New _PO_DataDataSetTableAdapters.PersonTableAdapter
    Dim GetPersonTable As New _PO_DataDataSet.PersonDataTable
    GetPersonAdapter.Fill(GetPersonTable)

    Dim ReportString As String = vbNullString
    For Each MyRow As _PO_DataDataSet.PersonRow In _
        GetPersonTable.Select("ID = " & PersonID.ToString)
      With MyRow
          ReportString &= "$HDG" & .NameFirst.Trim & " " & .NameLast.Trim & vbCrLf
          ReportString &= "$HD2Contact Details" & vbCrLf
          ReportString &= "Home Phone: " & .PhoneHome.Trim & vbCrLf
          ReportString &= "Cell Phone: " & .PhoneCell.Trim & vbCrLf
          ReportString &= "Address: " & .Address.Trim & vbCrLf
          ReportString &= "Email: " & .EmailAddress.Trim & vbCrLf
          ReportString &= "$HD2Other Details" & vbCrLf
          ReportString &= "Birthday: " & .DateOfBirth.ToShortDateString & vbCrLf
          ReportString &= "Favorites: " & .Favorites & vbCrLf
          ReportString &= "Preferred Gift Categories: "
          Dim GiftString As String = vbNullString
          If (.GiftCategories And 1) <> 0 Then GiftString &= "Books, "
          If (.GiftCategories And 2) <> 0 Then GiftString &= "Videos, "
          If (.GiftCategories And 4) <> 0 Then GiftString &= "Music, "
          If (.GiftCategories And 8) <> 0 Then GiftString &= "Toys, "
          If (.GiftCategories And 16) <> 0 Then GiftString &= "Video Games, "
          If (.GiftCategories And 32) <> 0 Then GiftString &= "Apparel, "
          If GiftString.Length > 0 Then GiftString = _
              GiftString.Remove(GiftString.Length - 2, 2)
          ReportString &= GiftString & vbCrLf
          ReportString &= "$HD2Notes" & vbCrLf
          ReportString &= .Notes & vbCrLf
      End With
    Next
    Return ReportString
End Function
```

2. This routine introduces another flag to indicate a different formatting option — `$HD2` for sub-headings. This needs to be replaced in the printing process with the correct formatting options.

Edit the `POPrintDoc_PrintPage` routine to cater to the new type of formatting:

```
If ReportLines(ReportCounter).Length > 4 AndAlso
ReportLines(ReportCounter).Substring(0, 4) = "$HDG" Then
    ReportLines(ReportCounter) = ReportLines(ReportCounter).Substring(4)
    PrintFont = New Font("Tahoma", 18, FontStyle.Bold)
ElseIf ReportLines(ReportCounter).Length > 4 AndAlso
ReportLines(ReportCounter).Substring(0, 4) = "$HD2" Then
    ReportLines(ReportCounter) = ReportLines(ReportCounter).Substring(4)
    PrintFont = New Font("Tahoma", 14, FontStyle.Bold Or FontStyle.Italic)
```

```
    Else
        PrintFont = New Font("Times New Roman", 12)
    End If
```

3. Change the `Print` and `PrintPreview` menu item event handler routines so they call the appropriate `GenerateReport` function to create the contents of `ReportString`. Change the `PrintPreview` routine to the following:

```
Private Sub printPreviewToolStripMenuItem_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles printPreviewToolStripMenuItem.Click
    ReportString = vbNullString
    If objPersonalDetails IsNot Nothing Then
        ReportString = GenerateReport(mCurrentUserID, objPersonalDetails.Person.ID)
    ElseIf objPersonList IsNot Nothing Then
        ReportString = GenerateReport(mCurrentUserID)
    End If
    If ReportString <> vbNullString Then
        Try
            With prnprvDialog
                .Document = POPrintDoc
                .ShowDialog()
            End With
        Catch PrintPreviewException As Exception
        End Try
    End If
End Sub
```

4. Change the `Print` menu item's routine to use the same logic as the preceding code and run the application.

Exercise 2 Solution

1. Add a `StatusLabel` to the `StatusStrip` and name it `tsPeopleCount`. Set its `Text` property to `0 people` so it is initialized.

2. Add a `ProgressBar` to the `StatusStrip` and name it `tsProgress`. Set its `Visible` property to `False` so that it is not shown by default.

3. Create a new function in `GeneralFunctions` and name it `GetPeopleCount`. Add the following code to return the number of `Person` rows stored with the current user ID:

