

Visual Basic™ .Net Power Tools

Evangelos Petroustos and Richard Mansfield



Acknowledgments

WE WERE FORTUNATE TO have several smart, thoughtful editors assist us in polishing this manuscript. First, we'd like to thank Development Editor Tom Cirtin. He deserves credit for his discernment, and the high quality of his editing. He's very good at dealing with his authors, and equally skilled at raising important questions and improving their chapters.

Technical Editor Greg Guntle carefully reviewed the manuscript and made many useful suggestions, caught a number of inconsistencies, and helped improve several code examples. Production Editor Leslie Light ensured that this book moved smoothly through production and was most helpful with suggestions about the graphics, drawings, and screen shots. Suzanne Goraj, copy editor, combed through every line of our text, making improvements throughout.

To all these, and the other good people at Sybex who contributed to this book, our thanks for the intelligence and care that they brought to this book. In addition, the authors would like to give special thanks to their agents, Matt Wagner and David Fugate, of Waterside Productions, whose contributions to the authors' careers goes above and beyond the call of duty.

Contents at a Glance

<i>Introduction</i>	xix
Chapter 1 • Understanding the .NET Framework	1
Chapter 2 • New Ways of Doing Traditional Jobs	23
Chapter 3 • Serialization Techniques	59
Chapter 4 • Leveraging Microsoft Office in Your Applications	93
Chapter 5 • Understanding .NET Security	119
Chapter 6 • Encryption, Hashing, and Creating Keys	139
Chapter 7 • Advanced Printing	159
Chapter 8 • Upon Reflection	191
Chapter 9 • Building Bug-Free and Robust Applications	215
Chapter 10 • Deploying Windows Applications	243
Chapter 11 • Building Data-Driven Web Applications	271
Chapter 12 • Peer-to-Peer Programming	289
Chapter 13 • Advanced Web Services	319
Chapter 14 • Building Asynchronous Applications with Message Queues	341
Chapter 15 • Practical ADO.NET	391
Chapter 16 • Building Middle-Tier Components	441
Chapter 17 • Exploring XML Techniques	475
Chapter 18 • Designing Data-Driven Windows Applications	505
Chapter 19 • Working with Regular Expressions	543
Chapter 20 • Advanced Graphics	589
Chapter 21 • Designing the User Interface	623

Chapter 22 • Using the .NET Compact Framework and Its
Emerging Technologies

[643](#)

Index

[665](#)

[Team Fly](#)

 Previous

Next 

Contents

<i>Introduction</i>	<i>xix</i>
Chapter 1 • Understanding the .NET Framework	1
Why Read This Chapter	2
Help!	3
Grappling with Framework Class Descriptions	5
The Hunt for a Grammar	6
Why Two Ways?	10
About Constructors	10
Assemblies Three Ways	11
Understanding Data Types	11
About System.Object	12
MemberWiseClone	12
Equals	12
ReferenceEquals	13
The Main Point about Equality	14
GetHashCode	16
GetType	16
ToString	16
Strong Typing Weakens	17
Is Color a Data Type?	17
Exploiting the Framework	18
A Useful Class View Utility	20
A Brief Lexicon	21
Summary	22
Chapter 2 • New Ways of Doing Traditional Jobs	23
Clipboard Access	23

Working with "Control Arrays"	24
Multiple Handles	27
Using Arrays	28
Zero-Based Collections (Sometimes)	28
Initialization	30
Arrays of Objects	30
Array Search and Sort Methods	31
Customized Sorting	33
Many Properties and Methods	34
The Flexible ArrayList	35
Mass Manipulation	36
Data Binding	36
Enumerators	37
Using HashTables	37
New Date/Time Techniques	38
Adding Time	39

Introduction

THIS BOOK ACTUALLY BEGAN in Athens, Greece in 1993. Evangelos Petroutsos wrote a very interesting outline, and several sample chapters, for a book about "fascinating and sophisticated things" you could do with Visual Basic. I agreed with my publisher that his ideas had potential, but Evangelos was a first-time author. I had a track record, though, so the publisher said they'd invest in this "Power Toolkit" book if I agreed to co-author it. Even a small book represents a \$50,000 gamble for a publishing house, and this was a very large book.

I merrily agreed because I thought the topics were compelling—fractals, encryption, processing graphics, animated transitions, multimedia, manipulating color palettes, recursion, and other topics that were largely ignored by other VB books. To our delight, the book became a runaway bestseller in 1995. Evidently many Visual Basic programmers were ready for a book about advanced, cutting-edge programming techniques.

In 2002, we decided to revisit this concept. Nearly a decade has passed, and we now have what amounts to a brand new Visual Basic language: VB.NET. We decided to follow the same path that we went down a decade ago: to explore aspects of VB.NET that have been largely ignored in other books, but are useful or interesting, or both.

Most of the topics covered ten years ago in the previous book are not repeated here—times have changed. But we feel that the subjects explored in this new book are compelling in their own right.

Aesthetics

Why would captivating topics be largely ignored in computer books? We think there are two primary reasons. The first category of ignored topics is seen as "trivial" or "marginal." Put another way, these subjects involve *aesthetics*. Programmers by and large prefer to consider themselves part of the scientific community, so examining such unscientific concepts as beauty or appearance seems to many programmers to be a step down. Two of the chapters in this book, nonetheless, boldly explore aesthetic subjects.

Truth be told, programming is an art, not a science. Some professors conjure up theoretical constructs and special terminology, but airy obfuscation and lofty-sounding jargon do not, by themselves, create a science—and all too often actually inhibit rational discourse.

Studies have shown that the best programmers are frequently English or music majors. Some of the best developers around today got into programming when they purchased their first Amiga computer—an early machine devoted to the creative side of computing. And although academic programming is generally allied with mathematics departments, there is very little real relationship between

math (or science) and programming—just as there is often very little relationship in general between many other academic studies and the real world.

Consider the primary current computer applications: word processing, database management, Internet communications, and spreadsheets. Only spreadsheets have much at all to do with math. Programming *can*, of course, involve math, but it's rarely central to the programmer's task. You could write an entire word processing program without even knowing long division, much less algebra or anything beyond.

And programming obviously isn't a science. Science involves theorizing and controlled experimentation, behaviors rarely associated with programming. Sure, there's a kind of experimental hacking that goes on while trying to fix bugs—but that's not scientific experimentation by any stretch of the imagination. Debugging is much closer to searching for a lost set of keys than sending a kite up into a thunderstorm.

Programming is basically *communication*—albeit between humans and machines. But it is a linguistic and expressive act. It's not exactly *rhetorical* (we don't need to persuade the machines, at least not yet). But it's certainly descriptive, grammatical, and fundamentally communicative.

The two chapters in this book that some will consider "unscientific" are Chapter 21, "Designing the User Interface," and Chapter 20, "Fractals: Infinity Made Visible." We agree. But then we think the entire subject of programming is unscientific, and we're not bothered by that fact.

Complexity and the Avant-Garde

Most of the remaining topics in this book fall into the second category: topics that are either too cutting-edge or too complex for inclusion in many books. For example, not much is written about VB.NET's splendid and extensive security features—even though security is a primary ongoing challenge for the computing community.

Security-related VB.NET programming is avoided not because the programming involved is inherently difficult or novel, but rather because the concepts underlying cryptology and other aspects of security are fundamentally complex. Many computer book authors simply don't know enough about encryption, for example, to explain its implementation in computer programming. Fortunately, cryptology has long been a hobby of one of the authors of this book.

Other topics are perhaps too new to be widely understood or implemented. Asynchronous programming, Web services, employing Office objects, using reflection, and the new .NET Compact Framework (how to squeeze programming and I/O into the highly restrictive platform of small, portable devices such as PDAs and cell phones) all fall into this category.

Several of the chapters in this book, we admit, have been covered fairly extensively in other books (database programming, debugging, printing), but we included them because we feel that we have something new to say. For example, we've yet to find any book that correctly describes how to print hard copy in VB.NET. All the programming examples we've seen either cut letters in half at the end of lines, or cut lines in half at the end of pages. This doesn't happen on *every* line or at the end of every page, but you'll agree that it's pretty bad when it happens even intermittently. If you've been looking for the solution to this problem, see Chapter 7.

Chapter 1 is unusual because it tackles an essential, yet widely avoided, question: Why was Visual Basic .NET designed by C programmers, and what are the implications? It's as if the Romans had been given the job of rebuilding Thebes—the result might be impressive, but it certainly wouldn't remain Egyptian.

Chapter 1 begins like this:

Visual Basic .NET WAS not written by Visual Basic programmers. The entire .NET family of languages was created by C programmers. C—and its cohort OOP—is an academic language. Visual Basic is a popular language. These facts have consequences.

The authors of this book are not beholden to any organization. We're not writing for Microsoft Press, nor are we affiliated with any corporation or school. Indeed, we like to think that we're not dependent on anyone for our paycheck—other than you, dear reader—and can therefore be more objective than many of our colleagues.

We can ask heretical questions such as why OOP should be used in all programming situations as many of its proponents insist. We can question the wisdom of allowing C programmers to write the narratives and code examples for the Help system in VB.NET. We can wonder why structures are included in VB.NET if OOP experts insist that you should *never* use them.

We can freely applaud VB.NET when it improves on traditional VB programming features (streaming and serialization, for instance), and point out when VB.NET creates needless confusion. (*Some* collections in VB.NET are zero-based; some are one-based. And there's no rhyme or reason involved, no pattern you can discover, no rule you can learn, to deal with this problem.)

Another benefit of being outside programming and academic officialdom is that we can be clear. There *is* a lingo developing around programming, and too much of it appears to serve no real purpose other than job protection. If others cannot read your source code, or even understand your comments, then it's likely they'll respect you and you'll keep your job. Likewise, if you follow the party line and keep your geek-speak up-to-date, you'll be *on the team*. So the usual little closed society of a priest class is being built. Remember that only a short time ago mass was said in Latin, a language that the churchgoers couldn't understand. And if you visit a college class in music theory or film theory today, you won't comprehend most of what's being said.

When we *do* now and then indulge in techie jargon in this book, it's usually to let you know what's meant by the latest catchphrases. True, the term *overloaded signature* is used in this book, but right next to it is the parenthetical explanation (*more than one argument list*), just so you'll know what the heck is being discussed. And when terminology, such as *strongly typed*, has several different meanings, we point that out to you.

There's one final benefit derived from the authors' status as independent writers, free of any obligation to particular corporations or institutions: we can be entertaining, or at least less boring than the average computer book. Academic articles and books, including many programming books, deliberately avoid amusing or interesting writing. It's thought in some circles that if your writing isn't obscure or tedious, then you must not be discussing anything sufficiently serious. We take the position that honest, understandable, direct, and interesting writing is preferable to the alternative.

Who Should Read This Book, and Why?

This book is intended to provide solutions for programmers who are ready to take the next step up to more complex, cutting-edge, or sophisticated topics. Chapters 1 and 2 are useful if you're making the transition to VB.NET from another programming language (such as classic Visual Basic, versions 6 and earlier).

This book is, we believe, accessible to any intelligent person with programming experience. We have tried to be clear throughout the book, explaining everything as directly as possible, regardless of degree of difficulty of the various topics.

[Team Fly](#)

 Previous

Next 

Chapter 1

Understanding the .NET Framework

VISUAL BASIC .NET was not written by Visual Basic programmers. The entire .NET family of languages was created by C programmers. C—and its cohort OOP—is an academic language. Visual Basic is a popular language.

These facts have consequences. Visual Basic was conceived in 1990 specifically as an alternative to C. VB was designed as a rapid application-development language—blessedly free of cant and obscurantism. VB was created especially for the small businessman who wanted to quickly put together a little tax calculation utility, or the mother who wanted to write a little geography quiz to help Billy with his homework. VB was programming for the people. Several hundred thousand people use C; millions use Visual Basic.

As with many cultures—Rome versus Egypt, USA versus France, town versus gown—programming languages quickly divided into two camps. C and its offspring (C++, Java, C#, and others) represent one great camp of programmers. Visual Basic is the *other* camp. However, .NET is an attempt to merge Visual Basic with the C languages—while still retaining as much as possible of the famous populist VB punctuation (direct, clear, straightforward, English-like), syntax, and diction.

Many professors, bless them, thrive on abstraction, classification, and fine distinctions. That's one reason why VB.NET is in some ways more confusing than necessary. It has many layers of "accessibility" (scoping) and many varieties of ways to organize data, some more useful than others. It has multiple "qualification" schemes; considerable redundancy; single terms with multiple meanings (*strong typing*, for example); multiple terms for a single behavior (`Imports` versus `Import`); and all kinds of exceptions to its own rules.

VB.NET, however, is clearly an improvement over earlier versions of VB in many respects. We must all find ways of moving from local to distributed programming techniques. And VB.NET is also quite a bit more powerful than previous versions. For example, streaming replaces traditional file I/O, but streaming can also handle data flowing from several sources—not just the hard drive. Streaming considerably expands your data management tools. You can replace a `FileStream` with a `WebResponse` object, and send your data to a Web client.

Nonetheless, in the effort to merge all computer languages under the .NET umbrella, VB had to give up some of its clarity and simplicity. In fact, VB now produces the same compiled code

that all the other .NET languages do—so under the hood, there is no longer any distinction to be made between the two linguistic cultures. It's just on the surface, where we programmers work, that the differences reside.

OOP itself, like biology, involves a complex system of classification. This means that people using OOP must spend a good amount of their time performing clerical duties (Where does this go? How do I describe this? What category is this in? Are the text-manipulation functions located in the Text namespace, or the String namespace? Does this object have to be instantiated, or can I just use its methods directly without creating the object first?)

Why Read This Chapter

If you're one of the millions of VB programmers who, like me, came upon VB.NET with the highly intelligent reaction "Whaaaaa?!?", this chapter might be of use to you.

I'm not unintelligent, and I'm assuming you're not either. But slogging through the VB.NET world makes one *seem* rather slow, especially at first. This chapter gives you some hard-won advice that can save you considerable confusion.

VB.NET is, of course, far easier to navigate if you have a background in C programming languages (and its lovely wife, object-oriented programming).

Millions of VB programmers deliberately decided *not* to use C. That's why we became VB programmers in the first place. We preferred the power of VB's rapid application development tools. We didn't care for the reverse-Polish backward syntax, the redundant punctuation (all those semicolons) and other aspects of C and its daughter languages.

The Internet changed all that—we must develop new skills and adapt to new programming styles. Leaving the cozy and predictable world of local programming (applications for Windows, running on a single computer) requires new techniques. You don't have to switch to C or its ilk, but you do have to expand your VB vocabulary and skills.

Today's programs are sometimes fractured into multiple programlets (distributed applications) residing in different locations on different hard drives and sometimes even using different platforms or languages. Web Services are the wave of the future, and this kind of computing greatly increases the impact of communication and security issues. Not to mention the necessity of encapsulating code into objects.

So gird your loins or whatever else you gird when threatened, and get ready for some new ideas. You've got to deal with some different notions.

Each of us (the authors) has written several books on VB.NET in the past few years — working within the .NET world daily for *three years* now—and we're still discovering new tools, concepts, and features. Part of this is simply getting to know the huge .NET Framework, and part of it is adjusting to OOP and other C-like elements that are now part of VB.

What they say of quantum mechanics applies to OOP: only ten people in the world understand it well, and nobody understands it completely. So be brave. You can learn some patterns and rules to help you get results in .NET, and the benefit is that VB.NET is quite a bit more powerful and flexible than traditional VB. There are considerable rewards for your patience and efforts.

You'll find ideas in this chapter that will deepen your understanding of the great, vast .NET environment and framework. You'll find useful information here that will improve your VB.NET programming—guaranteed. For example: What are structures, and when should you use them? (They're

[Team Fly](#)

 Previous

Next 

a replacement for classic VB's user-defined types, and you should never use them. OOP experts say that whenever you're tempted to use a structure, create a class instead—it's more flexible. Of course other OOP experts disagree, and the squabbling begins.)

Help!

A significant effect of the merging in of VB with C-style languages is that the VB.NET Help system and documentation were mostly written by C programmers. These people are not generally *writers* nor are they very familiar with Visual Basic. That's why you find techno-speak error messages, convoluted descriptions in the Help system, and other foggy patches.

So, instead of VB's justly acclaimed clarity, we get Help descriptions that sound like they were written by a barmy IRS bureaucrat. Here's an example:

Changing the value of a field or property associated with any one instance does not affect the value of fields or properties of other instances of the class. On the other hand, when you change the value of a shared field and property associated with an instance of a class, you change the value associated with all instances of the class.

Got it?

Not only is the VB.NET documentation all-too-often puzzling, the fact that C programmers wrote it means that the descriptions and even the source code examples are often some half-English, half-C beast.

Many Help source code examples listed as "VB.NET" versions are, in fact, written by C programmers. VB programmers must spend the time to translate this faux VB code. It's great that there is now so much tested, bug-free example code in Help. However, perhaps Microsoft would be wise to ask experienced VB programmers to go over the pseudo "Visual Basic" code examples, and translate them into actual VB-style programming.

For example, take a look at the entry in Help for `String.IndexOf`. If you scroll down the right pane, you can see all the ways that the sample code is not typical VB code. Many VB programmers will have to figure out how to actually make this code work. It can't just be copied and pasted.

VB programmers can be confused by some of the strange punctuation and other odd qualities of the following, and many other examples you find in Help. Although *nominally* Visual Basic source code, too many Help examples are alien in many particulars, as you can see in this sample code illustrating the `IndexOf` method:

```
Imports System
Class Sample
Public Shared Sub Main()
Dim br1 As String = _
"0-----1-----2-----3-----4-----5-----6-----"
Dim br2 As String = _
"012345678901234567890123456789012345678901234567890123456"
Dim str As String = _
"Now is the time for all good men to come to the aid of their party."
Dim start As Integer
```

In fact, until VB.NET, the Visual Basic language didn't even permit the use of braces, semicolons, or brackets. Blessed simplicity, including the avoidance of extraneous junk punctuation, has always been a hallmark of Visual Basic.

To make this usable, to-the-point, Visual Basic-style sample code, you have to eliminate the C-flavored elements. What follows is a simplified, and pure-VB.NET, translation of this same sample. In addition to being written in recognizable VB programming style, it also has the advantage of focusing on `IndexOf`, the method being illustrated. The example displayed above from Help is overly complex: involving a loop, word counting, and one of the less frequently used of the `IndexOf` method's over-loaded variations. The idea that the example is supposed to be demonstrating gets lost in a mess of irrelevancies. To be really helpful to VB programmers, Help code and Help narrative explanations should be written by a professional writer/programmer, not simply someone technically competent, with a strong C bias. And the example code should simply illustrate the method being explained, like this:

```
Private Sub Form1_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles MyBase.Load  
    Dim s As String = _  
    "Now is the time for all good men to come to the aid of their party."  
    Dim found As Integer = s.IndexOf("men")  
    Console.WriteLine(found)  
End Sub
```

Grappling with Framework Class Descriptions

You have to learn how to translate the class descriptions in the Object Browser, online documentation, or Help into useable VB code. Sure, there's example code in many Help entries, but that code all too often doesn't precisely demonstrate the syntax you are looking for (it *was* written by C programmers, after all).

Other entries offer no example code at all. There are tens of thousands of members in VB.NET, each with its own signature (parameter list) or set of signatures (thanks to overloading). And as you'll see in this chapter, even seemingly similar classes can require quite different instantiation and different syntactic interactions with other classes to accomplish a particular job.

You therefore frequently have to read a description in Help, the Object Browser, or other documentation and then translate that description into executable source code.

Press `Ctrl+Alt+J` to open the VB.NET Object Browser. Locate `System.IO.File`, then in the right pane locate the first version of the `Create` method, as shown in Figure 1.1. In the lower pane, you see this information:

```
Public Shared Function Create(ByVal path As String) As System.IO.FileStream  
    Member of: System.IO.File
```

Summary:

Creates a file in the specified path.

Parameters:

path: The path and name of the file to create.

Return Values:

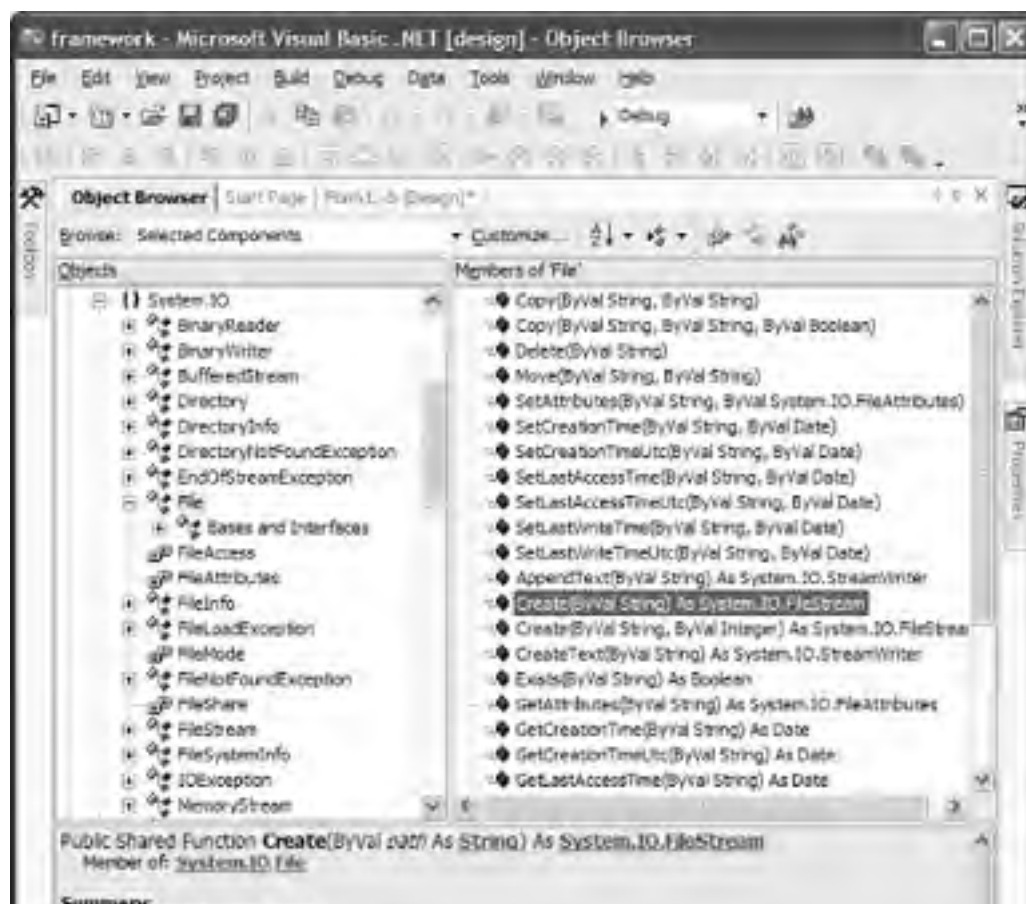
A System.IO.FileStream that provides read/write access to the specified file.

Many VB programmers aren't used to having to interpret this kind of information—VB used to be a simpler language. Now, with VB.NET, it's a new ball game.

You are certain to find yourself often looking at something like the description in Figure 1.1, and wondering how to change this into source code.

The Hunt for a Grammar

We want to think that there is an underlying set of rules, a grammar, that organizes .NET source code. We want to learn the rules so we can instantiate objects, and invoke their methods, without having to continually make educated guesses, then see error messages, then try again by adjusting the syntax, punctuation, or phrasing. We want to assume that the grammar of .NET is consistent—so we don't have to struggle time and again with constructions that follow no particular pattern.



Summary:

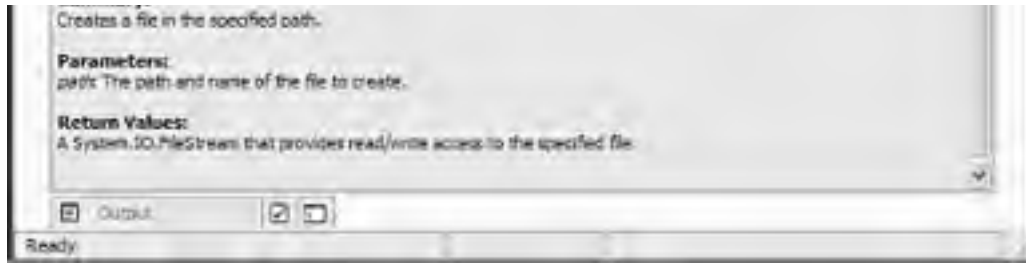


FIGURE 1.1 Often this is all the information you get about how to use a .NET class. Translating this into useable source code is up to you.

its `Sub New` is executed automatically. Because constructors can accept arguments, the person who writes a class can specify which arguments, if any, are required (or optional, in the case of overloaded constructors).

Assemblies Three Ways

Recall that you needn't use an `Imports` statement to use objects in the XML namespace or the data namespace. You *do*, however, need to use `Imports` with other namespaces, including quite commonly used ones such as `System.IO`. There appears to be no rational reason why certain assemblies are referenced by default, and others need to be imported. (One clue is perhaps that nearly any C code you look at always contains `#include <stdio.h>`, the standard I/O library.)

Beyond that, there is even a third class of assemblies that cannot be referenced by `Imports`. They are included in your projects by choosing `Project ➤ Add Reference`.

Another curiosity. You find one list of default assemblies in the References section of Solution Explorer, and a slightly *different* list of assemblies (including the Collections namespace) when you right-click the name of your project (it's boldface) in Solution Explorer, choose Properties from the context menu, then click the Imports option in the left pane of the Property Pages dialog box. You see that only some namespaces here duplicate those in Solution Explorer. What gives?

Understanding Data Types

Earlier you saw how to use the `GetDirectories` method of the `DirectoryInfo` object to obtain a list of subdirectories:

```
Dim di As DirectoryInfo = New DirectoryInfo('c:\')
    Dim dirs As DirectoryInfo() = di.GetDirectories()
```

Here's how you can use the `Directory` object to get a list of files. Notice that you use a string and string array here, rather than `DirectoryInfo` objects. Also, notice that the `Directory` object does not need to be instantiated at all:

```
Dim s As String() = Directory.GetFiles("c:\")
    Dim s1 As String
    For Each s1 In s
        Console.WriteLine(s1)
    Next
```

This is the information from the Object Browser, explaining that you need to assign the results of the `GetFiles` method to a string array:

```
Public Shared Function GetFiles(ByVal path As String) As String()
```

Experienced VB programmers are used to a finite set of variable data types (strings, objects, and a handful of numeric types). Now, in VB.NET, you must get accustomed to the fact that everything is an object. Further, some objects must be instantiated before they can be used (with the New statement), but other objects—viewed by the .NET designers as more "basic" objects, one assumes—do not need instantiation. The DirectoryInfo object does need instantiation; the Directory object does not.

If everything in .NET is an object, then everything is—at least abstractly—a data type. Fortunately, it's not this bad; within a given class, such as `Color`, there are enumerations that in themselves aren't, technically, objects. However, most VB programmers are used to just assigning a simple string (or at least a built-in enum, such as `VBBlue`) to many different properties. For example:

```
BackColor = "blue"
```

Now, in .NET, you must either use `Imports` to bring in a specialized namespace, or fully qualify your value:

```
Me.BackColor = Color.Blue
```

The `System.Drawing` namespace is now included in current VB.NET projects by default, so you don't have to *fully* qualify this one. The underlying problem here, though, is: How can you know that there is a `Color` class, and that it's located in the `System.Drawing` namespace? How do you deal with unfamiliar parts of the .NET Framework? `Color` might be easy enough to imagine, or you might finally come across example code by using the Help search feature to locate entries with `BackColor =`. But how do you deal with more complex situations, such as the `Security` namespace? Most books on VB.NET avoid discussing the `System.Security` assembly precisely because it is both obscure and massive. This in spite of the fact that communication and security are the primary issues that distinguish the .NET world from classical VB programming. Put another way: OOP itself is fundamentally a set of rules designed to solve communication and, especially, security problems. Code reusability, the primary justification for OOP, is largely an attempt to enforce communication rules to solve security problems—though doubtless OOP theorists will consider this a reductive generalization.

Exploiting the Framework

But back to our regular programming. What are the best strategies for tapping into the tremendous power (and consequent complexity) of the .NET Framework?

Remember that it's hierarchical. .NET APIs are divided into namespaces. Namespaces contain a set of related classes. Classes contain methods (and the methods are usually overloaded—permitting you to perform different, but related, jobs based on what parameters you pass).

Let's try to solve a common problem, to see some tactics you can use to locate the solution.

Assume that you have to parse a string. You've got a comma-delimited string from a use like "Barry Morgan, 12 Dalton Ln., Akron, OH, 22022" and you want to subdivide it into its substring parts. You want to create a string array holding each part.

Start by running VB.NET Help, then click the Search tab. Search for parse string. You get 500 hits, including a Dr. GUI article that tells you about the `Parse` method. Unfortunately, it doesn't *parse*, it converts a string into other data types. Somebody incorrectly thinks that "parse"

means convert. When computer languages are written, the specification committees don't include any English majors, so we get too many poorly named functions like this one, and too many nearly unreadable "help" narrative descriptions.

Trying to narrow your search using quotes to look for "parse a string" fails. Let's try the Index feature.

Click the Index tab in the Help screen and type System.Text (no luck here, unless you want to enter the complicated netherworld of Regex, which requires even more elaborate code than using InStr to loop through your string).

[Team Fly](#)

 Previous

Next 

Whoever wrote the Help entry for the Split method evidently was unaware of this approach. The code example employs the bizarre, cumbersome ToCharArray method instead.

Note that VB programmers are unfamiliar with using braces in their programming, but in VB.NET they can be used to fill an array with values, as this example illustrates. The rest of the code is straightforward:

```
Dim delim As Char() = {'','"'}
Dim tt As String = "Barry Morgan, 12 Dalton Ln., Akron, OH, 22022"
Dim split As String() 'create string array
split = tt.Split(delim)
For i As Integer = 0 To split.Length - 1
    Console.WriteLine(split(i))
Next
```

Here's an even more exotic alternative syntax. It combines the declaration of the string array with the declaration of the character array:

```
Dim split As String() = tt.Split(New [Char]() {"",""})
```

This is exotic to VB programmers partly because it uses brackets in addition to braces—neither punctuation is used in classic VB. It's also exotic in that *a single logical line of code* manages to declare two arrays, and to add a value to the second array.

Whatever. It's fine that there are several ways to create and fill a character array. Just fiddle around until you come upon a syntax that works.

A Useful Class View Utility

WinCV (Windows Class Viewer) comes with .NET. It offers you yet another view of the details of every class.

***TIP** If you're not sure where to begin in WinCV or other Framework references, or if you want a useful descriptive overview of the available .NET classes, click on the Contents tab at the bottom of the VB.NET Help window, then follow this path in the tree in the left pane: Visual Studio .NET\ .NET Framework\Reference\Class Library. You'll see all the namespaces there, and all the classes within them.*

To add WinCV to the IDE, choose Tools ➤ External Tools and type WinCV into the Title field. Click the ellipsis ... button next to the Command field. Now find WinCV.exe in this path: C:\Program Files\Microsoft Visual Studio .NET 2003\SDK\v1.1\Bin\WinCV.exe. (If you haven't upgraded to VS.NET 2003, the location may be .NET 2002 or simply .NET.)

Choose Project Directory in the Initial Directory field, and click OK to close the Open File dialog box. WinCV is now available from the VB.NET Tools menu.

A HANDY C# TO VB.NET TRANSLATOR

As you know, unfortunately some .NET Help and reference source code is written in C# (particularly online examples). However, C# is actually quite a simple language to understand and to translate into VB.NET. C# shares some of the backwards syntax of other C languages, and of course all those semicolons—but you can generally switch the C# source code lines around pretty quickly to get usable VB.NET code. However, if you don't want to bother, there's a handy translator that takes C# and generates the equivalent VB.NET. Find it at www.kamalpatel.net/ConvertCSharp2VB.aspx, and the utility is also available with full source code showing the process and the rules. If you prefer, though, you can just paste some C# and get the result automatically.

Choose Tools ➤ WinCV and then type *string* in the Searching For box. You now see all the members of the string class, in a helpful quasi source code format. It's not VB, unfortunately; it's C#—but if you ignore the useless semicolons ending each line, and make a few other allowances (such as changing brackets to parentheses), it often provides a better overall view of a class's members than you get in Help. For example, the entry for the Split method looks like this:

```
public string[] Split((char[] separator));  
public string[] Split(char[] separator,, int count);
```

You have to add the object for the method (String.), so after a little VB massaging it actually should look like this:

```
String() = String.Split(char() separator)
```

That's pretty descriptive pseudo-code. The WinCV is good for quick overviews of classes, their overloaded members, and correct syntax.

A Brief Lexicon

Given that C usage has permeated VB.NET, you might want to memorize the definitions and comparisons in Table 1.1. You'll come upon them now and then in the .NET documentation, and it helps to know what terms have shifted meaning. This small list supplements the other C-derived terms discussed at greater length throughout this chapter. Note that many of these term-pairs are still used interchangeably.

TABLE 1.1: VB vs. VB.NET TERMINOLOGY

TRADITIONAL VB USAGE	.NET
Private or Ppublic variables inside classes	Fields
DLL, library, or application	Assembly
Related DLLs, libraries, or applications	Namespace (imprecise in size: there can be multiple namespaces within a single assembly, or a single namespace can include several assemblies. Namespaces can even contain other namespaces.)

Project (an application and its dependencies)	Solution (design-time source code) or Assembly (runtime executable and support). A Solution can contain multiple projects, in which case you must specify the "startup project" in the Project menu.
Classes, arrays, modules, enumerations, structures, interfaces, and value types, collectively.	Types
Code	Managed code (runs under the control of the .NET runtime library. C++ programmers can choose to write unmanaged code.)
DOS Application	Console Application
Form1_Load	Sub Main
RecordSet	DataSet
Fields and records in databases	Columns and rows in databases
Built-in constants	An enumeration (or enum)
User-defined type	Structure
Error	Exception (or, when working with XML, <i>Fault</i>)
Trapping errors	Handling exceptions
Classes are Public by default	Classes are Friend by default

Summary

In this chapter I tried to explore and describe ways to successfully approach VB.NET. I wanted to help you come to grips with this important language, giving you tools and concepts necessary to master it. Learning to use VB.NET *is* well worth the effort: VB.NET is quite powerful and flexible, a significant improvement over traditional VB.

But there *is* effort required. .NET is a new world. And, alas, Microsoft's documentation for VB.NET (both online and in the Help system) is mostly written in a language somewhat like English, and the code examples are usually similar to VB.NET code. Similar, somewhat like, but...you'll laugh, you'll cry, you'll pull your hair.

All too often the C programmers who wrote the Help descriptions are better programmers than writers. And they're better C programmers than VB programmers, writing code examples that can best be described as Javaesque—Visual Basic from an alternative universe. Close, but not right.

Nonetheless, adapting to a powerful, new, cutting-edge computer technology requires that you sacrifice some time and make some effort. I hope that this chapter has given you guidance and tools that simplify the process. VB.NET—once you're at ease with it—expands your programming abilities in ways you never imagined.

[Team Fly](#)

 Previous

Next 

Chapter 2

New Ways of Doing Traditional Jobs

THIS CHAPTER COVERS VARIOUS techniques that are new in VB.NET and that most programmers need to know about. For example, we must now move to streaming and serialization, capabilities that extend the power of traditional data storage and retrieval, and that address the needs of I/O beyond the local hard drive. These and other important advances are the topic of this chapter.

Clipboard Access

To retrieve text contents from the Windows Clipboard, you used this code in VB version 6 and earlier:

```
Text1.Text = Clipboard.GetText
```

Now in VB.NET you can bring text in from the Clipboard using this code:

```
Dim txtdata As IDataObject = Clipboard.GetDataObject()  
  
' Check to see if the Clipboard holds text  
If (txtdata.GetDataPresent(DataFormats.Text)) Then  
    TextBox1.Text = txtdata.GetData(DataFormats.Text).ToString()  
  
End If
```

To export or save the contents of a TextBox to the Clipboard, use this code:

```
Clipboard.SetDataObject(TextBox1.Text)
```


Working with "Control Arrays"

When you had several controls of the same type performing similar functions, being able to group them into a control array was a valuable feature in classic VB, allowing you to manipulate the group efficiently. Also, a control array was the only way to create a new control (such as a brand-new TextBox or a new group of buttons) while a program was running.

Grouping controls into an array lets you manipulate their collective properties quickly. Because they're now labeled with numbers, not text names, you can use them in loops and other structures (such as `Select Case`) as a unit, easily changing the same property in each control by using a single loop or index-based scheme. Similarly, you can collapse all their individual Click events into a single, collective Click event.

There were several ways to create a control array, but probably the most popular was to set the index property of a control during design time. During runtime, you can use the Load and Unload commands to instantiate new members of this array.

Each control in a control array gets its own unique index number, but they share every event in common. In other words, one Click event, for example, would be shared by the entire array of controls. An Index parameter specified which particular control was clicked. So you would write a `Select Case` structure like the following within the shared event to determine which of the controls was clicked and to respond appropriately:

```
Sub Buttons_Click (Index as Integer)
  Select Case Index
  Case 1
    MsgBox ("HI, you clicked the OK Button!")
  Case 2
    MsgBox ("Click the Other Button. The one that says OK!")
  End Select
End Sub
```

(There is a way to simulate this all-in-one event that handles all members of a control array in VB.NET. It is described in the following section, "Multiple Handles.")

Control arrays have now been removed from the language. However, in VB.NET you can still do what control arrays did. You can instantiate controls during runtime, and also manipulate them as a group. You just use different techniques.

To accomplish what control arrays used to do, you must now instantiate controls (as objects) during runtime and then let them share events (even various different *types* of controls can share an event). Which control (or controls) is being handled by an event is specified in the line that declares the event (following the Handles command, as you'll see in the next example). Instead of using index numbers to determine what you want a control to do (when it triggers an event), as was the case with control arrays, you must now check an object reference. You are also responsible for creating events for runtime-generated controls. The Name property can now be changed during runtime.

***WARNING** Experienced VB programmers will expect VB.NET to assign names to dynamically added controls. However, be warned that VB.NET does not automatically assign names to new controls added at design time. Therefore, the Name property remains blank unless you specifically define it, as you will do in the following example (`textBox1.Name = "TextBox1"`)*

[Team Fly](#)

 Previous

Next 

TIP VB.NET creates the event in the code window for you, if you wish. Your `btnSearch` doesn't show up in the Design window, so you cannot double-click it there to force VB.NET to create a Click event for it. However, you can use the dropdown lists. After you have declared a control `WithEvents` (`Dim WithEvents btnSearch As New Button()`), drop the list in the top left of the code window, and locate `btnSearch`. Click it to select it. Then drop the list in the top right, and double-click the event that you want VB.NET to create for you in the code window.

TIP Each form has a collection that includes all the controls on that form. You access the collection, as illustrated previously, using `Me.Controls` or simply `Controls`. The collection can be added to, as shown in the previous example, or can be subtracted from `Me.Controls.Remove(Button1)`.

Note, too, that the `Me.Controls` collection also has several other methods: `Clear`, `Equals`, `GetChildIndex`, `GetEnumerator`, `GetHashCode`, `GetType`, `SetChildIndex`, `ShouldPersistAll`, and `ToString`. There are also three properties available to `Me.Controls`: `Count`, `Item`, and `IsReadOnly`.

Using Arrays

You probably should familiarize yourself with all the new, significant members available in VB.NET for the collection classes, including the various kinds of arrays.

Arrays can now contain objects (technically, that's *all* they now contain) and can search and sort themselves, and the new `ArrayList` class is especially worthwhile.

Zero-Based Collections (Sometimes)

Arrays are *always* zero-based in the .NET Framework. In classic VB you could use the `Option Base` statement to allow arrays to start with element 1 instead of 0. `Option Base` has been deleted from VB.

Therefore, you must wrestle with the artificial distinction between *dimension* (the size you declare) and *capacity* (the number of elements). For decades now, programmers have had to fiddle with their loop values to fix this silly distinction:

```
Dim a(3) As String
For i = 0 To a.Length - 1
```

Because dimensioning this array as 3 *actually creates 4 elements*, you must therefore subtract 1 from your loop counter.

We humans always count up from 1 when dealing with collections (lists, sets, groups, and so on). It's natural to our way of describing, and therefore *thinking about*, numbers. When the

first person arrives at your BBQ, you don't say "Welcome, you're the zeroth one here!" And when your child is one year old, you don't send out invitations titled "Jimmy's Zeroth Birthday Party!!" We quite properly think of zero as meaning *nothing*—absence, nonexistence.

You've doubtless had to fiddle around with this foolishness many times in your programming career. The old familiar error message, "An unhandled exception of type 'System.IndexOutOfRangeException' occurred...", has been unnecessarily triggered millions of times. *Unnecessarily* because mathematical diction, fundamental logic, elementary grammar, and simple common sense all require that lists begin with the first (not the zeroth) item. Computer languages, though, are designed by a certain kind of committee—a group that does *not* invite language specialists, such as English majors, to the table.

The Flexible ArrayList

The ArrayList, new in VB.NET, offers a variety of helpful features not typical of ordinary arrays. For one thing, it can dynamically resize itself, so you don't have to resort to ReDim and other techniques that an ordinary array can demand.

Here's one way to use an ArrayList to add values:

```
Dim MyArray as new ArrayList
    myArray.Add ("key")
    myArray.Add ("Name")
    myArray.Add ("Address")
Msgbox (MyArray(2))
```

Both array and ArrayList .NET classes can sort, search, reverse, and otherwise manipulate their data. The ArrayList, however, takes the idea of an array to new levels. One problem with arrays is that you can't easily add or delete items. If you want to remove, say, the tenth item in an array, you must write some code that loops through the array, moving each value down one in the index list from the tenth item up to the final element.

The ArrayList has built-in facilities to automatically handle any resizing and re-indexing that's needed if you insert or delete elements.

Put a ListBox and a Button on a form. Then type in the code in Listing 2.6, which illustrates how you can remove an element by using the RemoveAt method and specifying an index number.

LISTING 2.6: USING THE REMOVEAT METHOD TO DELETE AN ARRAY ELEMENT

```
Public Class Form1

    Inherits System.Windows.Forms.Form

    Public arrList As New ArrayList()

    Private Sub Form1_Load(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles MyBase.Load
        arrList.Add("ET" )
        arrList.Add("Pearl Harbor" )
        arrList.Add("Rain" )

        ListBox1.Items.AddRange(arrList.ToArray)

    End Sub

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

example with a `ListBox` and `Button`, replace the `Button`'s `Click` event with this code to see how to bind an `ArrayList` to a `ListBox`:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    Dim Monkey As New ArrayList()
    Monkey.Add('A')
    Monkey.Add("B")
    Monkey.Add("C")
    Monkey.Add("D")
    Monkey.Add("E")
    Monkey.Add("F")
    ListBox1.DataSource = Monkey
End Sub
```

Enumerators

You're used to looping with `For...Next`, `While`, and other structures, but now Microsoft encourages us to use *enumerators* when looping through a collection class. This example illustrates how to rewrite the previous example to display both elements in the *RangeOfArrayList* `ArrayList`:

```
Dim RangeArrayListEnumerator As System.Collections.IEnumerator = _
    RangeOfArrayList.GetEnumerator()
While RangeArrayListEnumerator.MoveNext()
    Console.WriteLine(RangeArrayListEnumerator.Current)
    Console.WriteLine()
End While
```

Using HashTables

The collection class called a *HashTable* is quite similar to the `ArrayList` in both design and features. However, a `HashTable` permits "strong data typing": You can give each element a *name* in addition to its index number.

In some situations, it's easier to work with a collection if each element is labeled. Say your collection holds the foods eaten by each animal in your private zoo. It's simpler to manage the data if each element is named after a different animal:

```
Dim Food As New Hashtable()
Food.Add("Lion", "Meat")
Food.Add("Bear", "Meat")
Food.Add("Penguin", "Fish")
Console.WriteLine(Food.Item("Bear"))
```

In this example, the names of the animals can be used as keys to access the elements, instead of index numbers. Each key must be unique, though the data itself can be duplicated ("meat" and "meat" in this example).

New Date/Time Techniques

Before VB.NET, the Date function used to give you the current date (for example, 11/29/00). The Time function used to give you the current time. Now you must use the Today and TimeOfDay functions instead.

NOTE The old DATES\$ and TIMES\$ functions have been eliminated.

In Visual Basic 6.0 and previous versions, a date/time was stored in a double (double-precision floating point) format (four bytes). In VB.NET, the date/time information uses the .NET Framework DateTime data type (stored in eight bytes). There is no implicit conversion between the Date and Double data types in VB.NET. To convert between the VB6 Date data type and the VB.NET Double data type, you must use the ToDouble and FromOADate methods of the DateTime class in the System namespace.

Here's an example that uses the TimeSpan object to calculate how much time elapsed between two DateTime objects:

```
Dim StartTime, EndTime As DateTime
Dim Span As TimeSpan
    StartTime = "9:24" AM
    EndTime = "10:14" AM
    Span = New TimeSpan(EndTime.Ticks - StartTime.Ticks)
    MsgBox(Span.ToString)
```

Notice the Ticks unit of time. It represents a 100-nanosecond interval.