```
Public Function GetPeopleCount(ByVal UserID As Integer) As Integer
    Dim GetPersonAdapter As New _PO_DataDataSetTableAdapters.PersonTableAdapter
    Dim GetPersonTable As New _PO_DataDataSet.PersonDataTable
    GetPersonAdapter.Fill(GetPersonTable)
    Return GetPersonTable.Select("POUserID = " & UserID.ToString).Length
End Function
```

4. Add the following line of code to the `Form_Load` event of the main form:

```
tsPeopleCount.Text = GetPeopleCount(mCurrentUserID).ToString & " people"
```

Add the same line of code to the `objPersonalDetails_ButtonClicked` event handler if the person is successfully added to the database. In addition, repeat this line of code in the Save button's event handler.

331

5. Change the `GenerateReport` function for the list of people so that it accepts an additional parameter of a `ProgressBar` control. This enables you to reference it as you process each row in the table and increment the `Value` property on the `ProgressBar` control for each row:

```
Public Function GenerateReport(ByVal UserID As Integer, _
    ByVal pnlProgress As ToolStripProgressBar) As String
    Dim GetPersonAdapter As New _PO_DataDataSetTableAdapters.PersonTableAdapter
    Dim GetPersonTable As New _PO_DataDataSet.PersonDataTable
    GetPersonAdapter.Fill(GetPersonTable)

    Dim ReportString As String = vbNullString
    For Each MyRow As _PO_DataDataSet.PersonRow In _
        GetPersonTable.Select("POUserID = " & UserID.ToString)
        pnlProgress.Value += 1
        With MyRow
            ReportString &= "$HDG" & .NameFirst.Trim & " " & .NameLast.Trim & vbCrLf
            ReportString &= "Home Phone: " & .PhoneHome.Trim & vbCrLf
            ReportString &= "Email: " & .EmailAddress.Trim & vbCrLf
            ReportString &= "Birthday: " & .DateOfBirth.ToShortDateString & vbCrLf
        End With
    Next
    Return ReportString
End Function
```

6. Set up the properties of the progress bar in both the `Print` and `PrintPreview` functions and modify the call to the `GenerateReport` function for the person list. Set the `tsProgress.Visible` property to `False` once the process has been completed. Here's the `PrintPreview` function as an example:

```
Private Sub printPreviewToolStripMenuItem_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles printPreviewToolStripMenuItem.Click
    With tsProgress
        .Minimum = 0
        .Maximum = GetPeopleCount(mCurrentUserID)
        .Value = 0
        .Visible = True
    End With

    ReportString = vbNullString
    If objPersonalDetails IsNot Nothing Then
        ReportString = GenerateReport(mCurrentUserID, objPersonalDetails.Person.ID)
    ElseIf objPersonList IsNot Nothing Then
        ReportString = GenerateReport(mCurrentUserID, tsProgress)
    End If
    If ReportString <> vbNullString Then
        Try
            With prnprvDialog
                .Document = POPrintDoc
                .ShowDialog()
            End With
        Catch PrintPreviewException As Exception
        End Try
    End If
    tsProgress.Visible = False
End Sub
```

Chapter 12

Exercises

1. Add events to the Wizard form so the calling application knows when the user navigates between steps.

2. Add an optional attribute to the Text Area component in the Wizard form that enables you to insert a Browse for File dialog.

3. Create an XML Schema Document (XSD) to enforce the structure of the Wizard Definition XML file created in the last Try It Out.

Exercise 1 Solution

1. Define an event at the top of the `WizardBase.vb` code that includes the old and new step numbers:

```
Public Event StepChanged(ByVal OldStep As Integer, ByVal NewStep As Integer)
```

2. Change the `NavigateToStep` subroutine so that it raises the event to the calling application:

```
Private Sub NavigateToStep(ByVal StepNumber)
    StoreNewValues()
    RaiseEvent StepChanged(mCurrentStep, StepNumber)
    pnlControls.Controls.Clear()
    mCurrentStep = StepNumber
    SetForm(mCurrentStep)
End Sub
```

3. Create a `ReadOnly` property called `StepValues` that returns the values for a specified step. This enables the calling code to retrieve values for a step when it receives the event:

```
Public ReadOnly Property StepValues(ByVal StepNumber As Integer) As String
    Get
        Dim myXmlDocument As New XmlDocument
        Dim MyNavigator As XPath.XPathNavigator = myXmlDocument.CreateNavigator()
        Using MyWriter As XmlWriter = MyNavigator.PrependChild()
            MyWriter.WriteStartElement("StepValues")
            With mSteps(StepNumber)
                If .Components IsNot Nothing Then
                    MyWriter.WriteStartElement(.Name)
                    For iComponentCounter As Integer = 1 To _
                        .Components.GetUpperBound(0)
                        MyWriter.WriteElementString( _
                            .Components(iComponentCounter).ComponentName, _
                            .Components(iComponentCounter).ComponentValue)
                    Next
                    MyWriter.WriteEndElement()
                End If
            End With
            MyWriter.WriteEndElement()
        End Using
        Return myXmlDocument.InnerXml
    End Get
End Property
```

4. Open `Form1.vb` in code view and move the definition of the `frmMyExportWizard` object to the top of the class. Declare the `frmMyExportWizard` object `WithEvents` so the program can intercept the new event. Remember to instantiate the form when the button is clicked:

```
Private WithEvents frmMyExportWizard As WizardBase
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    frmMyExportWizard = New WizardBase
    Dim sUserExportSettings As String

    Dim sWizardDefinition As String = _
        My.Computer.FileSystem.ReadAllText("TheFile.xml")

    With frmMyExportWizard
        .WizardDefinition = sWizardDefinition
        .ShowDialog()
        If Not .Cancelled Then
            sUserExportSettings = .WizardSettingValues
        End If
    End With
    frmMyExportWizard = Nothing
    MsgBox(sUserExportSettings)
End Sub
```

5. Add an event handler routine for the `StepChanged` event and display the values from the old step in a message box to confirm that it is retrieving the data correctly:

```
    Private Sub frmMyExportWizard_StepChanged(ByVal OldStep As Integer, ByVal
NewStep As Integer) Handles frmMyExportWizard.StepChanged
        MsgBox(frmMyExportWizard.StepValues(OldStep))
    End Sub
```

6. Run the application and test the new features.

Exercise 2 Solution

1. Create a new `Enum` to specify the types of dialogs that are allowed—Open and Save:

```
    Private Enum AllowedBrowseButtonTypes As Integer
        SaveDialog = 1
        OpenDialog = 2
    End Enum
```

2. Add two new properties to the `WizardComponent` class. One is a Boolean property that indicates whether a Browse button should be shown, and the other stores what type of browse button it is:

```
    Private Class WizardComponent
        Public ComponentControlType As AllowedControlTypes
        Public ComponentName As String
        Public ComponentCaption As String
        Public ComponentValue As String
```

```
        Public ComponentAllowedValues() As String
        Public ComponentBrowseButton As Boolean
        Public ComponentBrowseButtonType As AllowedBrowseButtonTypes
    End Class
```

3. Edit the `GetComponents` routine so that it extracts the information from two new attributes in the XML for a given component — `BrowseButton` and `BrowseType`. These go in the `Select Case ComponentAttribute.Name` block:

```
Select Case ComponentAttribute.Name
    Case "ControlType"
        Select Case ComponentAttribute.Value
            Case "RB"
                .ComponentControlType = AllowedControlTypes.RadioButton
            Case "TB"
                .ComponentControlType = AllowedControlTypes.TextArea
            Case "CB"
                .ComponentControlType = AllowedControlTypes.CheckBox
            Case "CM"
                .ComponentControlType = AllowedControlTypes.ComboBox
        End Select
    Case "Name"
        .ComponentName = ComponentAttribute.Value
    Case "Caption"
        .ComponentCaption = ComponentAttribute.Value
    Case "BrowseButton"
        .ComponentBrowseButton = True
    Case "BrowseType"
        If ComponentAttribute.Value.ToLower = "save" Then
            .ComponentBrowseButtonType = AllowedBrowseButtonTypes.SaveDialog
        ElseIf ComponentAttribute.Value.ToLower = "open" Then
            .ComponentBrowseButtonType = AllowedBrowseButtonTypes.OpenDialog
        End If
End Select
```

4. Add the Browse button if there is a Text Area that has the `BrowseButton` attribute set. Do this in the `AddTextArea` subroutine by modifying the code that adds the `TextBox` as follows. You add the code here so you can modify the `Width` property of the `TextBox` itself to make room for the Browse button:

```
        With newTB
            .Name = "TB" + ThisWizardComponent.ComponentName
            .Text = ThisWizardComponent.ComponentValue
            .Left = newLTBTextSize.Width
            If ThisWizardComponent.ComponentBrowseButton = True Then
                Dim newBRBrowseButton As New Button
                With newBRBrowseButton
                    .Name = "BR" & ThisWizardComponent.ComponentName
                    .Text = "..."
                    .Top = ThisControlTop
                    .Width = 20
                    .Height = newTB.Height
```

335

```
                .Left = pnlControls.Width - (.Width + 5)
                .Tag = ThisWizardComponent.ComponentBrowseButtonType
                AddHandler newBRBrowseButton.Click, AddressOf BrowseButtonClick
            End With
            pnlControls.Controls.Add(newBRBrowseButton)
            .Width = newBRBrowseButton.Left - (.Left + 5)
        Else
            .Width = pnlControls.Width - .Left
        End If
        .Top = ThisControlTop
    End With
```

5. You'll need two `Dialog` controls added to your form. Switch to the Design view of `WizardBase.vb` and add a `SaveFileDialog` control named `BrowseSaveDialog` and an `OpenFileDialog` control named `BrowseOpenDialog`.

6. Return to code view and add the `BrowseButtonClick` routine that is used to handle the button clicks:

```
Private Sub BrowseButtonClick(ByVal sender As System.Object, _
    ByVal e As System.EventArgs)
    Dim CurrentButton As Button = CType(sender, Button)
    ' get the current text
    Dim TBName As String = "TB" & CurrentButton.Name.Substring(2)
    Dim CurrentTB As TextBox = pnlControls.Controls(TBName)

    If CurrentButton.Tag = AllowedBrowseButtonTypes.SaveDialog Then
        With BrowseSaveDialog
            .FileName = CurrentTB.Text
            If .ShowDialog = Windows.Forms.DialogResult.OK Then
                CurrentTB.Text = .FileName
            End If
        End With
    ElseIf CurrentButton.Tag = AllowedBrowseButtonTypes.OpenDialog Then
        With BrowseOpenDialog
            .FileName = CurrentTB.Text
            If .ShowDialog = Windows.Forms.DialogResult.OK Then
                CurrentTB.Text = .FileName
            End If
        End With
    End If
End Sub
```

7. The code is now finished. To test it, edit the `WizardDefs.xml` file to enable users to browse for a file in the `Filename` component and run the application (see Figure C-3):