Here's another example illustrating the AddHours and AddMinutes methods, how to get the current time (Now), and a couple of other methods:

```
Dim hr As Integer = 2
Dim mn As Integer = 13
Dim StartTime As New DateTime(DateTime.Now.Ticks)
Dim EndTime As New DateTime(StartTime.AddHours(hr).Ticks)
EndTime = EndTime.AddMinutes(mn)
Dim Difference = New TimeSpan(EndTime.Ticks - StartTime.Ticks)
Debug.WriteLine("Start Time is: " + StartTime.ToString("hh:mm"))
Debug.WriteLine("Ending Time is: " + EndTime.ToString("hh:mm"))
```

```
        Debug.WriteLine('Number of hours elapsed is: ' + Difference.Hours.ToSt
        Debug.WriteLine("Number of minutes elapsed is: " + _
Difference.Minutes.ToString)
```

The following sections provide some additional examples that illustrate how to manipulate date and time.

Adding Time

Here's an example of using the `AddDays` method:

```
Dim ti As Date = TimeOfDay 'the current time
Dim da As Date = Today 'the current date
Dim dati As Date = Now 'the current date and time
da = da.AddDays(12) ' add 12 days
Debug.WriteLine("12 days from now is: " & da)
```

Similarly, you can use `AddMinutes`, `AddHours`, `AddSeconds`, `AddMilliseconds`, `AddMonths`, `AddYears`, and so on.

Using the Old-Style Double DateTime Data Type

There is an OA conversion method for currency data types and for date data types. (OA stands for *Ole Automation*, a legacy technology that still keeps popping up.) Here is an example showing how to translate to and from the old double-precision date format:

```
Dim dati As Date = Now 'the current date and time
Dim da as Date, n As Double
n = dati.ToOADate ' translate into double-precision format
n = n + 21 ' add three weeks (the integer part is the days)
da = Date.FromOADate(n) ' translate the OA style into .NET style
Debug.WriteLine(da)
```

Use `Now`, not `Today`, for these OA-style data types.

Finding Days in a Month

2004 is a leap year. Here's one way to prove it:

```
Debug.WriteLine("In the year 2004, February has " & _
Date.DaysInMonth(2004, 2).ToString & " days.")
Debug.WriteLine("In the year 2005, February has " & _
Date.DaysInMonth(2005, 2).ToString & " days.")
```

File I/O (Streaming)

The classic familiar VB file opening syntax is this:

```
Open filepath {For Mode}{options}As {#} filename {Len = recordlength}
```


Form References: Communication Between Forms

Before VB.NET, you could reference a form's properties in code inside that form by merely specifying a property (leaving off the name of the form):

```
BackColor = vbBlue
```

Or to reference a form from outside that form: If you want to show or adjust properties of controls in one form (by writing programming in a second form), you merely use the outside form's name in your code. For instance, in a `CommandButton_Click` event in `Form1`, you can `Show Form2`, and change the `ForeColor` of a `TextBox` on `Form2`, like this:

```
Sub Command1_Click ()  
Form2.Show  
Form2.Text1.ForeColor = vbBlue  
End Sub
```

Now in VB.NET, when you reference a form's properties from code inside the form, you must use `Me`:

```
Me.BackColor = Color.Blue
```

And to manipulate a form's contents from outside the form: Say that you want to be able to contact `Form2` from within `Form1`. You want to avoid creating clone after clone of `Form2`. If you use the `New` statement willy-nilly all over the place (`Dim FS As New Form2`), you'll be propagating multiple copies of `Form2`, which is not what you want. You don't want lots of windows floating around in the user's Taskbar, all of them clones of the original `Form2`. Remember that every time you use `As New`, you instantiate a new object.

Instead, you want to be able to communicate with the single, original `Form2` object from `Form1`. But how can you do that? How can you create an object variable in `Form1` that references `Form2`?

One way to do this is to create a public variable in `Form1`, like this:

```
Public f2 As New Form2  
Private Sub Form1_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles MyBase.Load  
    f2.Show()  
    f2.BackColor = Color.Blue  
End Sub
```

`Form1` is instantiated first when a VB.NET project executes (by default, it is the "startup object") in any Windows-style VB.NET project. So, by creating a `Public` variable that instantiates `Form2` (the `New` keyword does that), you can then reference this variable (`F2` here) any time you need to manipulate `Form2`'s properties or methods from within `Form1`. It's now possible for `Form1` to be a client of `Form2`, in other words.

The problem of communicating from Form2 to Form1, however, is somewhat more complex. You cannot use the New keyword in Form2 or any other form because that would create a *second* Form1. Form1 already exists because it is the default startup object.

[Team Fly](#)

 Previous

Next 

```
    If e.KeyCode = Keys.N And e.Control = True Then
        'they pressed CTRL+N
        searchnext() 'respond to this key combination
        Exit Sub
    End If
End Sub
```

Loading Graphics with LoadPicture

Before VB.NET, you put a graphic into a PictureBox with this code:

```
Set Picture1.Picture = LoadPicture('C:\Graphics\MyDog.jpg')
```

Now in VB.NET, LoadPicture has been replaced with the following code:

```
PictureBox1.Image = Image.FromFile("C:\Graphics\MyDog.jpg")
```

Managing the Registry

Although .NET applications avoid using the Registry, you may still nonetheless need to access it. Where should a VB.NET programmer store passwords or other customization information (such as the user's choice of default font size) instead of the Registry that you've used for the past several years? Cookies? What goes around comes around. You can go back to using good old once-disgraced .INI files, or similar simple text files (though they can be deleted). They are, however, quick and easy, and using them avoids messing with the Registry.

In VB6 and before, you could use API commands such as RegQueryValueEx to query the Registry. Or you could employ the native VB Registry-related commands such as GetSetting, like this:

```
Print GetSetting(appname := "MyProgram" , _
    section := "Init" , key := "Locale" , default := "1")
```

If you must use the Registry, here's how to access it from VB.NET. In VB.NET, you can query the Registry using the RegistryKey object. Type Listing 2.13 into a button's Click event.

LISTING 2.13: MANAGING THE REGISTRY

```
Private Sub Button1_Click_1(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click

    Dim objGotValue As Object
    Dim objMainKey As RegistryKey = Registry .CurrentUser
    Dim objOpenedKey As RegistryKey
```

```
Dim strValue As String

objOpenedKey = objMainKey.OpenSubKey _
(' Software\Microsoft\Windows\CurrentVersion\Internet Settings )

objGotValue = objOpenedKey.GetValue( "User Agent"

If (Not objGotValue Is Nothing) Then
    strValue = objGotValue.ToString()
Else
    strValue = " "
End If

objMainKey.Close()

TextBox1.Text = strValue

End Sub
```

You must also add `Imports Microsoft.Win32` up there at the top of the code window where all those other `Imports` are. The `Microsoft.Win32` namespace contains the Registry-access functions, such as the `OpenSubKey` method that you need in this example.

Press F5 to run this example, and click the button. If your Registry contains the same value for this key as my Registry contains, you should see a result similar to this:

```
Mozilla/4.2 (compatible; MSIE 5.0; Win32)
```

Note that the complete name (path) of the entire Registry entry is divided into three different locations in the example code (they are in boldface): first the primary key, `CurrentUser`, then the path of subkeys, and finally the actual specific "name":
`objOpenedKey.GetValue("User Agent").`

Writing to the Registry

The `RegistryKey` class includes a group of methods you can use to manage and write to the Registry. These methods include `Close`, `CreateSubKey`, `DeleteSubKey`, `DeleteSubKeyTree`, `DeleteValue`, `Get-SubKeyNames`, `GetType`, `GetValue`, `GetValueNames`, `OpenSubKey`, and `SetValue`.

Random Numbers

In VB6 and previous versions, you would generate a random number between 1 and 12 like this:

```
X = Int(Rnd * 12 + 1)
```

Or to get a random number between 0 and 12, you would use this code:

`X = Int(Rnd * 13)`

You used the Rnd and Randomize functions.

[Team Fly](#)

 Previous

Next 

```
For i = 0 To a.Length - 1
    Debug.WriteLine(i.ToString + ' ' . + " a(i).ToString)
Next
```

WARNING *Neither the old Rnd function nor the new Random object uses algorithms that are sufficiently sophisticated to be of direct use in most kinds of cryptology.*

SendKeys

You can still use the Shell command to run an application, such as Notepad. Shell works much as it did in VB6. An associated command, SendKeys, imitates the user typing on the keyboard. SendKeys works differently in VB.NET. This code will run an instance of Windows's Notepad, and then "type" This message into Notepad:

```
Dim X As Object
X = Shell("notepad.exe" , AppWinStyle.NormalFocus)
System.Windows.Forms.SendKeys.Send("This Message")
```

WARNING *If you put this code in a Form_Load event, it will only send the T into otepad (there are timing problems involved). So, put it into a different event, such as Button1_Click, and VB.NET will have enough time to get itself together and send the full message.*

Serializing

It's easy enough to store a text file. You just save it as Unicode characters, byte-pairs, or whatever. Simple, consistent variables are easily handled by the streaming techniques described earlier in this chapter.

When storing a simple integer or string variable, for example, there are only three things to worry about: the variable's name, its type, and its value. Such simple entities can just be directly streamed.

However, other kinds of data need to be stored or retrieved. More complicated constructions, such as arrays or objects, require that you also store an internal organization (the hierarchy, or other metadata). Such objects are sometimes fairly elaborate. What's more, there's no known pattern. Classes are defined by programmers. So how can VB.NET know in advance what to store or retrieve, the way it knows all about storing integers and strings?

The answer to streaming more complicated or unique data constructions is serialization. Serialization deconstructs a complicated construction into data and metadata that can be streamed. This deconstruction preserves the internal order of the construction, data types, scope, assemblies, and other details that must all enter a stream and be saved (or be retrieved). Serialization can handle objects, arrays, rectangles, and pretty much any other complex data construction. (I'm using the term *construction* rather than *structure* so you won't be puzzled by the other use of *structure* in VB.NET.)

VB.NET includes considerable serialization facilities. You can, for example, pick and choose which fields in a class you want serialized. To exclude a particular field from serialization, use the following syntax:

```
<NonSerialized(> Public Secrets As String
```

[Team Ely](#)

 Previous

Next 


```
        Console.WriteLine(d1.Price)
        Console.WriteLine(ar(2))

    End Sub

End Class
```

Notice that in this example, you created a stream (`fs1`) and then used it to deserialize *both* your Donut structure and the ArrayList. One stream can handle multiple serializations or deserializations.

Summary

This chapter is a kind of mini-encyclopedia of the primary differences between classic Basic's popular techniques and the way they're handled—the dissimilar way they're handled—in VB.NET.

I chose these particular techniques because of their usefulness, and the frequency with which most programmers use them in their projects, but also because they are handled in what traditional Basic programmers may consider unusual, novel, or counterintuitive ways. For example, in .NET setting Properties in Form1 from within the code of Form2 is not straightforward (this used to be easy).

You should be able to glance through the section titles in this chapter to locate the solution to a problem that's bothering you. I cover saving and loading data, including new techniques such as serialization (which saves disparate kinds of data); control arrays and other types of arrays, and the new methods available for sorting and searching them; date/time manipulations; random numbers; trapping keypresses; managing the Registry; loading graphics; the new flexible data binding; and so on. I don't claim that this chapter contains anywhere near the total list of differences between traditional Basic and VB.NET—only that it's a collection of many of the more significant, and less obvious, differences.

Chapter 3

Serialization Techniques

IN CHAPTER 2 YOU learned how to read and write basic data types from and to files using streams. As far as file manipulation goes, Visual Basic got a real facelift. However, most practical applications don't store simple numeric values and strings to files. They need to store objects and, quite often, collections of objects. These objects may be built-in objects (such as Rectangle, Color, and other simple objects) or custom objects. This is where serialization comes in. Serialization is one of the truly exciting features introduced with the .NET Framework.

Serialization is the process of converting an arbitrary object, or collection of objects, into a stream of bytes, suitable for transmission to another process or another computer, or for persisting to a disk file. In the preceding chapter you learned how to store information to files using streams. When you use streams to write information to a file, or read it back from a file, you're responsible for formatting your data and writing them to the file. To read back the data, you must know what data types you're reading from the file and place them into appropriate variables.

Serialization goes beyond saving data to a file. It's a mechanism for saving an object in a way that makes it easy to reconstruct it later using the reverse process, which is called *deserialization*. Serializing an object means saving its properties. The serialization process doesn't persist the definition of an object, just its state. In other words, you can't serialize the methods of an object. You can serialize the values of its properties, so that you can later reconstruct an instance of the same object that will be in the same state as the object you serialized. The application that will deserialize the object must have access to the object's code (i.e., the class from which the object was instantiated), so that it can recreate the persisted object. Most importantly, you don't have to specify how each property is serialized. The serialization classes of the .NET Framework will determine how each data type is serialized and will read back the values of the serialized properties.

Serialization is not entirely new to the .NET Framework. VB6 programmers are familiar with the PropertyBag object, which we used to store instances of objects. .NET's serialization classes are more flexible and powerful and they go beyond the binary format.

How Serialization Works

Let's say you have a class named `Person`, which stores information about persons (customers, contacts, and so on). The `Person` class obviously exposes properties such as `Name`, `Address`, `PhoneNumber`, and so on. Depending on the application, the `Person` class may store a person's date of birth, a customer's credit limit, or just about any property you will need in your code. For each person you want to manipulate in your code, you'll create an instance of the `Person` class and populate this instance with different data. Each instance of the `Person` class is a `Person` object and its state is determined by the values of its properties. To persist an object of the `Person` type, you need only save the values of its properties—this is what serialization is all about. The class may also expose methods that act on the data, but code is not serialized.

To read back the persisted values into the same application, you simply create new instances of the `Person` class and populate them with the values of the serialized properties. Once all properties and fields have been assigned values, you have an object that's identical to the original one (the object you serialized) and you can call any of the object's methods to manipulate them.

Assuming that the application that deserializes the object has access to the code of the `Person` class, it's almost trivial to serialize and deserialize objects. You create an instance of the appropriate `Serializer` class and call its `Serialize` and `Deserialize` methods. Serialized objects can also be used by applications that don't have access to the code of the class that produced the objects. The remote application can't recreate identical objects, but it can use the serialized data to reconstruct an object with a similar structure. Of course, the remote application will never call the methods of the original class, because it can't access them. As you will see, it's possible to serialize an object in XML format. XML contains a description of the object's values, making it possible to create a new class with similar properties. For example, the remote application can create a `Customer` class, which can read some of the serialized properties (it can use the `Name` and `Address` properties, say, but skip the birth date). When you serialize objects in XML, the resulting document holds not only data, but its structure as well. Another application can take advantage of the self-descriptive nature of the XML document and reuse the data.

Serialization Types

There are three types of serialization: binary serialization, SOAP serialization, and XML serialization. Binary and SOAP serialization are very similar; XML serialization is a little different, but it allows you to customize the serialization process. Binary serialization is performed with the `Binary-Formatter` class and it converts the values of the object's properties into a binary stream. The result of the binary serialization is very compact and the serialization/deserialization process is as fast as it can get. However, binary serialized objects can be used only by applications that have access to the code of the class that produced the objects and can't be used outside .NET. Another limitation of binary serialization is that the output it produces is not human-readable and you can't do much with a file that contains a binary serialized object without access to the original class's code. Because binary serialization

is very compact and very efficient, it's used almost exclusively to persist objects between sessions of an application, or between applications that share the same classes. For example, you can create an ArrayList of Person objects, serialize them to a file and reload the collection of the serialized objects from the file in a later session of the same application.

[Team Ely](#)

 Previous

Next 

The last few statements in the event handler extract the two objects (the Rectangle and Bitmap object) from the ArrayList and display some of their basic properties. The last statement retrieves the bitmap stored in the second item of the reconstructed ArrayList, casts it to the Bitmap type, and then uses it as the form's background image. The code displays the bitmap on the form to demonstrate that the bitmap was preserved during the serialization process.

BINARY SERIALIZATION PRESERVES TYPE FIDELITY

If you serialize the same ArrayList in SOAP and binary formats, you'll realize that the sizes of the two files are very different. The file that stores the binary representation is actually much smaller than the file with the SOAP representation of the same object. The binary file's size is nearly the same as the image file, with a few dozen more bytes for the Rectangle object. The SOAP file is nearly twice as large. The difference is not due to any form of compression; the SOAP file is quite verbose.

This remark brings us to a very important point about the various types of serialization: type fidelity. Binary serialization preserves type fidelity, because it has stored the bitmap as a GIF file. The SOAP serialization, on the other hand, has stored the actual bitmap, without the compression built into the GIF file. If you open the `Objects.bin` file generated by the `SoapFormatter`, you'll see that the image's bytes are encoded in Base64, but the element under which the bitmap is stored is called "Bitmap." In other words, the `SoapFormatter` serialized the actual bitmap, not the GIF file from which the Bitmap object was constructed.

In most cases, such extreme fidelity won't matter. After all, a bitmap is a bitmap, and as long as you can reconstruct the image from the serialized data you shouldn't care how the image was serialized.

An object can be serialized in binary and SOAP format with the same statements. You can uncomment the statements that refer to the `SoapFormatter` and comment out the statements that refer to the `BinaryFormatter` to test both serialization techniques. Keep in mind that binary serialization uses the CLR data types and generates the most accurate representation of the objects. However, it's limited to .NET; you can't use a binary serialized object outside .NET. If you need to exchange serialized objects with other systems, use SOAP, or XML, serialization.

Creating Serializable Objects

Just about any custom class created in .NET can be serialized, as long as it's marked with the `<Serializable>` attribute. Many of the built-in objects are serializable, but not all of them. Unfortunately, nonserializable .NET classes are not clearly marked as such in the documentation. As you can understand, all basic data types in .NET are serializable, and so are some of the collections (arrays and ArrayLists are serializable, but the HashTable isn't). If you want to serialize an ArrayList with numbers, or strings, you don't have to create a custom class, or do anything special. You simply call the `Serialize` method of the appropriate `Serializer` class and the collection will be persisted to a stream, which in turn will move the serialized data to a disk file.

.NET programming means developing and using custom classes. If you want to be able to serialize your custom classes, all you have to do is prefix their declaration with the `<Serializable>` attribute. You don't have to mark any of the class's public fields as

serializable, but you can exclude selected fields from the serialization process by marking them with the `<NonSerializable>` attribute. Fields

[Team Fly](#)

 Previous

Next 

There's a substantial overhead the first time you create an instance of the XmlSerializer class. This process, however, isn't repeated during the course of the application. The overhead is due to the fact that the CLR creates a temporary assembly for serializing and deserializing the specific type. This assembly, however, remains in memory for the course of the application and the initial overhead won't occur again. This means that, although there will be an additional delay of a couple of seconds when the application starts (or whenever you load the settings), you can persist the class with the application's configuration every time the user changes one of the settings without any performance penalty.

Custom Serialization

The .NET Framework makes it possible to override the default serialization process and take complete control of how your objects are serialized and deserialized. You may wish to control the serialization process if you want to serialize more than just public fields, but not every aspect of an object. To build classes that control their own serialization, you must first make sure that they implement the ISerializable interface:

```
<Serializable( )> _  
Public Class Employee  
    Implements ISerializable
```

The ISerializable interface contains a single method, the GetObjectData method, whose signature is the following:

```
Sub GetObjectData(ByVal info As SerializationInfo, _  
                 ByVal context As StreamingContext)
```

The SerializationInfo class exposes the AddValue method, which adds members to the serialized object. This method accepts as argument a key–value pair, as demonstrated in the following sample code. The first three fields can be public or private. The last field, CreationDate, need not even be a member of the class—it's created and populated during the serialization. The custom deserializer can take this field's value into consideration, or ignore it completely.

```
Private Sub GetObjectData(ByVal info As SerializationInfo, _  
                         ByVal context As StreamingContext) _  
    Implements ISerializable.GetObjectData  
    info.AddValue('classField1" , value1)  
    info.AddValue("classField2" , value2)  
    info.AddValue("classField3" , value3)  
    info.AddValue("CreationDate" , DateTime.Now())  
End Sub
```

The custom deserialization process is implemented as an overloaded form of the class's constructor, which has the following signature:

```
Friend Sub New(ByVal info As SerializationInfo, _  
              ByVal context As StreamingContext)
```


You can exploit this information to include additional information with (or exclude information from) the object you serialize, depending on its context. If the object is being serialized to a file, for example, you can include a date/time property to indicate when the object was serialized. Upon deserialization you can extract this value and set the object's "age." The following constructor serializes an object differently if the serialization process's destination is a file:

```
Public Sub New (ByVal info As SerializationInfo, _  
               ByVal context As StreamingContext)  
    If context.State = StreamingContextStates.File Then  
        ' Serialize a date//time member  
    Else  
        ' Serialize all other fields  
    End If  
End Sub
```

Serializing SQL Server Data

Before we end this chapter, we'd like to discuss an interesting topic that combines SQL Server's support for XML and serialization. SQL Server can return the results of a query in XML format. The XML document describing the result of a query corresponds to the serialized version of a custom object and, if we can create a class that matches the schema of the XML document, we'll be able to deserialize SQL Server's XML response into an instance of the custom class. In this section we'll describe a technique for moving data out of SQL Server and into an instance of a custom class, without setting up DataAdapters and populating DataSets. The custom object will be an object that represents an order, including its header and its detail lines. The advantage of this approach, as compared to a straight ADO.NET approach based on DataSets, is that you don't have to worry about related tables and accessing related rows in DataTable objects. The custom object that represents the order is a business object that represents one of the entities you work with in your code, and you can manipulate it through its properties (read the values of an existing order, or create a new order). The application we'll use to demonstrate this technique is the NWOrders project, which reads existing orders from the Northwind database and creates new ones using a custom business object. The application's form is shown in Figure 3.4.





FIGURE 3.4 The NWOrders project

Summary

Serialization is one of the major new components of .NET and it's used heavily throughout the Framework. Persisting object is not new to .NET, but it's far more flexible and powerful than the PropertyBag object of VB6. Web Services—the most promising of the .NET technologies—use SOAP serialization to pass objects between client applications and web servers. SOAP and binary serialization is also used in remoting to pass objects between remote applications.

Binary serialization is very efficient and very compact, but it's limited within an application domain, or to applications that share the same classes. XML is verbose and not as fast, but it's a universal format. Although XML serialization is quite verbose and doesn't preserve type fidelity, it's ideal for passing hierarchical data between layers of an application, or between remote systems. A typical example of the type of integration between components you can achieve with XML was demonstrated in the last section of this chapter, where you saw how to retrieve XML data out of SQL Server and use it to populate instances of custom classes that match the structure of the data.

Another advantage of XML serialization is that you can control the serialization process of a given class with the use of attributes in the class that will be serialized. Finally, XML is the format in which DataSets are serialized and persisted to the client. In Chapter 19 you'll see an example of persisting DataSets in XML format at the client. The resulting XML document describes not only the data, but the changes made to the DataSet as well, and it can be used to submit the changes to the database.

Chapter 4

Leveraging Microsoft Office in Your Applications

EFFICIENTLY ACCESSING ONE APPLICATION'S features from within another application is a longstanding goal in the programming community. After all, if you can run a spell-check on documents *within* Word for Windows, why can't that same Word spell checker work just as effectively on documents held anywhere within the computer? Restricting such a useful utility to only documents within a particular application seems like an unreasonable limitation.

Indeed, the difficulty of communicating data between applications and sharing functionality was originally one of the primary justifications for object-oriented programming. The idea was that you should be able to write your programs in ways that permitted the features (objects) in your applications to be self-contained, reusable, and capable of being consumed by other objects (either inside or outside your application).

As usual, these noble goals have been less achievable in practice than they seemed in theory. There's still quite a bit of individuality and idiosyncrasy floating around in the computer world. The goals of application independence, not to mention platform independence, always seem to move just a bit out of reach as we approach them.

Nonetheless, Microsoft-designed products such as the Office suite and the .NET languages *do* offer a degree of interoperability and free communication between objects. How to understand the object models and consume methods within .NET and Office applications is the topic of this chapter. We'll focus on three of the most useful Office products—Word, Outlook, and Excel—but the techniques described for accessing these applications are applicable to other Office and Works applications. (*Applicable*, but requiring the usual fiddling around necessary to get the qualification and syntax correct for accessing members. Just because nearly all contemporary computer programs use a *print* method, for instance, that doesn't mean it's a universal usage. As you'll see in this chapter, Word and Excel use the term *printout*.)

Using Word's Features

Before you can access functionality from an Office application, you must reference its object library. You need a COM wrapper, as they say. For Word, you have to add a reference in your VB.NET project to—as you might guess—the Word object collection:

1. Start a new Windows-style VB.NET project.
2. Choose Project ➤ Add Reference.
3. Click the COM tab in the Add Reference dialog box, then locate the Microsoft Word 10.0 Object Library (your version number may not be 10.0).
4. Double-click this object library, then click OK to close the dialog box and add the reference.

NOTE At the start of the sections on Outlook and Excel below, you'll find instructions on referencing their objects.

The following sections describe how to use various handy features of Microsoft Word in your VB.NET applications.

Spell-Checking

To start things off, let's see how to check spelling in a VB.NET TextBox by borrowing that capability from Word's spell checker. We'll explore three different ways to accomplish this goal:

1. Sending TextBox text into a Word document, employing the spell-check dialog box to interact with the user to fix any spelling problems, then sending the fixed text back to the TextBox after spell-checking, via the Clipboard.
2. Feeding text directly into the Word spell-check utility and getting back an all or nothing (yes or no) answer as to whether there were any spelling errors in the string.
3. Same as number 2, except in this case getting back a list of misspelled words, and lists of alternative spelling suggestions for each misspelled word. With this approach, you can construct your own VB.NET version of the Word spell-check dialog.

To try the first example (Listing 4.1), start a new Windows-style VB.NET project and add a TextBox and a Button to Form1, then type this into the Button's Click event.

LISTING 4.1: SPELL-CHECKING A VB.NET TEXTBOX

```
Private Sub Button1_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles Button1.Click  
  
    Dim w As Object = New Word.Application 'instantiate a Word app  
    w.Visible = False 'don't confuse things by showing the app
```

```
Dim d As Object = w.Documents.Add 'create a word document  
Dim id As IDataObject 'will contain the results sent back fr
```

[Team Fly](#)

 Previous

Next 

Using Outlook Objects

One quick way to add an e-mail-reading feature to a VB.NET application is to employ the facilities in Outlook.

First, use VB.NET's Project ➤ Add Reference dialog box to add a reference to the COM library named *Microsoft Outlook 10.0 Object Library*. (Your version might not be 10.0.)

The Outlook object hierarchy exposes several "folder" objects: olFolderCalendar, olFolderContacts, olFolderDeletedItems, olFolderDrafts, olFolderInbox, olFolderJournal, olFolderNotes, olFolderOutbox, olFolderSentMail, olFolderTask. If you've worked with Outlook, you'll notice that these folders correspond to the various utilities and features available within Outlook.

To access the Outlook folders, you have to instantiate them indirectly by creating a MAPI message store (like API, only with the term *messaging* prepended).

Don't bother yourself with the wearisome nonsense that "explains" why you must first create a special Outlook namespace, then get a MAPI store. You'll add nothing to your understanding of programming by trying to follow the reasoning for this unique way of accessing Outlook's object library. Only Outlook does things this freaky way, so no use learning a process that occurs only once. You wouldn't study mammalian behavior by examining a platypus. Just copy the code in the examples below and you're home free.

WARNING Before you can test this example code, you must ensure that there's at least one e-mail message in your Outlook Inbox. Otherwise, the code will fail—I'm not bothering here to employ a **Try...Catch...End Try** failsafe error-trapping structure.

Put a TextBox, ListBox, and button on a form, then type Listing 4.13 into the Button's Click event.

LISTING 4.13: GETTING INBOX E-MAIL MESSAGES

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

    Dim o As New Outlook.Application
    Dim MAPI As Outlook.MAPIFolder
    Dim NSpace As Outlook.Namespace = o.GetNamespace("MAPI")

    MAPI = NSpace.GetDefaultFolder(Outlook.OlDefaultFolders.olFol
    Dim it As Outlook._Items = MAPI.Items

    Dim i As Integer, s As String
    Dim m As Outlook.MailItem

    m = it.Item(1)
    s = m.SenderName & " : " & m.Subject 'get the name and topic
```

```
ListBox1.Items.Add(s)  
TextBox1.Text = m.Body 'send the actual message to the TextBc
```

End Sub

[Team Fly](#)

 Previous

Next 

Accessing Excel

Use Project ➤ Add Reference, then click the COM tab and add the Microsoft Excel 10.0 Object Library (yours might be 9.0 or some other version number).

As you probably realize by now, if you've read this chapter, your first job when accessing Excel objects is to declare an application object:

```
Dim exl As New Excel.Application
```

Then you contact the primary unit of organization in Excel, the range object (which can be as small as a single cell or as large as a worksheet). The remaining sections describe how to use handy features of Microsoft Excel in your VB.NET programs.

Evaluating Math Expressions

One useful feature that Excel has to offer us VB.NET programmers is its ability to evaluate mathematical expressions. Given that VB.NET includes lots of math functions, such as SIN, COS, and so on (if you first `Imports System.Math`), why would you need to use Excel? For one thing, Excel allows you to submit a string for evaluation, thereby making it relatively simple to permit users to enter expressions into your application and have them evaluated.

Listing 4.14 shows how you can calculate a SIN in VB.NET, while Listing 4.15 is an example that uses Excel to evaluate the same expression.

LISTING 4.14: EVALUATING MATH EXPRESSIONS IN VB.NET

```
Imports System.Math

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

    Dim n As Double = 4.3444
    n = cos(n)

    MsgBox(n)

End Sub
```

LISTING 4.15: EVALUATING MATH EXPRESSIONS IN EXCEL

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

    Dim exl As New Excel.Application

    Dim n As Double
```


Summary

In this chapter you saw how to access Microsoft Office applications' features from VB.NET projects. You saw how to send data to and from Word, Outlook, and Excel. And you explored various ways to employ the major tools and utilities within Office applications.

First you saw how to reference, then instantiate, Word and exploit its spell-checker utility three different ways from within VB.NET. Then you saw how to send a fax from VB.NET via Word's faxing facilities. You learned how to load a document, and get statistics about documents, such as a word count. You saw how to suppress unwanted dialog box messages, and how to get a list of files from within directories and their subdirectories.

You explored Word further, seeing how to feed text into documents—including manipulation such as insertion and replacement—and also how to format and print text from within Word.

Then you learned to use Outlook to access incoming e-mail messages. Finally, you saw how to employ Excel to evaluate math expressions, print tabular data for reports, and format, calculate, and read or write .xls files.

Chapter 5

Understanding .NET Security

CONSIDER THE PARADOX IMPLICIT in this chapter: You are about to read details about security measures in Windows and .NET, but if you can read about it, how can it remain secure? Shouldn't security rest on *secrecy* and depend on the fact that people can't buy books describing precisely how it works?

Well, yes and no. *Somebody* has to have the keys to Fort Knox. It may as well be you, the trusted programmer, or trusted IT administrator.

In fact, there are multiple layers of security within today's computer systems and they generally work on an all-or-nothing premise: *all* the layers must grant permission to the agency (consuming caller or user) attempting to try anything potentially dangerous. By *dangerous* we usually mean any kind of file access (whether to read private data, to write and maybe add viruses to files, or to have the ability to reformat drives and so on) or access to the Registry, to peripherals such as printers, or to the security system itself (where they can fiddle around and make themselves administrators and fling the doors open).

Security features in .NET are extensive, comprehensive, and powerful. You should familiarize yourself with them because, as we all know, security is Topic A in many IT departments these days. Few programmers, though, have much experience with encryption and other security measures.

In this chapter, you'll learn about the various levels of Windows (generally role-based) and .NET (generally code-based) security, including aspects of "trust," the various kinds of permission management, and the interactions between role-based and code-based permissions. This subject is quite large, but this chapter is intended to provide you with an overview of the major tools at your disposal as you attempt to ensure the integrity of your .NET applications—prevent them from being breached, or from being misused to breach other resources.

NOTE *This chapter gets you well on your way down the long road to ensuring system security. For deeper coverage of the topic, see .NET Development Security Solutions by John Mueller (Sybex, 2003).*

Chapter 6 covers a different aspect of .NET security: encryption and hashing, highly effective tools for ensuring the integrity of transmitted messages and for protecting privacy.

Security: An Overview

Security features proliferate throughout today's computers. In fact, most anything larger than a single procedure can have some kind of security feature that can be adjusted by a user, an administrator, a programmer, or all three groups. As you'll see later in this chapter, programmers can even use .NET to specify security behaviors for single procedures.

There are all kinds of levels and varieties of security—and some of them conflict with each other, stepping on each other's toes. You find self-contained security feature sets in applications such as Internet Explorer, utilities, the operating system, networks, databases and database languages, servers, Internet applications, languages such as VB.NET and ASP.NET, IIS settings, and so on.

There's a more-the-merrier quality to current computer security efforts—you find locks and bolts, checkpoints and identity verifications all over the place. If you've ever lived in New York or another large city, you probably know someone whose idea of increasing security is to add yet another deadbolt to the 12 locks they already have on their apartment door. Doors in big city buildings are like Houdini's escape-proof suit—straps, chains, alarms, sliders, and what have you.

And if you've ever struggled to get your .NET prototype applications working with a database or SQL Server, for example, you've entered the security house of mirrors.

Obviously, security isn't something that is designed to be easily circumvented—by definition, security measures are supposed to be, if not obscure, at least somewhat difficult for the average user to understand and manipulate. You're not *supposed* to have a helpful message box pop up saying "You need to adjust your logon identity permission level before you can access this database. To make this adjustment, choose Start ➤ Programs ➤ Microsoft SQL Server ➤ Enterprise Manager. Expand the Security node, then click Logins to see the list of users who are permitted to log into SQL Server. If you have this permission, take the following step"

No, you have to dig around to figure out that in addition to your Windows role (the security group you belong to, as identified by your logon name), you have another, separate role to define with SQL Server. If you're told that you don't have permission to create a connection to a particular database, you have to get down and give yourself permission.

Beyond Windows and SQL Server, there are yet other layers. For example, when VB.NET's managed code is expected to work with SQL Server, then SQL Server's security apparatus comes to life and, possibly, denies access on this level. Perhaps you're running an *application* that doesn't have permission (or doesn't grant permission). Perhaps a particular file is set to read-only. The list goes on.

Some security settings are specified by the user, such as adjusting which macros Word allows to execute, or whether or not Outlook Express warns you about executable attachments to e-mail.

Other security settings are under the control of administrators, the IT professionals who look after the safety of workplace operations. Still other aspects of security are managed by developers and programmers who can specify various levels of access and permissions right within their applications' code.

Nearly all of the various types of access security, though, come down to one thing: Who *is* this user, and what exactly do they have permission to do? The answers to these questions fall mostly to what's called role-based security, and the key to role-based security is the administrator—the person or persons with total access to a computer or network, and the one who defines everyone else's role (or "level of trust," also known as permissions). Anyone who can figure out how to gain administrator status can run riot.

mobile code—networked, or web-based applications—can of course be more complex. You often don't know who's on the other end of an Internet connection, or what hard drive is being used as the server, or, most important, what methods are being executed against *your* local hard drive or network.

However, .NET insists that in all cases, *both* role-based and code-based security settings must be satisfied for a particular action to take place. For example, if you attempt to load a file into a .NET TextBox, several security settings are triggered and *all* must be satisfied before the file is loaded. The .NET application's identity is checked; is it from a trusted source? Does this application have permission (from code-access security settings) to read this file? And does this user have permission from the Windows security settings to read this directory and this particular file? If any of these questions are answered No, the file doesn't get into the TextBox.

This last question—Windows permissions—becomes impossible to answer when you're consuming a remote Web service, for example. Of course the author of the Web service response doesn't have permission to access your Windows machine at any level. That foreign person is unknown to your installation of Windows and isn't a member of any group known to your administrator.

Understanding Code-Access Security

One solution to communication with strangers is to keep them in the lobby and talk to them through an intercom, or if you're running a gas station, encase your clerks inside bullet-proof Plexiglas. In other words, fix it so you can communicate with strangers, but don't let them get next to you physically. Don't let them completely *in*. This, in essence, is the idea of "partial trust," the notion that you keep the stranger at a distance—close enough to talk to, but beyond the range of a knife or bullet.

Similarly, you can communicate with unknown Internet servers and other strangers by partially trusting them—letting them near, but not actually in, your system.

The CAS system has been developed to permit you to consume mobile executable code securely within .NET (or indeed other contexts). In fact, unless you specify otherwise, *any* executable coming in from the Internet is by default executed within this "partially trusted" context. Foreign, unrecognized executables are kept in the lobby by security, so to speak.

One meaning of the term *mobile code is distributed* code (code not local to an application on your machine, but rather coming into your machine from the Internet, an intranet, or modules distributed on separate servers). In other words, it's alien code that resides outside the local environment. As is usually the case, however, new computer terminology forks rapidly into more than a single meaning. *Mobile* is also being used these days to describe portable devices, specifically PDAs and cell phones (see Chapter 22 for details on programming for these mobile devices).

Scripting was one effort in the past to permit harmless mobile code to execute safely on your machine. The idea was: We'll take a language like VB and strip it of any methods that can manipulate the hard drive, the Registry, or other sensitive resources. Then, with this new "VBScript," people can trust that it's unable to do damage. Alas, this solution, like verification and other initiatives, was only partially successful. After all, hackers have learned how to embed executables in strings, and other techniques that make scripts potentially just as damaging as traditional executables.

Verification slows things down. One type of authentication surprises users with a dialog box asking them if they trust this Authenticated site. This not only halts execution, it throws the responsibility for virus attacks onto the user—many of whom are not equipped to respond usefully to the

enforces no permissions when native code executes, and SRP enforces none of its permissions when managed code executes.

If you've never worked with SRP, you can quickly take a look at its capabilities (limited capabilities, in fact, when compared to the greater range of CAS options). In Control Panel, open the Administrative Tools icon and choose Local Security Policy. In the left pane of the Local Security Settings dialog box, open the Software Restriction Policies node and look around. You can adjust these policies here for this individual machine. For more details on using this technology to block rogue ActiveX controls, virii, tainted scripting, and other dangers from unmanaged, alien code execution, see:

<http://www.microsoft.com/windowsxp/pro/techinfo/administration/restrictionpolicies/default.asp>

Managing .NET Security Policy

Now that you've got an overview of the layers of Windows security and how they interact with .NET security features, it's time to go down into another dungeon and see how to manage .NET security itself.

When you fire up an XP or Windows 2000 machine for the first time, it has a generally predictable set of security policies—the defaults that Microsoft thinks make sense for the average user. Here's an overview of the default settings for XP machines:

- Code from within the Internet zone (as Windows calls Internet locations) has a restricted permission level. The default setting for this zone is Medium (see Table 5.1). No code originating within the Internet is allowed to execute. If your computer or network requires that this policy be loosened, the administrator must explicitly adjust permissions. Run Internet Explorer, then choose Tools ➤ Internet Options and click the Security tab in the Internet Options dialog box. Move the slider to see the various options, and make any adjustments you want by clicking the Custom Level button.
- Code from the restricted sites zone is similarly forbidden from execution. The default setting for this zone is High.
- Code in the trusted sites zone has fairly limited permissions. The default setting is basically Low, but Java permissions are adjusted to Medium and unsigned ActiveX controls can be downloaded.
- Code from your local network (intranet) has certain default capabilities (it can read, but not write, environment variables), but it is forbidden access to the security system, the Registry, and so on. The intranet zone includes network paths and any sites that are bypassed by the proxy server. The default setting for this zone is Medium-Low.
- Code executed from the My Computer zone, however, is unaffected by settings adjustable from within Internet Explorer.



FIGURE 5.6 Use this Security Adjustment Wizard for specific, emergency shutdowns.

Click Next and choose Local Intranet. Move the slider to No Trust. Now let's see Nicky use his computer to make more mischief. Of course, if you find him using other people's computers, you'll want to adjust his user policies—and you might just want to get into Windows role-based settings and tie his hands there as well.

Before concluding this chapter with some suggestions that programmers can use to improve the security of the applications they write, the following is one final caution for administrators:

Avoid, of course, opening up anyone's quivering, vulnerable hard drive to TotalTrust levels in the Internet zone (or indeed other zones). You don't want to invite trouble, and Full Trust is just asking for it because no .NET Framework security tests will be conducted against executing code from the Internet. Operating system settings will remain in effect, but if any of them permit trust beyond what you'd allow for trusted local execution—beware. Full Trust is a broad and dangerous permission. If you're tempted to use this setting, consider instead modifying the permissions individually using the Trust Assembly Wizard. With that tool you can more rationally fine-tune permissions. Don't simply blow open all doors by using Full Trust.

Programming for Security

Programmers don't ordinarily consider themselves on the front line of security. They usually assume that if they provide relatively bug-free code, they are doing their job. Programmers make the tools, and it's up to security enforcers (normally IT administrators in offices throughout the land) to ensure that those tools are used for their intended purposes.

However, .NET offers you, the programmers, the opportunity to take steps to create safer code. You've already seen some ways to write code that contains some security elements earlier in this chapter, and ways to employ CAS to increase the safety of your programs. Let's conclude this chapter with an overview of techniques available to programmers working with .NET security facilities.

If your .NET application doesn't get called by other code ("consumed," as they say), you can probably relax and not worry about the security issues. After all, .NET itself automatically demands permissions for many kinds of sensitive behaviors, and performs a code-access stack walk to throw exceptions as necessary. You can rest on CAS for many Windows-based applications.

```
' But remember and beware that any Shared
' methods in this class can bypass instantiation
' so in the following case, if you must use
' Shared, you have to repeat the permission test:
Public Shared Sub ReadAFile()
    Dim p As New FileIOPermission(PermissionState.Unrestricted)
    p.Demand()
End Sub

'The rest of the (not Shared) methods in
'this class don't have to test security--
'they won't even exist if the above constructor test fails.
Public Function SaveFile() As String
    ' do some I//O here
End Function

End Class
```

Use the Demand method, as illustrated in this code, to make certain that callers are allowed to access something (in this example, files). Here, before allowing this class to be instantiated, you demand a security check. The entire call stack is checked and all must have permission. If there is no security exception thrown, then the Demand is met.

You can use this security check class to test individual methods within the class (as illustrated by the SaveFile method in Listing 5.2 above), or you could have this class generate a special key that the caller can use during the entire session with your application.

***TIP** For simplicity I used FileIOPermission in the example in Listing 5.2; however, note that the .NET security system automatically demands File I/O permissions (and other, similar sensitive resource permissions). You don't typically need to write special code for this kind of thing. However, you can use these techniques to provide additional protection within database access procedures and other situations.*

Summary

In this chapter, you saw the ways that a programmer can address security issues to prevent hackers from breaching a system via your application, from using your application to access sensitive resources, and from other kinds of attack.

You saw that application security is divided into two primary levels: role-based (derived from the user logon) and code-based (derived from assertions or denials made within .NET code itself).

You saw how Windows built-in permissions groups are accessed and what they mean. Then code-access security (CAS) was examined, and how it interacts with role-based security features. You worked with the Framework Configuration tool, and saw how to employ various of its features to specify how .NET security is enforced. Finally, you explored some of the ways that you can protect consumed code such as Web services by setting up a permissions gateway through which the caller(s) must pass.

[Team Fly](#)

 Previous

Next 

Chapter 6

Encryption, Hashing, and Creating Keys

THE.NET ENCRYPTION FEATURES are among the most useful of the framework classes, but are rarely mentioned in books and articles. There's nothing terribly difficult about employing these classes, but perhaps there are a couple of reasons that most authors avoid this topic. First, many people are only vaguely familiar with the concepts underlying cryptography, and some of *those* concepts can be indeed complex. Second, the best word to describe the current state of affairs in computer security is probably *havoc*.

Computer security divides into two primary categories: safety (protection from attack), as described in the previous chapter, and privacy (concealing information), which is the topic of this chapter.

Fortunately, there are extremely simple solutions to both of these security dangers. If you are concerned that a virus might erase your hard drive or otherwise mess up your machine, simply back up your data frequently (and also make use of the System Restore feature in XP in case the virus goes after the Registry and other key files, as some do).

If you are concerned that someone might read your private files, simply encrypt them.

All too often, however, these simple security measures are not practical. In many business situations, the majority of employees are incapable of managing their own backup or encryption needs. Either the IT department has to intervene, or these processes must be in some way automated for the ordinary user.

In this chapter you'll see how to use the .NET encryption classes to programmatically encrypt, decrypt, and manage keys. This can provide the foundation for writing applications that automate the job of encrypting and decrypting files. You can also use these techniques to build encryption features into your own programs.

The Main Problem

The primary problem when enforcing workplace security policies is the creation and management of passwords. A user types in a character-based secret string that can (and should) contain digits as well. Then that password is usually transformed into an all-digit key that is used by the computer to encrypt or decrypt a file. In public key encryption systems, random keys are

Just send the hash value along with the file and the recipient can hash the file on their end to see if the hash values match, demonstrating that the file has not been altered during transmission.