```
<Component ControlType="TB" Name="Filename" Caption="Filename:" BrowseButton="true"
BrowseType="save">C:\Temp\ExportData.xml</Component>
```
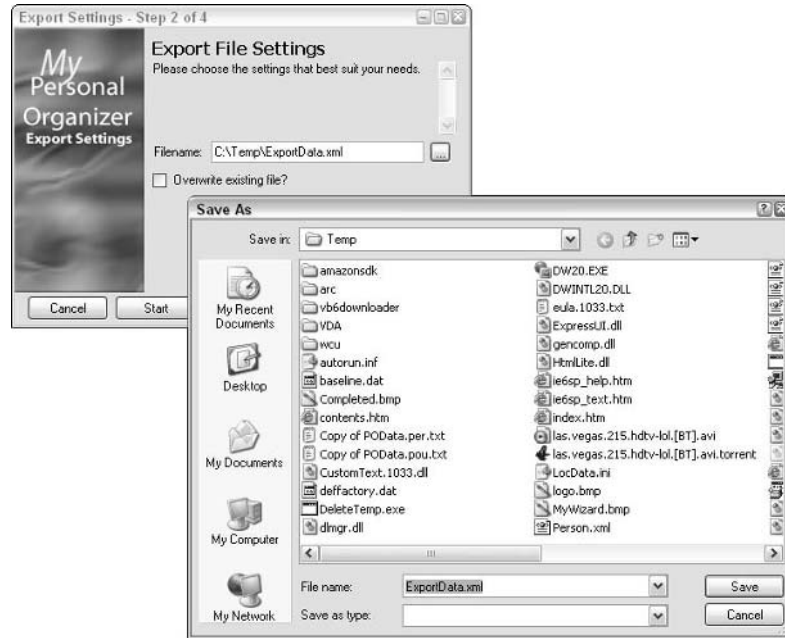
Figure C-3

Exercise 3 Solution

1. A sample XSD for the Wizard Definition file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="Wizard" type="WizardType"/>

  <xs:complexType name="WizardType">
    <xs:sequence>
      <xs:element name="Step" type="StepType" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="Name" type="NameType" use="required"/>
    <xs:attribute name="Title" type="xs:string" use="required"/>
    <xs:attribute name="GlobalGraphic" type="xs:anyURI" use="optional"/>
    <xs:attribute name="AllowFinish" type="xs:boolean" use="optional"/>
  </xs:complexType>

  <xs:complexType name="StepType">
    <xs:sequence>
      <xs:element name="Heading" type="HeadingType"/>
      <xs:element name="Description" type="DescriptionType"/>
      <xs:element name="Graphic" type="GraphicType" minOccurs="0"/>
      <xs:element name="Component" type="ComponentType" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
```

```
      <xs:attribute name="Name" type="NameType" use="required"/>
    </xs:complexType>

    <xs:complexType name="ComponentType" mixed="true">
      <xs:sequence>
        <xs:element name="AllowedValue" type="AllowedValueType" minOccurs="0"
  maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="Name" type="NameType" use="required"/>
      <xs:attribute name="ControlType" type="ControlTypeType" use="required"/>
      <xs:attribute name="Caption" type="xs:string" use="required"/>
    </xs:complexType>

    <xs:complexType name="AllowedValueType" mixed="true">
      <xs:attribute name="Name" type="NameType" use="required"/>
      <xs:attribute name="Selected" type="xs:boolean" use="optional"/>
    </xs:complexType>

    <xs:simpleType name="NameType">
      <xs:restriction base="xs:string">
        <xs:pattern value="([A-Za-z][^ ]*)"/>
      </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="HeadingType">
      <xs:restriction base="xs:string"/>
    </xs:simpleType>

    <xs:simpleType name="DescriptionType">
      <xs:restriction base="xs:string"/>
    </xs:simpleType>

    <xs:simpleType name="GraphicType">
      <xs:restriction base="xs:string"/>
    </xs:simpleType>

    <xs:simpleType name="ControlTypeType">
      <xs:restriction base="xs:string">
        <xs:enumeration value="CB"/>
        <xs:enumeration value="CM"/>
        <xs:enumeration value="RB"/>
        <xs:enumeration value="TB"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:schema>
```

Chapter 13

Exercise

1. While decrypting the password from the database might work for comparing it to the string the user has entered, it's not as secure as it could be. Change the logic so that the `UserPasswordMatches` function encrypts the entered string and compares it to the already encrypted database field to ensure that the fields match.