Encrypting

The goal of encryption is to rearrange information so that it makes no sense to an intruder. *Rearrange*, not destroy. You don't want to so completely disturb the original information (the *plaintext*) that it is impossible to restore. You don't want to reduce the data to a fuming wreck.

However, precisely because the encrypted information (the *ciphertext*) is restorable, the wrong people—the intruder—can potentially restore it, read it, and make use of it. *Intruder* or *Eve* are the traditional names for people who intercept messages—*Eve* for *evesdropper*.

DES is the most popular strong encryption in use today to encrypt large amounts of data. Everything from money wire transfers to secret government communications are transmitted after having been encrypted via DES.

The government went to IBM in the early seventies and asked them to come up with an encryption standard for government and business communications. The government wanted the system to be computer-based and impossible to break, and they got their wish in 1976. Ever since, it's been the standard. Some observers say that DES has been cracked, but others disagree. In any case, it would require tremendous multi-processing power—many tens of thousands of personal computers working in tandem—to hope to crack a DES encrypted message. (Some experts suggest that the government has such power, but prefers to keep DES the standard because they want to be able to read messages and keep track of things.)

If you feel that your information is likely to draw attention from the government or 90,000 personal computer users who will gang together to focus on your secrets, .NET offers even stronger encryption functions. You might be particularly interested in the asymmetric public key system (RSA) described at the end of this chapter. The problem with asymmetric systems is that they are less efficient, slower. Some people advocate dividing the job into two processes: Using asymmetric encryption to transmit keys (which are short, compared to the size of most messages), then using DES to encrypt the messages.

Or, if you just want a beefier version of DES, .NET offers a couple of other algorithms, including TripleDES. How much slower is TripleDES than DES? Three times. But unless your messages are huge or your computer is slow, you probably won't be bothered by the speed issue.

There are many dozens of ways to encrypt files in .NET. Listing 6.3 illustrates one way to encrypt and decrypt a file using the DES algorithm.

LISTING 6.3: ENCRYPTION AND DECRYPTION A FILE WITH DES

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

```
        encrypt()  
        decrypt()  
End Sub  
  
Public Sub encrypt()
```

[Team Fly](#)

 Previous

Next 

In this code, you first create a key and an IV, then create a byte array (barray) holding the plaintext. You use the ComputeHash method to get the hash value of the plaintext. (Notice that these techniques require lots of byte arrays—so far, you've used four of them.) Then you use a cryptostream object to encrypt and save the results (the ciphertext) to a file, and then to append the encrypted hash values to the same file. Both your encryption and decryption procedures know how many bytes to remove from the end of the file because the hash value size (in bytes) is available from the SHA1 hashing object. For example, the decryption routine reads this value from the SHA1's HashSize property:

```
Dim hashSize As Integer = sha1.HashSize / 8
```

The results reported from this property are in bits (who knows why?) so you have to divide by 8 to get the actual number of bytes. The answer for SHA1 is 20 bytes.

The Decrypt procedure mostly follows the same steps as the encryption procedure. First you define a key and vector, then you open the file into a filestream and decrypt it via a cryptostream, and finally you are ready to test the hash. Recall that the hash values in the encrypting procedure were calculated only on the plaintext message, not on the entire contents sent to the file. You cannot get into the house of mirrors that would be created were you to try to get a hash value of your plaintext+hashvalue.

So, in the decryption procedure you create a byte array (sArray) that holds the entire file contents (ciphertext+hashvalue). Then you extract the hash value from the message portion of (sArray.Length - hashSize). In other words, you subtract the length of the hash value (hashSize) from the size of the byte array.

Next you extract the hash value that was appended to the file, and store those 20 bytes into the byte array messageHashValue. Finally, you use a loop to compare each byte in the message's hash value (messageHashValue) to each byte in the hash value computed in the decrypt procedure (hash Value). If any of the 20 bytes don't match, you alert the user that tampering occurred and that the ciphertext file's integrity has been compromised.

Asymmetrical Encryption

It's slower, but stronger. Asymmetrical encryption technology was illegal only a few years ago. The government felt that the bad guys—mobsters, dealers, wheeler dealers, lounge lizards, Eurotrash, and whatnot—would have a way of communicating that the FBI and others couldn't monitor. The problem with these laws was the usual one: laws don't deter lawbreakers.

The law against strong encryption was withdrawn. So now you can go ahead and use the most advanced encryption, asymmetrical algorithms. They're *asymmetrical* because the method used to encrypt is different from the method used to decrypt. It's a little more cumbersome than symmetrical systems, though, and slower. Because it's slower, you quite frequently find people using an asymmetric system such as RSA to encrypt the keys, but encrypting the actual plaintext message with a faster symmetrical system such as DES.

RSA and other asymmetric systems allow quite a bit of information to be public. They're sometimes even called *public key* systems. Three elements are permitted to be viewed by everyone, including any intruders: the enciphering process itself (the algorithm used to encipher and decipher), the ciphertext message, *and a key*. With all that information, Eve should be able to figure out the plaintext, don't you imagine? Guess again.

[Team Fly](#)

 Previous

Next 

In a real-life situation, there's no big problem sending the public key or the ciphertext—capturing them would do an intruder no good, so you don't have to be concerned about their security. However, the potential weakness in the whole asymmetric system is the public/private key pair string that the decryptor (recipient) must somehow protect from prying eyes.

In a temporary session, the recipient can just generate a public/private key pair, and send the public part to the encryptor. Then the message can be encrypted, sent to the recipient, deciphered, and all the keys thrown away. But in other situations, keys are used repeatedly. Perhaps you want to use RSA to encrypt some files and keep those files for future reference. You must then also keep the private key that decrypts them.

Or perhaps a set of everyone's public keys is published in a list and given to everyone in the office. Maybe it's inconvenient to change these keys more than every month or so. When public keys are reused for more than a single session, each recipient must retain the private key that works with their public key. Actually, *retain* is probably not the right word; *conceal* would be more like it. If a private key isn't kept totally private, the game is over.

If you write applications that employ RSA and you don't want to limit communications to short sessions, you'll want to add some code to securely persist the private keys. Alas, there is no feature in .NET that explicitly solves this problem, but you can work with key containers available via the CryptoAPI.

Summary

In this chapter you entered the secret and, to me at least, fascinating world of hidden messaging—encryption, the effort to disguise the meaning of text.

It's been thousands of years in the making, but today's cryptographic schemes (several of the best are available in .NET) no longer rely on the various and fallible historical tricks. A lord would send a hunter carrying a string of dead rabbits to the next castle in the Middle Ages. One of those rabbits had a message in its stomach.

In ancient Greece they shaved a guy's head, wrote a message on his skull, then waited for his hair to grow out before sending him on his way.

Le Roi-Soleil's patsies and minions wrote long letters to each other using lemon juice, which dries invisibly but can be restored by holding the paper over a candle. These messages were sent among the chateaux, and as an additional precaution, only every 12th word contained the true message. Of course, several courtiers lost their heads when they couldn't explain why they were sending blank pages to each other. Twits.

You needn't resort to these ineffectual and messy tactics. As you saw in this chapter, you have at your command some of today's best cryptographic power tools. Tap into the .NET Framework's security features and use DES, TripleDES, or ramp up to full RSA protection. It's more than doubtful that your secrets will be revealed if you hide them inside this technology.

[Team Fly](#)

 Previous

Next 

Chapter 7

Advanced Printing

AMONG THE MOST VISIBLE features introduced to VB.NET are the improved printing and print previewing mechanisms of .NET. You'll find it very easy to write your own printing routines, and the same code will produce both printouts and previews. To access the printing capabilities of .NET, you must use a few new controls. There's no longer a Printer object; you must use one of the controls that facilitate printing, and we examine these special controls in depth in the chapter. These controls can't be used to build interfaces; they simply expose the printing functionality of .NET to your application.

While printing has gotten both simpler and more powerful in VB.NET, the Windows controls we use to build our interfaces don't support printing. None of the controls that come with .NET provide a Print method, not even the TextBox control. Most developers will sooner or later face the problem of generating simple (or not so simple) printouts for their applications, and they'll have to write their own printing code. The other alternative is to buy a third-party control that supports printing, which is the suggested course of action if you need to print formatted text, but most developers will be handling simple printing tasks.

This chapter doesn't contain only advanced printing topics. It starts with an overview of the printing process in the .NET Framework (a process that's entirely different from the equivalent VB6 process) and it also covers simple topics such as printing text. The reason we've included this seemingly trivial topic is that we haven't found a reliable tool for printing text. Even the TextBox control doesn't provide a Print method, so we felt that a solid explanation of the process of printing text is in order. And as you will see in the corresponding section, printing text isn't as trivial as you may have thought. To make the sample code a little more useful, we've added a Print method to the TextBox control.

Another very common task in business applications is the printing of tabular data. We've decided to demonstrate this topic by creating a class that can print the contents of a ListView control.

Printing in .NET

The basic printing component in .NET is the PrintDocument control. To send something to the printer, you must first add an instance of the PrintDocument control to the project. This control is invisible at runtime and its icon appears on the Components tray at design time. To initiate the

***NOTE** You can use the `SetClip` method of the `Graphics` object that represents the page to impose the margins. This method prohibits printing outside a specified rectangle, and you can use it to make sure that all graphics elements that fall outside this rectangle are clipped. In most cases, however, we write code to arrange the graphic elements on the page taking into consideration the margins. When we print text, for example, we write code to break the lines when the text reaches the right margin, or start a new page when the text reaches the bottom of the page.*

When the event handler exits, the appropriate graphics commands are sent to the printer and the page is actually printed. If you need to print additional pages, you set the `e.HasMorePages` property to `True` just before you exit the event handler. This will fire another `PrintPage` event. The same process will repeat until all the pages have been printed. When you're finished, you set the `e.HasMorePages` property to `False`, and no more `PrintPage` events will be fired. The default value of this property is `False`, so you need not set it when you're done printing.

Printer and Page Properties

One of the most common tasks in writing code to generate printouts is to retrieve the settings of the current printer and page, as they were specified by the user on the `PageSetup` and `PrinterSetup` dialog boxes. The properties specified on these two dialog boxes are reported to your application through the `PrinterSettings` and `PageSettings` objects. The `PageSettings` object is a property of the `PrintPageEventArgs` class, and you can access it through the `e` argument of the `PrintPage` event handler. The `DefaultPageSettings` property of the `PrintDocument` object is also a `PageSettings` object.

The `PrinterSettings` object is a property of the `PrintDocument` object, as well as a property of the `PageSetupDialog` and `PrintDialog` controls. Finally, one of the properties exposed by the `PageSettings` object is the `PrinterSettings` object. These two objects provide all the information you may need about the selected printer and the current page through the properties listed next.

The `PageSettings` Object

The `PageSettings` object exposes the following properties, which you can use to retrieve the properties of the current page.

Bounds Returns a `Rectangle` object that represents the current page. Its dimensions are expressed in hundredths of an inch. The `PageSettings.Bounds` property is equivalent to the `MarginBounds` property of the `e` argument of the `PrintPage` event handler. The `Bounds` property doesn't take into consideration the margins specified by the user on the `PageSetup` dialog. For a letter-size page the dimensions of this rectangle are 850×1100, and for an A4 page they are 827×1169.

Color Returns a `True/False` value indicating whether the current page can print in

color. You can set this property to determine whether the page should be printed in color or not.

Landscape Returns a True/False value indicating whether the page should be printed in landscape or portrait orientation. Use this property to find out the orientation specified by the user on the PageSetup dialog; setting this property won't affect the printout, because you still have to provide the appropriate code to print in landscape orientation (i.e., swap the page's width and height).

Margins Returns a Margins object, which exposes the user-specified margins as properties (Top, Left, Right, and Bottom). The properties of the Margins object are expressed in hundredths of an inch.

IsPlotter A True/False value indicating whether the printer is a plotter.

IsValid A True/False value indicating whether the PrinterName corresponds to a valid printer.

LandscapeAngle Returns an angle, in degrees, by which the portrait orientation must be rotated to produce the landscape orientation.

MaximumCopies Returns the maximum number of copies that the printer allows you to print at a time.

MaximumPage, MinimumPage Two properties that return or set the largest and smallest values the FromPage and ToPage properties can have. Set these two properties if you want users to select a range of pages on the PageSetup dialog box.

PaperSizes Returns all the paper sizes that are supported by the selected printer.

PaperSources Returns all the paper source trays on the selected printer.

PrinterName Returns or sets the name of the printer to use.

PrinterResolutions Returns all the resolutions that are supported by the selected printer. This PrinterResolutions property is a collection of PrinterResolution objects and its members are read-only.

PrintRange Determines the options available to the user on the Page Setup dialog box for selecting the range of pages to be printed. This property can be set to one of the members of the PrintPage enumeration (AllPages, Selection, or SomePages) and it determines which of the page selection options will be enabled on the Page Setup dialog box. Read the value of this property to find out the type of selection made by the user on the Page Setup dialog box. If the user has selected a range of pages, use the FromPage/ToPage properties to find out the numbers of the starting and ending pages.

SupportsColor Returns a True/False value indicating whether the selected printer supports color printing.

CreateMeasurementGraphics Returns a Graphics object that represents the page's drawing surface. We use this object to calculate the dimensions of text when rendered on the printer with a specific font.

The Printing Dialog Boxes

In addition to the PrintDocument control, there are three more printing controls, which are visible at runtime as dialog boxes: the Page Setup dialog box, the Print dialog box, and the Print Preview dialog box. The PageSetupDialog control displays the Page Setup dialog box, shown in Figure 7.2, which allows users to set up the page (its orientation and margins). This dialog box returns the current page settings in a PageSettings object. The settings specified by the user on the Page Setup dialog must be taken into consideration by your application to produce a printout limited within the page's margins, with the proper orientation, and so on. As you can see, there aren't many parameters to set on this dialog box, but you should display it and take into account the settings specified by the user.

Page Layout and Printing

The process of printing is identical to displaying graphics on a Form or PictureBox control. You can use any of the drawing methods of the Graphics object that represents the page. It's your responsibility to place the graphics elements on the page and determine when the current page has been filled and start printing a new page.

The origin of the page is its upper-left corner; the coordinates of this point are (0, 0). The origin of the printed element is the upper-left corner, too. If you print a string at coordinates (0, 0) it will be printed just inside the page, but it will be printed in its entirety. If you print the same string at the lower-left or lower-right corner of the page, nothing will appear on the printout.

The default unit of the page is a hundredth of an inch (there are 100 units in an inch). A lettersized page's dimensions are 850×1100. Every professional-looking printout has a respectable margin on all four edges. Printouts that cover the entire page look very odd—you've probably never generated such a printout. You can change the default units by setting the PageUnit property of the Graphics object to one of the members of the System.Drawing.GraphicsUnit enumeration. Among the members of this enumeration are Inch, Millimeter, Point, Pixel, Display (1/75th of an inch), and Document (1/300th of an inch). You can also use your own units by setting the PageUnit property to World.

However, not all printers can cover the entire page. There's a small margin that laser printers ignore, a very small margin compared to the user-specified margin. The printer's margin is usually a tenth of an inch. In most cases, we don't care about this margin, because the user-specified margin is much larger. However, if you plan to create printouts that cover the entire page, you must take into consideration the printer margin. The printer margin will cause a very disconcerting problem, namely a discrepancy between the preview and the actual printout. The monitor has no such margin, so you can preview graphics elements very near the edges of the page. When the same printout is sent to the printer, the printer margin may affect the appearance of the printout—it will not be identical to the page's preview on the monitor. Let's consider the printout of a rectangle that fills the entire page. If you preview the printout you'll see a rectangle that fills the page as expected. If you send the same document to a laser printer, the rectangle's origin will be displaced by a tenth of an inch (or so) from the upper-left corner of the page. The lower-right corner of the page will end up outside the page. If you print a rectangle that fills the printable area of the page (that is, the entire page excluding the user-specified margins), the rectangle will be the same, both on the preview pane and the printed page.

This behavior is caused by the fact that any shape whose origin falls within the printer margin is displaced slightly. If the element's origin is within the printable area of the page, the element is not displaced. To handle the printer margin, you can use the RenderingOrigin property of the Graphics object that represents the page. The RenderingOrigin property exposes the X and Y properties, which are the coordinates of the top-left point that can be printed on the page.

***NOTE** The `RenderingOrigin` property of the `Graphics` object is new to version 1.1 of the .NET Framework. If you're using version 1.0 of the Framework, you can't use this property.*

The DrawString and MeasureString Methods

You can use any of the `Graphics` object's methods to draw shapes, but for business applications the method you'll be using the most is the `DrawString` method, which renders strings in a specified font with a specified brush on the printer's page. Printing text is not a trivial operation, as you will see

```
Else
    strFormat.Alignment = StringAlignment.Center
    e.Graphics.DrawString("Print Mode", tFont, tBrush, _
        New RectangleF(e.PageBounds.X, 50, _
            e.PageBounds.Width, 100), strFormat)
End If
```

The last three statements in the `PrintPage` event handler print three rectangles: a red rectangle that delimits the printout's margins, a yellow rectangle around the page (from the page's upper-left corner to the page's lower-right corner), and a gray rectangle around the page's printable area, which is the entire page minus the printer margins, given by the `RenderingOrigin` property. The yellow rectangle doesn't take into consideration the areas of the page that can't be printed and will not be visible in the preview pane (it will be drawn at the very edge of the page). If you print the page on a color printer, you'll see that the yellow rectangle doesn't start at the upper-left corner of the page and is slightly smaller than the page. The three rectangles are printed with the following statements:

```
e.Graphics.DrawRectangle(New Pen(Color.Red, 3), _
    New Rectangle(e.MarginBounds.X, e.MarginBounds.Y, _
        e.MarginBounds.Width, e.MarginBounds.Height))
e.Graphics.DrawRectangle(New Pen(Color.Yellow, 3), _
    New Rectangle(0, 0, e.PageBounds.Width, _
        e.PageBounds.Height))
e.Graphics.DrawRectangle(New Pen(Color.Gray, 3), _
    New Rectangle(e.Graphics.RenderingOrigin.X, _
        e.Graphics.RenderingOrigin.Y, _
        e.PageSettings.PaperSize.Width - e.Graphics.RenderingOrigin.X, _
        e.PageSettings.PaperSize.Height - e.Graphics.RenderingOrigin.Y))
```

The `X` and `Y` properties of the `RenderingOrigin` object are the sizes of the two non-printable bands at the upper-left corner of the page. There are two equivalent bands at the page's lower-right corner as well.

You've seen examples of printing simple graphics elements, such as text and rectangles, how to take into consideration the page's geometry, and how to control the appearance of the graphics elements on the page, especially the appearance of text. It's time to look at a few more practical and interesting examples.

Printing Plain Text

Our first real-world example is a `Print` method for the `TextBox` control. We're actually wondering, What good is a `TextBox` control without a method to print its contents? A procedure that prints a text segment should be fairly simple, but it's not. As you will see, there are various parameters you must take into consideration, and a robust text-printing mechanism is a must-have tool for a developer. Most professional developers will purchase a third-party tool that can generate elaborate printouts, but you may find a few good uses for a simple text-printing tool.

The TextBox control uses a single font for its text, which simplifies our code immensely. To demonstrate how to print text with the Framework's printing controls, we'll create an enhanced TextBox control with a Print and a Preview method. If you were asked to suggest an enhancement to the TextBox

[Team Ely](#)

 Previous

Next 

Printing Tabular Data

In this section we're going to build another custom control by adding printing capabilities to the ListView control. The ListView control is a flexible tool for displaying tabular data, but like all other built-in controls doesn't provide printing capabilities. Another limitation of the ListView control is that it can't break its contents into multiple lines when displaying data in detail mode (which is the most common mode in typical applications). Figure 7.8 shows the test form of the application and a Print Preview window with the control's data. The control's data will be printed as shown on the preview window. Notice that the long lines that don't fit in the control's cells are printed on multiple lines.



FIGURE 7.8 Printing tabular data with an enhanced version of the PRNListView control.

The PRNListView control exposes a single method, the Print method. The code picks up the data as well as the formatting information from the control itself and prepares the printout, which is displayed in the Preview dialog box first. The user can send the output to the printer by clicking the Print button on the dialog box's toolbar.

In addition to the Print method, the control exposes a number of properties that allow you to customize the appearance of the report. All of the enhanced control's property names start with the prefix "Print"; they are as follows:

PrintBorderColor The color of the border around each cell.

PrintColumnPadding The extra space between columns (expressed in pixels).

PrintMaxCellLines The maximum number of lines in each cell. This property prevents a cell with a lot of text from becoming too tall. Normally, the control fits the text in its cell by breaking it into multiple lines.

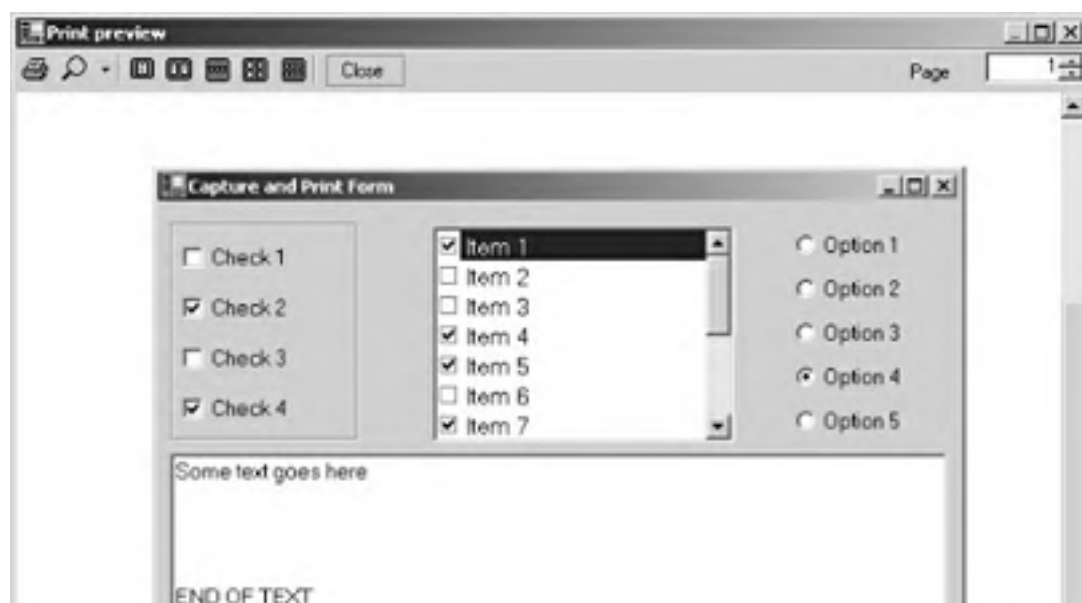
After printing a row, the program draws the horizontal line that separates it from the following row. The vertical lines are printed last, because we need to know the vertical coordinate of the last row on the page. The code isn't really complicated, but it's quite lengthy. We've inserted comments to explain its operation and we hope you'll find these comments useful in customizing the printing process.

A PrintScreen Utility

The last sample in this chapter is rather odd, in the sense that it uses API calls to capture the screen (or the current window) and print it. I've received requests from readers in the past about a simple technique to print a form. Obviously, many VB6 programmers used this technique in the past and they need a quick mechanism to generate printouts of the current window. Our recommendation is that you write code to generate proper printouts, as explained in the preceding sections of this chapter. If you think a screen-printing utility suits you, use the code of the PrintScreen sample application. Figure 7.9 shows the preview of a screen capture.

While printing the current form in VB6 was trivial, there's no simple mechanism in GDI+ to capture the screen. The closest we were able to come to a screen-printing utility was to use the BitBlt GDI32 function. This function can copy the bitmap from any device context onto any compatible device context. We'll use it to copy the entire screen (or the current window) onto an Image object, and then we will use the .NET Framework's printing mechanism to send the bitmap to the printer. The current window is not the active window on the desktop. By "current window" we mean the form that contains the code. There's no simple mechanism to capture a keystroke meant for another application, and our code can't print any other window (the "active" window) on the desktop.

Start a new project, the PrintScreen project, and paste the function declarations from Listing 7.5 at the form's level. These are three API functions that we'll call from within our VB.NET application.



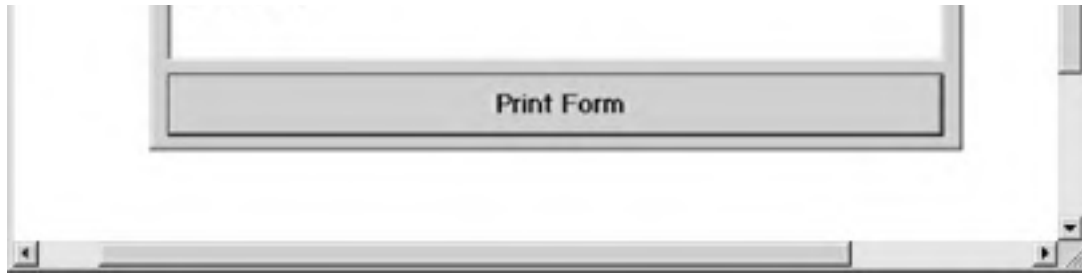


FIGURE 7.9 Printing the screen, or the current window

LISTING 7.7: THE PRINTBMP() SUBROUTINE

```
Private Sub PrintBMP()  
    PD = New Printing.PrintDocument  
    Dim PGSETUP As New PageSetupDialog  
    PGSETUP.PageSettings = PD.DefaultPageSettings  
    If PGSETUP.ShowDialog = DialogResult.OK Then  
        Dim PP As New PrintPreviewDialog  
        PP.Document = PD  
        PP.ShowDialog()  
    End If  
End Sub
```

In the PrintPage event handler we print the bitmap by calling the DrawImage method of the Graphics object. The code calculates the coordinates of the bitmap's upper-left corner so that it will be centered on the page (regardless of the margins) and then prints the bitmap, with the statements of Listing 7.8.

LISTING 7.8: PRINTING A BITMAP CENTERED ON THE PAGE

```
Private Sub PD_PrintPage(ByVal sender As Object, _  
    ByVal e As System.Drawing.Printing.PrintPageEventArgs) _  
    Handles PD.PrintPage  
    Dim img As Image = bmp  
    Dim X, Y As Integer  
    X = Math.Max(0, (e.PageSettings.Bounds.Width - img.Width) / 2)  
    Y = Math.Max(0, (e.PageSettings.Bounds.Height - img.Height) / 2)  
    e.Graphics.DrawImage(img, X, Y)  
End Sub
```

Notice that this time we retrieve the page's width and height from the PageSettings.Bounds property and we don't have to worry about the orientation of the page. In Listing 7.2 we used the DefaultPageSettings property of the PrintDocument object to extract the coordinates and dimensions of the printable area of the page, and we had to swap the width and height from within our code to account for landscape orientation.

Summary

In this chapter we thoroughly discussed the printing capabilities of .NET. We've demonstrated the printing process with practical examples, which you can use in your applications or extend by adding more features.

Chapter 8

Upon Reflection

REFLECTION IS ONE OF those new technologies that is described in various different ways by various experts (rather like Web services). It's new, at least, to Visual Basic programmers.

And we cannot pretend that reflection isn't just a *little* bizarre. Like recursion, it can involve a kind of self-consumption—an esoteric process. *Something* about reflection isn't quite normal.

At its most elemental level, reflection means finding out details about the contents of assemblies during runtime. And, like much in .NET, reflection has its antecedents in the C language, specifically the Runtime Type Information (RTTI) feature of C++.

Reflection permits you to learn the type, and members, of an object, while a program is running. But there's more: After you've discovered this information about objects, you can then *do* something with your knowledge. You can use reflection to execute the discovered methods, access discovered properties, pass parameters, and even generate, compile, and execute new code during runtime.

What Use Is It?

What good is all that? Some cool tricks become possible. For example, with reflection you can write code that will later (during runtime) consume objects that have not yet been designed. Or you can defer making the choice of which methods to invoke until runtime.

Sometimes you don't know the context or environment that will be in effect during runtime. One way to deal with this problem is to use reflection, thereby permitting your code to select an appropriate method during runtime. Think of this use of reflection as an advanced form of `Select...Case`.

Reflection can also be used to build custom object browsers, code-generators, sophisticated self-commenting code, utilities that examine and secure compiled executables, and advanced, dynamic debugging tools that facilitate runtime error trapping.

Understanding Types

The target of reflection is usually an entire assembly. The term *assembly* is new in VB.NET. It's what was traditionally called an *application*—a collection of related "types" and resources that do a particular job, such as word processing.

Seeing Reflections

Before getting into some additional aspects of reflection, try it in some code. You first must instantiate a Reflection. Assembly class and also access an existing assembly. There are several ways to do this (as is usually the case in .NET—you can choose from a variety of coding styles).

Accessing a Type

Probably the simplest example of reflection is accessing a single type (a class, in this case, and in most cases). Add a TextBox to a new VB.NET Windows style project, change the TextBox's MultiLine property to True, delete its default Text property, add a vertical scrollbar, then type Listing 8.1 in.

LISTING 8.1: A SIMPLE EXAMPLE OF REFLECTION

```
Imports System.Reflection

Public Class Form1

    Inherits System.Windows.Forms.Form

    ' Form designer code goes here

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

        Dim cr As String = ControlChars.CrLf

        Dim t As Type = GetType(puria)

        TextBox1.Text = _
        ''Here are the public constructors of the type " & _
        t.ToString & cr

        Dim cinfo As ConstructorInfo() = t.GetConstructors((BindingFl
        BindingFlags.Instance))

        Dim m As MemberInfo
        For Each m In cinfo
            TextBox1.Text &= m.ToString & cr
        Next m

        TextBox1.SelectionLength = 0 'turn off the default selection

    End Sub

End Class
```

```
        TextBox1.SelectionLength = 0

    End Sub

End Class

Public Class TestClass

    Private Field1 As String = "'5255"
    Private Field2 As String = "Info goes here"

End Class
```

Executing Discovered Code with CreateInstance and Invoke

It's well and good to be able to thoroughly examine the types, members, and parameters within an assembly. But reflection has two additional tricks up its sleeve. You can also execute reflected code during runtime (as illustrated by the following example).

In the example in Listing 8.8, you provide the user with a list of methods in a ListBox. The user clicks on that list, and you then ask the user to type in the correct parameters required by the method they chose. Finally, you execute the method.

This illustrates how you can write a program that explores an unknown assembly (unknown at least to you, the programmer, while writing your code). Your program explores the unknown assembly, displays its classes and members during runtime, permits a user to choose among the displayed items, tells them what parameters to pass, and then executes the code from within the reflected classes.

Listing 8.8 ties together several of the techniques introduced throughout in this chapter. Start a new VB.NET project and add a TextBox, a label, a ListBox, and a button. Then type this in:

LISTING 8.8: EXPLORING AN UNKNOWN ASSEMBLY

```
Imports System.Reflection

    Dim t As Type = GetType(TestClass)
    Dim obj As Object = Activator.CreateInstance(t)
    Dim mInfo As MethodInfo
    Dim l As Integer 'number of parameters
    Dim paramInfo() As ParameterInfo
    Dim cr As String = ControlChars.CrLf

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

However, imagine my excitement when I discovered the new Parse method. For a moment I thought my dream had come true and *this* was the built-in way to parse a string. No. Instead, they are using the term *parse* in a way it's never been used in English before. The Parse method doesn't parse (examine, divide into components); it casts.

Whatever. You can use it to change a string into various numeric data types.

In this example, I merely check for string or int32 types. However, in a real-world application you would include a Case to handle each possible data type that might be discovered as a parameter in an unknown assembly.

Each parameter's correct data type is created and added to a parameter array (p) in this loop:

```
For Each s In split
    'figure out parameter's variable type
    Select Case paramInfo(C).ParameterType.Name
        Case "String" 'case sensitive
            p(C) = split(C)
        Case "Int32"
            Dim NewInt As Integer = Integer.Parse(split(C)) 'turn string into integer
            p(C) = NewInt
    End Select
    C += 1
Next s
```

And, finally, you use the Invoke method to pass the array of parameters (p) to the instantiated method (obj):

```
mInfo.Invoke(obj, p)
```

If there are no parameters to pass to an instantiated method, your job is easier. Just pass Nothing:

```
mInfo.Invoke(obj, Nothing)
```

Emission

If you think reflection has a bit of a Twilight Zone quality to it, just wait until you find out about its ability to *emit* code. The `Reflection.Emit` namespace contains facilities that permit you to generate code at runtime. Yes, generate *code*.

It's not VB.NET source code; it's *Intermediate Language (IL)*, so the drawback is that you have to use a kind of opcode (assembly language) low-level programming. It's so specialized that few of us are likely to learn the language in order to emit. It features the usual assembly-language specificity, though in fact most of the instructions in this language should be familiar to you (the usual features

of any programming language: looping, branching, moving and editing data, and so on, along with low-level techniques such as stack management):

```
ILang.Emit(OpCodes.Ldarg_1) 'load 1 or 0 from the stack  
ILang.Emit(OpCodes.Ret)
```

The .NET compiler translates your VB.NET source code into MSIL (Microsoft intermediate language), a sort of Esperanto that can be converted to CPU-specific native code later, prior to execution in a specific environment. The .NET CLR (Common Language Runtime) includes JIT compilers for every supported environment. MSIL is converted to native code just in time to be executed.

In addition to MSIL code, emission (like the .NET compiler) also produces associated metadata—type definitions, signatures (parameter lists) for members, and so on. MSIL code, combined with its metadata, is sent to a file—a "portable executable" (PE) file.

Emission is, well, a rather *specialized* technique, to say the least. (I expect some of you may even consider it a bit twee.)

You can generate types during runtime using the Builder classes within the Emit namespace. These classes, including MethodBuilder and AssemblyBuilder, emit MSIL code.

If you are interested in code that generates code, you have to explain to people *why*. How would you use it? There are some security applications I can think of (whenever you move toward greater abstraction, you decrease the number of people who can figure out what you're doing). Also note that VB.NET itself uses code-generating-code in various ways. You find it in wizards, in ADO.NET (for example, to transform a DataSet into a serialized XML version), and in the text processing available via regular expressions, among other locations in the framework.

If this interests you, you can find many tools (such as the ILDasm, an MSIL Disassembler) and lots of documentation in the .NET help system and in the SDK. Look in C:\Program Files\Microsoft Visual Studio .NET 2003\SDK\v1.1\Tool Developers Guide\docs\StartDocs.htm. You'll find extensive documentation there covering the Common Language Infrastructure (CLI), and the huge IL it contains.

Summary

In this chapter you discovered what the technique of reflection does, and how it does it. You also saw how reflection interacts with security issues and how to manage the results sent back when reflection is used against various kinds of targets.

You saw how to filter reflections and how to access loaded assemblies. Also covered was the somewhat perplexing terminology currently in use to distinguish the various kinds of types (arrays, modules, enumerations, structures, interfaces, and value types). Beyond that, another level of abstraction and categorization was discussed (the differences between the terms *assembly*, *solution*, *project*, *module*, and *namespace*).

Finally, you learned about discovered code, CreateInstance, and Invoke. And the idea of code emission was discussed. If you suspect that reflection might be a technique of use in your programming, this chapter's example code and discussions should have provided you with a good launching pad to further exploration of this intriguing new technique.

[Team Ely](#)

 Previous

Next 

Chapter 9

Building Bug-Free and Robust Applications

YOUR BASIC TASK, As a developer, is to write functional, robust applications. To write functional applications, you must keep the interface as simple as possible, use your common sense, and listen to the users. If you take the users' comments into consideration while designing your application's interface, you will produce a functional application. You should also carefully examine similar applications; keep the good ideas and make sure you don't repeat the mistakes of others. The design of functional applications can't be taught. If you've been around in this field for a while, you already know that this is an acquired skill and there's no substitute for experience. Younger developers tend to make their applications more complicated than they should, simply because they think that users will appreciate a brilliant piece of code. It took most of is quite a while to learn how to "keep it simple."

Writing robust applications, on the other hand, is not as hard. While writing functional applications is an art, writing robust application is a technique and it can be taught. It takes a lot of code, but it's well within the average developer's skills. A robust application is one that will continue its operation under adverse conditions. The most typical such condition occurs when users supply the wrong data. Users will enter data that defy any logic and your code should be able to handle them. At the very least, the application shouldn't crash. Your application may discard some of the user-supplied data, but it shouldn't terminate without a good indication of what went wrong. If possible, you should give users a chance to correct their mistakes. At the very least, your application shouldn't terminate without giving users a chance to save their data.

In this chapter we'll discuss structured exception handlers, which allow you to write robust applications that execute gracefully even under unforeseen conditions. The goal is to write applications that can handle everything users throw at them, as well as cope with unexpected situations beyond the program's control. To handle user errors, we provide extra code that can handle situations that throw off the "regular" code ("regular" code being all the statements that process perfect data).

the Common Language Runtime Exceptions section of the Exception window. You can choose an exception in the upper window of the dialog box and specify how the Debugger will handle it in two frames near the bottom of the dialog box. The options in the upper frame determine how the debugger will react to an exception as soon as it's thrown and before your code is given a chance to handle it through the appropriate exception handler. The options in the lower frame determine how the debugger reacts when an unhandled exception is thrown. The options you set in the two frames affect the selected exception, and you can handle different types of exceptions differently. The option "Break into the debugger" breaks the execution of the program as if there were a break point on the statement that caused the exception. The option "Continue" allows the application to continue its execution. If the statement that threw the exception is in a structured exception handler, the handler will be activated. The "Use parent setting" option handles the exception as it would handle an exception of the parent category (if there is one). Notice that you can add your custom exceptions to the list of exceptions with the Add button.



FIGURE 9.3 The Exceptions dialog box

If an error occurs in an error handler's code, you can retrieve the statement that caused the initial error with the `InnerException` property of the `Exception` object.

Debugging Techniques

In addition to handling user mistakes, you must also handle your own mistakes. How many times did you write code that contains no syntax errors, executes fine, but doesn't produce the correct results? It's happened to everyone who has written some serious code. Your code contains logical errors, which you must identify and then fix. When you're dealing with logical errors, you must step back and reevaluate your algorithm. Make sure that you're using the proper steps to get to the desired result. The algorithm is usually correct, but in most cases it's not implemented correctly.

[Team Fly](#)

 Previous

Next 



FIGURE 9.8 The Call Stack window

Summary

In this chapter you learned how to write robust applications that can handle user errors or abnormal conditions that may never occur in the design phase. These conditions cause runtime exceptions, which you must handle from within your code by inserting the appropriate exception handlers. A professional-grade application should be robust, which means that you'll have to write more code to handle exceptions than to actually perform useful tasks.

You also learned the basics of the integrated debugging tools, which allow you to locate logic errors in your code and make sure that your application works correctly, even in the absence of exceptions. How many times were you absolutely convinced that your code was correct, but users discovered bugs in your code? It's been said that if builders built homes like programmers write code, the first woodpecker that came along would have destroyed our civilization. On the other hand, we can afford to experiment and make mistakes because our computers are so fast and there are so many tools to help us write better code.

All debugging tools are based on a simple premise: the code that fails should be executed one statement at a time and we should be able to examine the effect of each statement. We can view how each statement affects the variables in its scope, execute statements outside the application, and monitor the progress of the application.

Chapter 10

Deploying Windows Applications

ONE OF LAST PHASES in an application's development cycle is the deployment process. While developing, testing, and debugging an application, many developers suddenly realize that they must deploy their application to a number of workstations. If you don't think of the deployment process while you're developing your application, you may run into surprises when you attempt to install the application on another machine. As you code the application, you make changes to the development environment. You may install a peculiar font (an OCR font, for example), use icons from the folder in which they exist, install custom components on the development machine's *global assembly cache* (GAC), and so on. When the application is deployed to a target machine, it may not find a drive or folder that existed on the development machine, or a component that's not installed in the target machine's GAC.

You should always keep in mind that your application will be distributed to other people's workstations. If you're using icons, for example, place them first in a folder under the project's Bin folder and then use them. Distribute the folder with the icons along with the application's executable files, to make sure that your code will find the icons. In addition, you should decide on your deployment method early in the process and deploy the application to a production machine from time to time. Production machines are set up differently than development machines, and you'll be surprised how often an application that works as expected in the development environment misbehaves when installed on a production machine. The problem is usually simple to resolve (most often components that have been installed on the development machine, but not on the target machines), but you shouldn't postpone the deployment problems to the very end of the cycle.

In this chapter we're going to explore the various deployment techniques for .NET Windowsbased applications. There are two common deployment scenarios:

- ◆ Applications that will be distributed in a corporate environment
- ◆ Applications that will be distributed to the general public

Most developers write applications that will be used within their corporation, and we'll focus on a technique for deploying applications in a fairly controlled, trusted environment. You'll also learn how to create setup programs to distribute your .NET applications to the public.

Another related topic is that of upgrading applications that have already been installed on the target computer. Let's say you have written and deployed an application to a large number of users throughout your corporation. What happens when you need to update the application? Do you distribute a new setup program and ask your users to run it? It's inevitable that while some users will be running the new version, others will still be running the old one. If you upgrade the application (or some of its components) several times, it's certain that most versions of the application will be in use at your company. We will address the issue of upgrading an existing application in our discussion.

Installing the .NET Framework Runtime

The client computers to which you're going to deploy your .NET applications must have the .NET Framework runtime installed. If not, the application won't run. To install the .NET Framework on the target computer, you must run the Dotnetfx.exe setup program. You can obtain this file from Microsoft and deploy it with your applications. This file is an installer that contains the Common Language Runtime and .NET Framework class libraries necessary to run .NET Framework applications. You can find this file on the Visual Studio CDs (it's on the .NET Framework SDK CD in the \dotNETRedist directory), or download it from Microsoft at: <http://msdn.microsoft.com/library/default.asp?url=/downloads/list/netdevframework.asp> or at <http://www.windowsupdate.com>. (Keep in mind that both of these links may be invalid by the time you're reading this book, so you should search the MSDN site).

Installing the .NET Framework takes a few moments, but it's an unattended process and it will either install the Framework successfully or will fail and the original computer configuration will be restored. You can install the .NET Framework on Windows 98 computers and your applications will also work under this pre-.NET operating system. The next version of the Windows operating system will come with the .NET Framework preinstalled. This will simplify the deployment of .NET applications even more (ignoring the fact that we'll have to deal with updates in the .NET Framework itself).

To install the .NET Framework on a client computer, the user must log on with administrator privileges. This isn't usually the case in a corporate environment, so it's best to leave this task to the system administrator, who can install the .NET Framework on all client computers using the Systems Management Server. You can also install the .NET runtime files silently, along with your application. The process is described in detail in the documentation (search for the item Redistributing the .NET Framework).

In the following sections we're going to look at the deployment methods for Windows forms-based applications, starting with the XCopy method, which is as simple as copying the executable files from the development machine to the production workstations. This is a very basic deployment method that can be used with simple applications—and it's a seriously limited deployment method, because it doesn't allow you to perform custom actions, such as installing a shortcut on the user's desktop, or a new font on the target computer.

The deployment method we'll explore in detail is the Internet-based deployment, or no-touch deployment, which is ideal for corporate intranets. The application's files are copied to a virtual directory of the web server and users can run the application by pointing their browser to the application's URL.

The last deployment method is to create a Windows installer package, distribute it to the target machines, and ask users to run the Setup program, which will install the application and integrate it

with the user's environment (create shortcuts, add entries to the Registry, request product registration, and so on). Programs installed through Windows installer can later be removed through the Add Or Remove Programs snap-in of the Control Panel.

XCopy Deployment

Once the .NET Framework runtime has been installed on a client computer, you can deploy an application by copying its files to the client computer. Simple applications consist of just an EXE file. Large applications may contain DLL files with custom components. The good news is that you don't have to register the DLLs and worry about versions. The DLLs are copied along with the EXE and you can have different versions of the same DLL running side-by-side. Different versions of the same DLL reside in different folders, along with the version of the application that uses them. You can even install different versions of a DLL in the GAC, where applications look for a DLL if it's not in their path.

This type of deployment is called XCopy deployment, because the application is installed on the target machine by simply copying the files in the application's Bin folder (and any subfolders with custom files that may exist in this folder) to the target computer. No components are registered and no changes are made to the target computer's file system. To remove the application, you simply delete the folder and no trace of the application will remain on the computer—except perhaps for a shortcut the user may have created on the desktop. You can also install multiple versions of the same application, which can run side-by-side. Each application folder contains its own DLLs and dependencies and they won't interfere with one another. The fact that DLLs are treated as application files and need not be registered at the client computer may bring an end to the situation known in pre.NET days as DLL hell.

If a component is going to be used by multiple applications, we usually place it in the GAC. To install a DLL in the GAC, open a Command Prompt window, switch to the folder that contains the DLL, and execute the following statement:

```
cagutil -i component.dll
```

To uninstall the same component, call the cagutil with the -u argument. When using XCopy deployment, it's a good idea to avoid the GAC, because you must remember to install the new version of each component to each client's CAG. If you decide to create an installer package to distribute your application, you can automatically install components to the GAC from within the installation project.

As convenient as this method of deployment may sound, it's not flexible at all, and can't be used with anything but the simplest projects. You can't even create a shortcut on the target computer's desktop; users will have to do so manually. If your application needs to install a component at the GAC, or install a new font to the target computer, then you can't use this deployment method. You must resort to a setup project that will install the application to the target computer and perform custom actions.

XCopy deployment eliminates the update problems as well. To deploy a newer version on the client machine, just copy the new files over the existing ones. The next time the user on this client runs the application, they will see the new version of the application. Of course, if you copy the files to a different folder, both versions of the application will coexist on the client.

[Team Fly](#)

 Previous

Next 

Internet Deployment

This type of deployment is new to .NET and you'll find it extremely convenient if you're working for a company that uses an intranet. Internet deployment is also known as no-touch deployment, because you don't have to install the application on the target machines. Users can connect to a web server and download the application to their workstations, where it will be executed. It's also known as zero-install and zero-administration deployment, and we'll explain why immediately.

The problem with installers is that every time we make a change to the application, we must create a new installation project and redistribute the application to all clients. Deploying an application to hundreds of user desktops is a nightmare for system administrators, which explains the popularity of Web applications. Web applications run from a web server, and no components need be installed on the client computers. However, a browser-based application can't offer the rich user experience of Windows forms applications. A WebForm can't provide immediate feedback to user actions as Windows forms can, and WebForms make numerous trips to the server. With no-touch deployment, we can simply copy the files generated by the compiler in the application's Bin folder to a virtual folder on the company web server and be sure that all clients will see the latest version of the application. In short, no-touch deployment combines the best of Windows forms– and WebForms–based applications. The executables are downloaded to the client where they're executed, while no components are installed at the client.

Internet-based deployment eliminates the problem of distributing upgrades. You can simply replace the original executables on the web server with the newer ones, and the next time a user connects to the application, they will see the new version. If you're developing an application in a corporate environment, this type of deployment is your best option, because you can upgrade your application on all desktops with zero downtime. You'll never have to ask users to stop their applications and install a newer version. The worst-case scenario is that you may have to ask users to exit the application and restart it.

To run an application from a web server, users must start their browser and enter the URL of the application's main executable file on the server. Alternatively, you can create a simple web page with a hyperlink to your application and ask users to connect to this page's URL. If your company runs an intranet and users connect to a starting page every morning, you can place the hyperlink to this page.

Note that users need not start their browser to connect to an application deployed through a web server. They can create shortcuts to the URL of the application on the desktop and start the application by double-clicking this shortcut. When the shortcut is double-clicked, the browser's window comes up for a moment and then the application's form will appear.

With this type of deployment, the application's files are copied to the download cache of the client, from where they'll be executed. Every time the user starts the application, the CLR compares the hashcode of the application in the local cache to the hashcode of the application on the web server. If they're the same, the application is started from the cache. If not, the CLR

downloads the newer version from the web server to the cache and then executes it. As you will see, it's possible to download components from within the application, a technique that allows you to download components on a separate thread while the application is running. Practically speaking, you aren't going to use this type of deployment unless you know that all clients have a high-speed connection to the server. This means that new components won't take long to download to the client, so you expect users to wait for a few moments to download the newer version of an application (or component) from time to time.

The following code segment shows how to download the NoTouchDeployment project's EXE file to the client, then extract the application's main form and display it:

```
Dim appURL As String = 'http://localhost/NWEmployees/NoTouchDeployment.exe"  
Dim asm As [Assembly] = [Assembly].LoadFrom(appURL)  
Dim formType As Type = asm.GetType("NoTouchDeployment.Form1")  
Dim objForm As Object = Activator.CreateInstance(formType)  
Dim Form1 As Form = CType(objForm, Form)  
Form1.Show()
```

Notice that the name of the Assembly class is embedded in a pair of brackets, because "assembly" is a reserved word in VB. Strangely, it's not used, but it's a reserved word. The assembly could be an EXE or a DLL. What this short code segment demonstrates is how to start an application from within another application, even though none of the applications lives at the client.

Deploying with Windows Installer

The last option for deploying .NET applications is the most advanced one and involves the generation of a setup project, which users must run on the client machines to install the application. This is also the most professional method of deploying an application, and it's the only option for distributing an application to the general public. Using the Windows Installer we can create shortcuts on the user's desktop, add items to the user's Programs menu, provide custom dialog boxes to customize the installation process, and do a lot more. The setup program is a bootstrap application that opens an MSI package and installs the application and its components on the client computer, according to instructions embedded into the package at design-time.

Creating a simple Windows installer package with Visual Studio .NET is a straightforward process, because the setup project can be part of the same solution as the application for which the package is created. In earlier versions of Visual Studio, setup projects were created with a tool outside Visual Studio. Creating a flexible installation program for a large application may become quite a task, but at the very least you can design and test the setup project in the IDE of Visual Studio.

A Windows installer package is a database with all the data needed to install the application. The information stored in the database remains at the client, and you can run the setup program again to either repair or uninstall the application. Every application installed at the client computer with the Windows installer package is assigned an item in the Add Or Remove Programs snap-in; and this is how users repair or remove applications from their machines. Figure 10.10 shows the Add Or Remove Programs snap-in window after the installation of the NWOrder application. If you click the Support information hyperlink, you will see the SupportInfo window, which is also shown on the same figure. This is the application for which we'll create a Windows installer package to demonstrate the process of deploying a Windows application with a setup project.

To demonstrate the process of deploying an application through a Windows installer package,

we'll build a setup project for the NWOrders application. This is one of the sample applications we'll explore in detail later in this book. The NWOrders application lets you create orders for the North wind database. Users can specify the products to be added to the order either by their ID, or by their name. The selected products are added to a ListView control along with their prices, quantities, and discounts, and the order is committed to the database when the Save button is clicked. Figure 10.11 shows the main form of the NWOrders application.

You should experiment a little with the custom dialog boxes to get exactly what you want. Look up the default appearance of each custom dialog box in the documentation and then adjust them through their properties. The custom actions you can take based on the user's choice(s) on these custom dialog boxes are quite limited. Some of the dialog boxes allow you to start a custom application. The Register User dialog box, for example, has an Executable property, which you can set to the name of an EXE file. When the user agrees to register the application, the executable is invoked automatically. The Register User dialog box also exposes a property named Arguments, which you can set to a string with arguments to be passed to the executable that will handle the user registration.

Summary

In this chapter you learned the basics of deploying Windows forms applications. The new deployment technique is the Internet-based deployment, which makes the deployment of Windows forms applications as simple as the deployment of WebForms applications. With this type of deployment, nothing is installed at the target machines; applications are downloaded to the clients from a web server and executed. When the application is upgraded, the clients will detect the newer versions at the web server and will download them automatically. Internet-based deployment is also known as no-touch deployment and zero-install/zero-administration deployment, which indicates the expectations of Microsoft for this type of deployment.

Internet-based deployment is a very convenient deployment mechanism in corporate environments, but you have to deal with security issues. As far as the client is concerned, the application is downloaded from the Internet and as such it will be executed in a context of seriously limited privileges. Use the Microsoft .NET Framework Configuration to give the application the proper privileges. If the application doesn't interact with the local resources, then you don't need to assign additional privileges to the application. Web applications don't interact with the local computer's resources and you don't have to fiddle with their security settings. However, if you want to provide a rich user experience by making the most of the client, you must request that your application is executed with additional privileges. Because of this, the Internet-based deployment is best suited for applications that are deployed within a corporation.

The classical deployment method that relies on Windows installer has become a lot more flexible than those with previous versions of Visual Studio. The setup project is part of the solution and you can set up the installation actions with point-and-click operation in the IDE. We've explored the basics of creating Windows setup projects, which should be all you need to deploy an application within a corporate environment.

Chapter 11

Building Data-Driven Web Applications

THIS CHAPTER COVERS THE primary new tools—especially server-side controls such as the essential DataGrid—that make creating database-connected web pages a pleasure to program. If you've struggled with the task of attaching databases to websites in the past, you'll appreciate how much work .NET's tools and controls do for you, and how quickly you can build an effective solution.

First you'll read an overview of the important new advances that make .NET Internet-database programming so much more efficient than previous technologies. Then you'll see how to best make use of the DataGrid, DataList, and Repeater controls. You'll next see how to handle post-back and validation. Finally, you'll find out how to send graphics and when you would perhaps want to revert to old-style HTML controls, rather than the new .NET WebForm controls. Now, on to the overview of the advantages of .NET's approach to creating database-driven web pages the easy way.

New Features in ASP.NET

The .NET database features integrate well with its web page features. Both ADO.NET and WebForms offer high-level tools, rapid application development elements, and programmatic support for hooking up Internet browsers to databases.

In addition you'll find up-to-date special capabilities, such as the capacity to translate database tables into XML and vice versa. Life has become much easier for the Internet programmer, thanks to ADO.NET and ASP.NET.

There's no point in reviewing the mind-numbing struggles of the past few years as developers wrestled with e-commerce "solutions" that involved tedious client-side scripting (often either blocked for security reasons or the victim of browser incompatibilities), ActiveX, JavaScript, and other attempts to bridge the gap between databases and web pages. ASP.NET gives you a full server-side language that, along with client-side scripting, security via compiled code, and other features, makes our work far less wearisome.

The client's browser accepts this composed HTML, of course (though it would likely reject an executable object, such as a traditional ActiveX control, for security reasons). In this way, a relatively sophisticated user-interface is created in the client browser—sophisticated compared to traditional HTML GUI controls such as a Submit button. When the user interacts with their browser, the results are posted back to the server, which can then send a response to the client. In this way, rich client controls are made available, in spite of the usual security and bandwidth problems. The server control solution is effective for most kinds of GUI. Server controls are also browser agnostic, so the few remaining users of Netscape products can see the GUI too.

If you have any doubts about the efficacy of server controls, compare the more flexible, sophisticated visual and user-interaction features of the DataGrid (discussed later in this chapter) with the rather poor features of the traditional HTML table. If you want to display data in browsers, the DataGrid is clearly the superior choice.

Displaying Data on a WebForm

For the first example in this chapter, let's see how to use the code-behind feature of the .NET WebForm to make a connection to a database, then display some of its fields on the client browser. This example involves no server-side controls; it's simply directly written on the user's browser with the HTML `Response.Write` command.

Start a new VB.NET ASP.NET Web Application. Double-click the WebForm to get to the code-behind window (by default it's named `WebForm1.aspx.vb`).

Add these two Imports statements to the top of the code window:

```
Imports System.Data
Imports System.Data.SqlClient
Then type Listing 11.1 into the Page_Load event.
```

LISTING 11.1: CONNECTING TO A DATABASE

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

    Dim conString As SqlConnection = New SqlConnection _
    ('Data Source=localhost;Integrated Security=SSPI;Initial Catalog=puk
    conString.Open()

    Dim SQLc As SqlCommand = New SqlCommand("SELECT * FROM Author
    Dim datReader As SqlDataReader = _
    SQLc.ExecuteReader(CommandBehavior.CloseConnection)

    While datReader.Read
        Response.Write(datReader.GetString(1) & ",")
        Response.Write(" " & datReader.GetString(2))
        Response.Write(" --- " & datReader.GetString(3))
```

```
        ListBox1.DataSource = MyArray
        ListBox1.DataBind()

    Else ' it is a postback, so process the user's click
        Response.Write('You clicked: ' & ListBox1.SelectedItem.Text)
    End If

End Sub
```

Here you only need to fill this ListBox the first time you send the page to the user. So `If Not IsPostBack` Then makes this decision. If the user is sending a postback, the ListBox remains filled, but now you have to react to the user's click.

Validation

Users have been known to enter all kinds of wrong input, including zip codes for area codes, their date of birth in reverse Polish notation, and their mother's maiden name for their favorite pet. There's not much you can do to detect that last one—pet names cannot easily be distinguished from maiden names—but most user input can be screened before adding it to a database or using it to fulfill a catalog order.

If they're ordering a shirt and they type in 125 as their neck size, you can politely request that they revise this measurement. If they type in nine digits for an area code, you can respectfully suggest that they try, try again.

You can validate user input either programmatically or via the new .NET validation controls, as described in the following sections.

Programmatic Validation

There are various ways to programmatically validate user entries, but one of the most useful involves Regex, a complex language for various kinds of text management. In general, you don't want drive yourself barmy by trying to construct regular expressions yourself—there's nothing regular about them.

However, they can come in handy, so to use them just locate libraries of pre-written expressions on the Internet, or examples in VB.NET Help. I located the following mind-bender in Help. It determines whether or not a string is a valid e-mail address. Here's what it looks like:

```
"^([\w-\.]*)@((\[[0-9]{1,3}\. [0-9]{1,3}\. [0-9]{1,3}\. |(([\w-]+\.)+)) ([a-zA-Z]{2,4} | [0-9]{1,3}) (\?)$"
```

See what I mean?

This next example illustrates another way to interact with the user via the middle tier. Add `Imports System.Text.RegularExpressions` to the code window. Put a button and a

TextBox on a WebForm. The user enters an e-mail address in the TextBox, then clicks the button. Change the Text properties of these controls so they look like this:

```
TextBox1.Text = "Please enter your email address..."  
Button1.Text = "Click to validate"
```

Then type Listing 11.5 into the button's Click event.

[Team Fly](#)

 Previous

Next 

Sending Graphics

You can provide handsome backgrounds or dynamically generated graphs and other quick-response images using the new `Response.OutputStream` property. You can use this property to transmit a variety of different kinds of binary data to the client. In this example, you build a graphic using the powerful new GDI features in .NET, then you serialize the graphic to the `Stream` object returned by the `Response.OutputStream` property. Listing 11.6 generates the gradient and the Bezier curves shown in Figure 11.5:

LISTING 11.6: USING RESPONSE.OUTPUTSTREAM FOR RAPID GRAPHICS

```
Imports System.Drawing
Imports System.Drawing.Drawing2D

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

    'set format
    Dim b As New Bitmap(300, 500, Drawing.Imaging.PixelFormat.Format32bpp
        Dim g As Graphics = Graphics.FromImage(b)

        'build gradient
        Dim rect As New Rectangle(0, 0, 300, 500)
        Dim bl As New LinearGradientBrush(rect, Color.DarkGoldenrod,
            Color.PaleGoldenrod, LinearGradientMode.ForwardDiagonal)
        g.FillRectangle(bl, rect)

        'superimpose Bezier whip curves
        Dim p1 As New Point(54, 12)
        Dim p2 As New Point(212, 122)
        Dim p3 As New Point(134, 129)
        For i As Integer = 10 To 400 Step 100
            g.DrawBezier(Pens.BlueViolet, p1, p2, p3, New Point(i, 50))
        Next i

        'blank current contents and specify jpeg as response type
        Response.Clear()
        Response.ContentType = "image/jpeg"

        b.Save(Response.OutputStream, Imaging.ImageFormat.Jpeg)

        g.Dispose()
        b.Dispose()
        Response.End()
    End Sub
```



FIGURE 11.5 Send generated graphics or any binary data directly to the client browser.

This technique is not for a typical web page that the user would simply surf to via hyperlink or such. Just use an Image control for that kind of graphic. Instead, this is a page that is a *response* to a user query, and it is *dynamically generated*—not some graphics file that you're sending or embedding in a web page. Use this technique to, for example, generate a histogram out of the sales figures stored in a database. This on-demand graph could be quite useful to the traveling salesmen in your company: they could see at a glance, on a real-time basis, their current status relative to their competition. This feature transmits binary data, so it's not limited to graphics. Indeed, I could send this Word .doc file.

Using HTML Controls

So far you've worked with WebControls in this chapter, but it's possible that in some situations you might want to resort to the old-style HTML controls located in a separate tab on the Toolbox. For most situations, the WebControls are preferable, if for no other reason than they

permit you to easily integrate VB.NET via the code-behind process. Also, if you've ever tried to manage GUIs within HTML, you'll recall that it's confining, to say the least. All the HTML controls' functionality is available in equivalent, but usually more powerful and efficient, WebForm controls (WebControls). The only exception to this is the File Field control, which you can use to have the client upload a file to your server.

Both WebControls and HTML controls have an attribute collection, but WebControls are simply far richer than HTML controls. WebControls have a full set of properties and also feature a consistent and type-safe object model—consistent meaning that if you know how to set a BorderStyle

[Team Fly](#)

 Previous

Next 

property in one WebControl, you know how to deal with it in all WebControls. WebControls also sometimes offer high-level abstractions that have no HTML equivalent; the Calendar control is one example.

HTML controls can be useful, though, if you have to maintain or revise an existing ASP or HTML page. This approach might be simpler than translating the pages into .NET WebControl-based WebForms. HTML server-side controls either contains the `runat='server'` attribute or are enclosed with a `form` element that itself has the `runat=server` attribute, like this HTML password control:

```
<form id= "WebForm1" method== "post" runat= "server" >
<input
type=password>
</form>
```

Should you want to compel an HTML control to execute server-side like a server control, right-click the control in the design window and choose Run As Server Control. The Button control, for example, defaults to client-side execution.

Summary

This chapter explored the novel features that ASP.NET and ADO.NET bring to the previously formidable job of building data-driven websites. The chapter begins with a survey of the several problems solved by using web pages in concert with server-side controls such as the powerful, flexible DataGrid. You saw how using server-side controls to compose HTML pages for transmission to client browsers solves issues as disparate as security and bandwidth.

This chapter describes how to use various WebForm controls such as DataGrid, DataList, and Repeater. You also saw how to employ templates, deal with postback, and handle validation (both programmatic and control enforced validating). A technique for efficient graphics transmission was discussed. Finally, we covered a couple of reasons why—in spite of the clear superiority of server-side controls—you might want to occasionally revert to using legacy HTML controls instead.

Chapter 12

Peer-to-Peer Programming

TODAY'S APPLICATIONS ACCESS RESOURCES on remote servers, and the Internet is becoming an extended network that allows us to reach any computer (almost) as if it belonged to our local area network. We want to access information wherever it exists—retrieve the most up-to-date information and process it as needed, and where it's needed. New technologies, such as Web services, allow us to easily expose information to other systems, or consume information from remote systems.

Even Web applications are not always limited to a browser connected to a web server. You can write a Windows application that contacts a web server and downloads one or more files to process locally. It's also possible to upload files to the web server, as long as you specify the name of an application that runs on the server and knows what to do with the uploaded files.

The .NET Framework provides a number of tools for exposing objects to remote systems, as well as for consuming objects on remote systems. In addition to the new tools, Microsoft has enhanced the traditional tools for peer-to-peer programming. Sometimes we don't need to expose our data to the world, just to specific remote systems. To enable two computers to talk to each other, you must use the System.Net namespace, which exposes the required functionality. In this chapter we explore the System.Net namespace and we show examples of peer-to-peer programming. You'll see how to write applications that run on two different computers, contact one another, and execute commands on the remote computer. These applications are written in pairs and they allow you to determine how the two computers will exchange information. You can use your own encryption techniques to protect your data, use custom authentication techniques, and have complete control over the flow of data between the two machines.

Internet Addressing

Before we start our exploration of sockets and peer-to-peer programming, we'll briefly discuss the System.Net.Dns class, which simplifies the task of addressing computers on the Internet. You're probably familiar with the topics of this section, but we'll repeat a few basic terms for the sake of VB programmers who are new to Internet programming.

Every computer on the Internet is identified by a unique address, known as the IP address. The IP address is a long number that is written as a group of four numbers, each one in the range of 0

Aliases property This property returns (or sets) a list of aliases associated with a host.

HostName property This property returns (or sets) the friendly name of the host. The System.Net.Dns class exposes a few methods to manipulate computer addresses, which are:

GetHostByAddress(IPAddress) method This method accepts an IP address as argument and returns an IPEndPoint object. On my computer, the statement

```
Console.WriteLine(GetHostByAddress("127.0.0.1").HostName)
```

returned the string "PowerToolkit." The GetHostByAddress method will return a hostname if the client with the specified address is on the same local network, or if it belongs to network with a registered name.

GetHostByName(hostname) method This method accepts a hostname as argument and returns the host's IP address. The following statement will return your computer's IP address, if you change the hostname to your computer's hostname:

```
Console.WriteLine(System.Net.Dns.GetHostByName(_  
    "myHost").AddressList(0))
```

If you're on a local area network and the Internet at the same time, the IPAddressList array will have multiple elements (multiple IP addresses).

Resolve(hostname) method This method accepts as argument an IP address or a hostname and returns an IPEndPoint object that represents the host. The argument can be either a friendly name (like "PowerToolkit" or "www.domain.com") or an IP address.

Now we can switch our attention to the classes for peer-to-peer programming, starting with the concept of sockets.

Using Sockets

At the lowest level, network programming consists of programming with *sockets*. Sockets are an old concept in network programming and they represent input points at a system, where a remote system can connect and make requests. There are many types of sockets, but the most common ones are the Internet sockets, because they deal with Internet addresses. Internet sockets come in two flavors: UDP (User Datagram Protocol) sockets (also known as Datagram sockets) and TCP (Transmission Control Protocol) sockets. The difference between the two is that UDP sockets are connectionless. Every time you need to send data using a UDP socket, a new connection to the remote machine is established. The connection is closed automatically when the data arrives at the remote machine. Every package of data is independent of the others, because it carries in its header all the information needed for its delivery.

TCP sockets require that a link between the two computers be established before they start exchanging data. The advantage of TCP sockets is that they're more reliable than UDP sockets. Packets sent through a UDP port may arrive in different order than the order in which they were sent. Moreover, a UDP packet may be lost without any indication. The sending machine will not receive a positive or

[Team Fly](#)

 Previous

Next 


```
        TCPSocket.Close()  
    End If  
End If  
End Sub
```

The Socket object exposes asynchronous versions of its methods. They're the BeginListen, Begin-Accept, and BeginReceive methods, which initiate the appropriate action in asynchronous mode. You need to implement AsyncCallbacks to intercept and process the event of the completion of the operation. However, most typical applications use the TcpListener, TcpClient, and UdpClient classes. These classes abstract the operations of the two types of connections and simplify coding. However, a basic understanding of sockets and the basic principles demonstrated in the previous sections are necessary to use the classes that are specific to a protocol.

Because the TCP protocol is inherently more reliable than the UDP protocol and is also used more often, we're going to demonstrate how to use the TcpListener and TcpClient classes to build a chat application.

The TCPChat Application

In this section you'll build a fairly advanced application that allows multiple remote clients to engage in a chat. Figure 12.3 shows the server and a few clients of the application. Each client joins the conversation by establishing a connection to the server. Once the client is connected to the server, it sends messages to the server. The server displays the incoming messages on a TextBox control on its own interface and then broadcasts them to all clients. Once a message arrives at a client, it's displayed on a TextBox control, along with the name of the person who sent the message. As each client establishes a connection to the chat server, it also establishes a username for the session.