Exercise 1 Solution

1. Change the `UserPasswordMatches` function so it uses `EncryptString` on the password text the user entered and compares that to the encrypted string in the database:

```
Public Function UserPasswordMatches(ByVal UserName As String, _
    ByVal Password As String) As Boolean
    Dim CheckUserAdapter As New _PO_DataDataSetTableAdapters.POUserTableAdapter
    Dim CheckUserTable As New _PO_DataDataSet.POUserDataTable

    CheckUserAdapter.Fill(CheckUserTable)

    Dim CheckUserDataView As DataView = CheckUserTable.DefaultView
    CheckUserDataView.RowFilter = "Name = '" + UserName + "'"
    With CheckUserDataView
        If .Count > 0 Then
            Dim SecretKey As String = "785&*(%HUYFteu27^5452ewe"
            If .Item(0).Item("Password").ToString.Trim <> vbNullString Then
                Dim EncryptedPassword As String = EncryptString(Password, SecretKey)
                If .Item(0).Item("Password").ToString.Trim = EncryptedPassword Then
                    Return True
                Else
                    Return False
                End If
            Else
                If Password = vbNullString Then
                    Return True
                End If
            End If
        Else
            Return False
        End If
    End With
End Function
```

Chapter 14

Exercise

1. Update the Personal Organizer application to verify that updates work through the ClickOnce publishing process.

Exercise 1 Solution

1. Make any kind of change to the Personal Organizer application. It would be best if it were something that you can easily determine had been changed or not, such as the background color of a form or the wording on a button or dialog box.

2. Publish the solution via Build ➪ Publish Personal Organizer.

3. When the publish is complete, run the application from the Wrox's Starter Kit ➪ My Personal Organizer Start menu item. After a moment, the ClickOnce background checking will inform you of a change to the application and prompt you for an update.

4. Click the OK button to update the application and confirm that your changes have been applied.

Index

SYMBOLS

& (ampersand) `ToolStrip` control text property prefix, 174

* (asterisk) multiplication operator, 71

^ (caret) `SendKeys` method replacement character, 150

= (equals sign) assignment operator, 71

- (minus sign) subtraction operator, 71

{ } (parentheses)
  event parameter delimiters, 99
  subroutine delimiters, 72

% (percent sign) `SendKeys` method replacement character, 150

. (period)
  method prefix, 101
  object suffix, 98
  property prefix, 18, 101

+ (plus sign)
  addition operator, 71
  `SendKeys` method replacement character, 150

" " (quotation marks) XML attribute delimiters, 243

/ (slash) XML closing tag prefix, 243

~ (tilde) `SendKeys` method replacement character, 150

A

`AcceptButton` property, 162

Access Point URL, 181

Access, software program, 33

Actions dialog box, 65

Add ⇨ Class, 95

`Add` method, 88, 193

Add New Item dialog box, 36, 95

Add New Table menu (SQL Server), 37

Add Web Reference Wizard, 180, 181

`AddHandler` statement, 116, 117

`AddMode` property, 138

`AddPerson` function, 136, 138

`AddPersonRow` method, 136, 252

`AddPOUserRow` function, 290

`AddRow` method, 129

`AddSalary` method, 160

alignment
  control, 13, 53, 54, 257
  text, 56, 162

`Allow Nulls` property, 44

`AllowWebBrowserDrop` property, 170, 174

Amazon web service
  `AWSECommerceService` object, 185
  Documentation page, 184–185
  function, calling, 191
  license agreement, 183
  Personal Organizer Database application, integrating with
    `btnGetGiftIdeas_Click` subroutine, 189–190
    `btnSearch_Click` subroutine, 191, 192, 193
    calling web service, 192
    Cancel button, 194
    `Cancelled` property, 195
    `CheckedListBox` control, 186, 191, 193, 194, 195
    debugging, 213
    `GetGiftIdeas.vb` file, 186
    `GiftSuggestions` property, 195
    `ItemSearch` method, 184–185, 191, 192
    `ItemSearchResponse` object, 185, 192–193, 213
    `mbCancelled` variable, 194–195
    `msFavorites` variable, 188
    `msGiftSuggestions` variable, 195
    `PersonDetails` control setup, 185–186, 187, 189
    `RadioButton` control setup, 186–187, 188
    referencing web service, 191
    Save button, 195, 198
    `SelectedText` property, 192
    title, concatenating into string, 195
  registration, 183
  starter kit, building application access using, 20–23
  Subscription ID, 183, 191