FIGURE 12.3 The TCPChat server and clients

[Team Fly](#)

◀ Previous

Next ▶

Interacting with Web Resources

Another group of classes in the System.Net namespace handles the interaction with web resources. The WebClient class provides the functionality needed by a Windows application to interact with a web server: retrieve HTML pages or files and upload files to the server. The WebClient class is very simple and supports synchronous operations only. However, it abstracts the details of accessing a web server and makes the process of exchanging files with the web server as simple as reading from, or writing to, a local file. The WebClient class provides methods for exchanging information with a web server through streams, similar to accessing local files.

The WebClient class is part of the System.Net namespace, which you must import to your application. Then you can create instances of the WebClient class and call its methods. A WebClient object need not establish a connection to the web server explicitly; you just specify the desired URL when you request a document, or when you want to upload a document from the local computer. Because of this, using the WebClient class is almost trivial, and its functionality is exposed through a small number of methods, discussed next:

DownloadData method The DownloadData method downloads data from a web server and returns them in an array of bytes. The syntax of the method is:

```
WebClient.DownloadData(documentURL)
```

where documentURL is the URI of the document to download. If you're downloading an existing file, you specify the URI of this file. You can also specify the URI of an ASP application that generates its output on the fly. The output of the script is transmitted to the client and you can retrieve it as an array of bytes. The following statements will download the main page of the Sybex site and store the HTML document in an array of bytes:

```
Dim wClient As New WebClient  
Dim bytes() As Byte  
bytes = wClient.DownloadFile("www.sybex.com")
```

To convert the byte array to a string, use the members of the System.Text.Encoding class. The following statement will display the HTML code of this page on a message box:

```
MsgBox(System.Text.Encoding.UTF8.GetString(bytes))
```

DownloadFile method The DownloadFile method is similar to the DownloadData method, but a little more convenient, because it allows you to specify the path of the file where the data will be stored at the client. The syntax of the DownloadFile method is:

```
WebClient.DownloadFile(documentURL, localFileName)
```

The first argument is the document's URL and the second argument is the path of the local file where the downloaded data will be stored. The following method will download the main page of the Sybex site and store it to the specified local file:

```
Dim wClient As New WebClient  
wClient.DownloadFile("www.sybex.com", "C:\DLoads\Data\Sybex.htm")
```

[Team Fly](#)

 Previous

Next 

```
        txtResponse.AppendText(data & vbCrLf)
        data = RStream.ReadLine
    End While
    wResp.Close()
End Sub
```

Reading lines off the incoming stream synchronously is not a very efficient process either. The Stream object exposes the BeginRead/BeginWrite and EndRead/EndWrite methods, which are very similar to the asynchronous methods of the WebResponse object: they accept a delegate and they invoke it when the read/write operation has completed. You can edit the code of the application and make the read operations asynchronous as well.

Summary

The .NET Framework was designed from the ground up with the Internet in mind. It provides numerous tools that simplify the communication between remote systems, including the all-new Web services. In addition to the new ways of harnessing the Internet, the .NET Framework includes the basic classes that expose the traditional functionality of sockets.

In this chapter you've learned the basics of the Socket class, as well as how to use specific classes to exchange data with the TCP and UDP protocols. These classes are the TCPListener, TCPClient, and UDPClient classes and they abstract the basic operations you'd have to perform with traditional sockets to move data between two computers using the TCP and UDP protocols, respectively. The TCP protocol requires a dedicated connection, and each computer engaged in the conversation has a distinct role, either as a client or as a server. The UDP protocol is connectionless, and the computers involved in the conversation are all clients.

You've also learned how to use the WebClient, WebRequest, and WebResponse classes to interact with web resources from within your code. The WebClient class is the simplest one; it allows you to exchange data with a web server in a synchronous mode. The other two classes provide asynchronous methods, which enable you to write functional and responsive interfaces that interact with remote resources.

Chapter 13

Advanced Web Services

Nobody Yet Knows What the final ratio between web-based computing and local computing will eventually be. Will it end up 10% local, or even 0% local—with all databases and computations residing on the Internet with your home and portable devices merely dumb terminals? Or will the speed, and especially security, advantages of local computing cause a backlash against the current trend toward distributed computing?

No matter how it turns out, it's clear now that data storage and processing are currently migrating from local machines to Internet servers. What we used to call personal computing is mutating into something more like extended computing, with software subscriptions potentially replacing ownership, remoting replacing self-contained applications, and servers located whoknows-where replacing your resident hard drive.

And we programmers have to deal with this new .NET world, learning new techniques. For example, debugging might require that we step through a series of procedures located on various hard drives around the world. And how do you preserve state in a "stateless" environment?

Obviously new communication and security issues arise when you call a procedure across the world, and wait for the response. If you substitute the words *Web* for "across the world," and *service* for "procedure," you come up with Web service—the idea that a query-response messaging relationship can be set up between widely distributed computers, and that this relationship can be both efficient and secure.

That's the hope for Web Services, a novel technology built upon familiar components (fundamentally, computing is *always* about data that gets processed, and always will be, no matter what new communications protocols are invented, or what new names are used for it).

What *Are* Web Services?

First, how big is a Web service? Some say that Web services can be as large as a full business solution—a set of applications working together to handle a complete distributed enterprise.

Others say that Web services are small, individual procedures—single functions that accept some data, process it, and send back a result. We shall see. Currently the term *Web service* is used to describe both large and small processing—the essence being that a message is sent over the Internet to trigger the Web service.

Here is a list of characteristics that define Web services:

- ◆ They have no user interface.
- ◆ They are published and consumed via a network (the Internet, an intranet, and so on). They are not *localized*.
- ◆ SOAP and other XML-based technologies send messages using *ordinary, plain text*. This, obviously, raises some security issues. Plain text can be read by anyone, and aside from that, many people mistakenly think that a Web service could not transmit a virus. Given that firewalls are supposed to pass XML through (it's sometimes simply seen as a flavor of HTML), Web service communications slide right in. Hackers, though, know full well that virii executables can be embedded as ASCII strings.
- ◆ They are similar to traditional objects (they expose methods and properties, and their clients consume what they expose). The difference between a classic object and a Web service is that the latter isn't tightly coupled, doesn't demand that the consumer and service share the same object model. Instead, XML (and its derivatives) is the shared language into which members are translated during the request-response communication.
- ◆ They communicate via XML (SOAP variation), theoretically thereby eliminating language-, and even platform-, dependence. They are not proprietary, like COM objects. However, the Web service itself is written in a computer language—not XML. XML is a metalanguage: It can describe language and data, but cannot actually compute. When the Web service receives the client's request (or the client receives the service's response), the XML message has been automatically repackaged into an object. If you're familiar with .NET object-to-XML serialization, you understand that this process is handled for you—there are built-in .NET serialization features. (You won't have to handle the serialization process in the Web service examples in this chapter.) Note that this process is similar to DCOM, but open XML is used to transmit the object rather than a proprietary format.

Creating a Web Service

In this example you create and test a Web service that requires parameters. You'll pose as the client to test the service's response. The service exposes a method that accepts a string and returns the words in it reversed (*one two three* becomes *three two one*). Start a new VB.NET project and double-click the ASP.NET Web Service icon in the New Project dialog box. (If this type of new project cannot be created, read the VS.NET documentation to find out how to install IIS.) You'll see a design window for this project, but it's not intended to be used for visible UI controls such as TextBoxes; rather, it's for adding database-connection controls and such. Switch to code view.

This line is required:

```
Imports System.Web.Services
```

Delete the commented sample code lines and replace them with Listing 13.1.

[Team Fly](#)

 Previous

Next 

Not surprisingly, ASP.NET has facilities for caching data and for allowing you to define how long it remains cached. Here's how to do it.

Assume that you want to cache the response to your Headlines Web service for a half hour, then refresh it. If you were providing headline news service that you wanted to update every half hour, this would be the way to do it.

```
<WebMethod(Description := "'Number of times this service has been accessed", _  
    CacheDuration := 1800, _  
    MessageName := "Headlines")> _  
    Public Function Headlines() As String
```

The CacheDuration is expressed in seconds, and in this example, the first time this Web service is called, the response is calculated and returned to the client, but is also placed into the cache. For the next 30 minutes, any subsequent calls are *not* calculated. Instead, the cached response is merely sent to the clients. Obviously, this technique will often improve response time and Web service performance.

Consuming a Web Service

In the previous example, you saw how to write and test a Web service. Now let's move to the other side and see how to consume a Web service.

Later in this chapter you'll see how UDDI, WSDL, and other initiatives facilitate the publication, discovery, and consumption of Web services. For now, though, let's create a quick example that illustrates how, from within VB.NET code, you would go about consuming a Web service. The first step is to add a Web reference to a VB.NET project. For this example, you'll consume the Web service from within a Windows-style project.

Create a new VB.NET Windows-style project by double-clicking that icon in the New Project dialog box. Add a TextBox to the form, then choose Project ➤ Add Web Reference. You see what looks like a streamlined browser window (shown in Figure 13.3) where you can search for Web services and see details about them, such as the parameters they expect when you submit a message to them for processing.

For this example, you want to contact and consume the Web service you created in the previous example in this chapter. Click the Web Services On The Local Machine link and VB.NET will provide you with a list of all the .ASMX files on your machine. Scroll past any QuickStart or other sample services until you locate Service1 (the default name VB.NET gives new Web services you write—we didn't change this default name in the previous example). You also see a path, like this:

<http://localhost/services/Service1.asmx>

as the "URL" for your service. Just to be sure that you have the correct service, click the Service1 link, as shown in Figure 13.4, to see that this is the correct service. You should see this description:

Reverses the words in a submitted string

The following operations are supported. For a formal definition, please review Service Description.

ReverseWords

[Team Fly](#)

 Previous

Next 

If you look at the service description, you see the necessary data types and parameters required: `ReverseWords(s As string) As string`. This is the right service, so click the Add Reference button. Look in Solution Explorer to see that your reference has been added.

This reference is similar to the simulation of Internet connections that is used to test ASP.NET projects and Web services (as in the previous example). The *local* host refers to your local machine pretending to be an Internet URL.

Now write the code that consumes a Web service by feeding a parameter to it, then receiving the response and displaying the result in the TextBox.

Type in the source code shown in Listing 13.2.

LISTING 13.2: CONSUMING A WEB SERVICE

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

    Dim WebServiceAnswer As New localhost.Service1

    Dim param As String = "This is my sentence"

    TextBox1.Text = WebServiceAnswer.ReverseWords(param)
    Me.Text = WebServiceAnswer.Url

End Sub
```

Press F5 to instantiate a new Service1, then invoke it and pass a parameter. If you have a firewall, it will probably ask if you want to permit your VB.NET project to "contact the Internet," or it may ask your permission to let this service connect to local host port 80. You get back from the Web service this response: *sentence my is This*.

TIP If you edit a Web service, be sure to right-click localhost in the Solution Explorer, then choose Update Web Reference in the context menu to rebuild the service.

Preserving State

Like many other Internet communications, Web services are theoretically stateless. Parameters are passed to the service, but that data is only persisted in memory while the service is generating a response. After the response is sent, the parameters are discarded. Normally, details like a client's fax number are not *retained* by the server. Statelessness is often necessary—you usually don't have room nor reason to store the number of every visitor to your popular website.

Some Web services don't need to retain data about the client. There's no need to retain their zip code after sending a client the local weather report, for instance. But what if you do want to retain data about certain returning visitors, such as customers, so they don't have to repeatedly supply you with their fax number every time they place an order?

Using Session State

ASP.NET includes a Session object that can persist data and can make that data global to all the pages in a given website. To see how to use the Session object, start a new ASP.NET Web service project and replace the commented green template code with the code in Listing 13.3.

LISTING 13.3: SAVING STATE IN THE SESSION OBJECT

```
<WebMethod(Description:='Figures visits.',
enablesession:=True)> Public Function CountVisits() As Integer

    If Session("counter") Is Nothing Then

        Session.Add("counter", 1)
    Else
        Session("counter") += 1
    End If

    Return Session("counter")

End Function
```

WARNING You must add the *Description* argument in this method, because the *enablesession* argument can't come first in an argument list.

Your session variable named *counter* tracks the number of times this method is invoked during a given session. Notice that the name within the parentheses—here, *counter*—is used as an ordinary variable name. If you wish, you could get a unique ID from the session object by changing the declaration, as in

```
Public Function CountVisits() As String
```

and then replacing the Return with this line:

```
Return Session.SessionID
```

When this Web service is consumed, you get the ID, which looks something like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<string xmlns="http://tempuri.org/services/Service1"> d2lpz1q5pk1ctnqntwcb4a4
</string>
```

Making a Database Connection

Given that most all computer business applications require database connections, you want to know how to connect a database to Web services. This next example shows you how to make that connection. The example assumes that you have the Pubs sample databases available on your hard drive. If


```
<job_id>2 </job_id>
<job_desc>Chief Executive Officer</job_desc>
<min_lvl>200</min_lvl>
<max_lvl>250</max_lvl>
</Jobs>
<Jobs diffgr:id='Jobs3' msdata::rowOrder= "2">
  <job_id>3</job_id>
  <job_desc>Business Operations Manager</job_desc>
  <min_lvl>175</min_lvl>
  <max_lvl>225</max_lvl>
</Jobs>
```

Implementing WSDL

XML attempts to self-describe—to contain both data and information describing that data. To assist with this difficult job when XML is used to send Web service messages, a new language named WSDL (Web Services Description Language) was developed. WSDL is an effort to standardize descriptions of responses, formats, and protocols used during Web service messaging. WSDL can describe these aspects of a Web service message:

- ◆ The address of the Web service
- ◆ What kind of processing should be carried out on the data
- ◆ The type of exchange (one-way, multicast, response/request, solicit/response)
- ◆ The type of data being exchanged between client and server
- ◆ The type of message used for input and output (procedure-style or document-style)
- ◆ The protocol used to send the message(s)
- ◆ Error information

WSDL descriptions are written in XML and are usually placed within an XML schema or set of schemas. The client and service employ the same schema and agree both on how the client should format its message and how the Web service should process the data it gets from the client. In this way, WSDL permits various proprietary models to easily couple, such as COM or ERP.

WSDL is yet another in the many initiatives designed to promote interoperability. It's called a "contract" between client and service, and ideally should describe all the information necessary to permit successful Web service consumption without the need for human intervention. Alas, this remains more a dream than a practical reality, like other aspects of the XML program. Nonetheless, WSDL can improve the readability (by humans) of the interaction between client and service.

Simply put: WSDL describes what kinds of messages a given server accepts, specifying the format required and the types permitted.

Here are the specific elements of WSDL:

Message Optional, can appear in various places within the document.

The binding element describes the protocol, any serialization, and encoding for the message transmission.

The service element concludes a WSDL description, and it provides yet another listing of the final destination of the client message. A message can move to multiple locations on its trip to the service, but in the service element is the actual, final address where the Web service itself is located. However, WSDL descriptions can contain multiple Web services—so consumers are thus able to select between options (preferring, say, a Web service that returns its response in pounds rather than dollars).

```
<service name='Service1'>
  <documentation>Adds or Multiplies to Integers</documentation>
  <port name="Service1Soap" binding="s0:Service1Soap">
  <soap:address location="http://localhost/xx/Service1.asmx" />
  </port>
</service>
```

Seeing SOAP, WSDL, and the Reference Map

You don't have to generate the dependency SOAP, WSDL, and Reference Map files—VB.NET does this for you when you create a Web service, then add the service to a project. Nonetheless, you should take a look at them to see the information they contain. In particular, there's a `Reference.vb` file that contains information you need to know to consume a Web service (the parameters it expects, and what it returns), written in familiar VB.NET code.

Use Project ➤ Add Web Reference to add a Web service to your current project; for this example I'll use the ShowJobs service created previously in this chapter.

Now slowly move your mouse pointer across the icons in Solution Explorer's title bar to locate the one labeled Show All Files. Possibly the icons are not visible, which means you're in a mode in the IDE that the designers thought wasn't a context in which you'd want to see the icons (this kind of thing seems a little *too* helpful to me; what harm is done by leaving these icons always visible?). To make them visible, click the name of your project (it's the line in boldface) in Solution Explorer.

Now expand the Web service node (Web References\Localhost). Take a look at the Disco (discovery) file, then also double-click the WSDL file to see what it looks like for this service. Disco is Microsoft's alternative to UDDI (or their supplement, you might say). Disco is supposed to be an easier way to figure out which Web services are available on a particular server. Where UDDI is an Internet-wide registry, Disco does the same job for smaller, intranet services.

The `Reference.vb` file in Solution Explorer can be the most useful to programmers trying to figure out how to access a Web service (unless the service's own documentation is clear). In particular, this VB code makes it pretty clear that no parameter is expected, and that a dataset will be returned from this Web service:

```
Public Function ShowJobs() As System.Data.DataSet
```

UDDI: The Registry

Web services employ SOAP, WSDL, and UDDI. SOAP explains how to use the service (and possibly additional documentation). WSDL describes the entire transaction between client and service.

[Team Fly](#)

 Previous

Next 

To complete the Web service package, you need UDDI (Universal Description, Discovery, and Integration) as well. It provides specifications for a directory of Web services. The UDDI Business Registry (also called UBR and *cloud services*) is the actual registry where potential consumers can search through UDDI lists, and where service publishers can register and describe the Web services they offer. The registry is divided into three sections:

Yellow pages The most abstract layer, simply listing data such as the company's product or type of business (geologic research, for instance). Think of it as similar to the phone book yellow pages.

White pages More specific details about a company that is offering a Web service (addresses, phone numbers, salespersons' contact info, and so on).

Green pages The actual nitty gritty details: specs explaining the Web service itself. You can put whatever you want in the Green pages, but typically you include the URI to the address of the Web service, or reference associated SOAP or WSDL files. (You aren't required to use SOAP here. You can use alternative descriptions. And, if you wish, you need not include details about your Web service, but instead simply provide an e-mail address or Web page where customers can look for further information.

You can browse Microsoft's node of the UDDI Registry, or you can even add your own Web service to it to test it. To register your own service, choose Help ➤ Show Start Page, click the Online Resources tab, then click the XML Web Services option in the left pane.

You can also use this registry to add Web services to your .NET projects. When you open the Project ➤ Add Web Reference dialog box, you can access UDDIs in several ways (in addition to the "local machine" option described previously in this chapter):

Browse UDDI Servers on the Local Network Click this link to see servers in your LAN that are currently publishing UDDI described Web services.

UDDI Directory With this link you can traverse the Microsoft UDDI Registry and discover the Web services that have registered on Microsoft's node.

Test Microsoft UDDI Directory Use this link to search for test Web services that have been posted here, so you can experiment with the Web service technology as a consumer. You can also use this test directory to register and publish your own Web services for testing purposes.

For further information on the UDDI Registry in general, look at:
<http://www.uddi.org/register.html>.

Testing a Published Web Service

If you want to try consuming a Web service that's published on the Internet (written by someone else), you can give it a try. It's useful practice, to see if you can manage to figure out someone else's intentions and successfully get back a response.

I've looked around and many of the test services listed are either impossible to figure out without further help from their authors or no longer active at their URL. Nonetheless, you might find one on your own that you can discover (figure out) and consume (use).

[Team Fly](#)

 Previous

Next 

When you press F5 to run the program, you should see an HTML-formatted message, returned from this Web service, appear in your message box. If this example doesn't work for you, try other available Web services until you discover one that does work.

Security Considerations

The third oldest profession is security—the attempt to conceal information without actually destroying it in the process, or the attempt to defend yourself without actually imprisoning yourself in the process.

These goals—privacy and protection—have been sought since envy became a factor in human relations. In other words: since Adam's boys. And each time we think we're getting close to an effective solution, the goal recedes and we realize that the envious are just as clever as the envied.

Web services are just as vulnerable to security problems as any other technique that involves Internet messaging. You have the problem that someone might intercept the message and read it. This can be solved pretty effectively with strong encryption, as described in Chapters 5 and 6. Similarly, the related problem of validation (has someone changed \$1000 to \$100000?) can be solved via encryption. If they cannot read the message, they cannot modify it.

The other security issue, authentication, is less easily solved, especially when you consider that one goal of Web services can be described as letting strangers into your server so they can execute commands and invoke procedures. That's just the sort of thing that virus protection software and firewalls are designed to prevent.

HTTP and HTML are supposed to slide into servers right through port 80. Firewalls permit this because HTTP and HTML transport and express only harmless documents, not executables.

SOAP, though, extends these capabilities beyond page description and text messages into the ability of a remote client to invoke procedures and issue commands on the server.

Some experts suggest blocking text/xml content types, or messages with SOAPAction in their headers, but this throws the babies out with the bathwater. The committees that govern XML and, by extension, Web services have been trying to come up with practical recommendations. Likewise, firewall vendors are also seeing what can be done to tell the bad guys from the good guys. Can the problem of entertaining strangers be solved? Time will tell, but history suggests that the answer is no. Security can often be strengthened, but never perfected.

Summary

This chapter covers Web services and related technologies. You saw that computing—no matter how distributed it becomes, and no matter what names they give new technological twists—always comes down to two things: data and processing. Web services are no different. True, they send messages via XML, they operate remotely, and they face special security and communications challenges. But, in essence, they accept a request to process some data, just like any classic function, utility, or application.

You learned how to write a Web service, how to cache data, and how to consume a Web service. You also saw how to preserve state using the Session object and how to deal with database connections.

The chapter concluded with an examination of WSDL, the Web service description language; XML and SOAP features; and the UDDI registry, the official Universal Description, Discovery, and Integration registry where you can list your Web services, and locate others' services, along with descriptions of how to consume them.

Chapter 14

Building Asynchronous Applications With Message Queues

WHILE MOST PROGRAMMING TASKS are synchronous, there are situations when we must implement asynchronous systems. An asynchronous system involves two or more computers that must exchange information, but we can't assume that they're always connected, or that each system will perform certain tasks in a timely manner. In an asynchronous system, we should be able to send a request from one computer to another and be sure that the computer that receives the request will eventually process it. A web server that accepts orders does not usually process them. The orders are forwarded to another machine to be processed, and this machine is usually on a different network. The processing of an order may actually involve communication with a remote system as well (ordering an out-of-stock item from its publisher, or placing large orders to a warehouse) and it may take a while to complete. The computers involved in an operation may not be connected at all times, or one of the computers may take a while to perform a task. The other computers involved in the process shouldn't have to wait for each task to complete.

In this sense, the architecture just described is that of an asynchronous system. More specifically, it's a loosely coupled system. In such a system we assume that all resources are available, but not necessarily at all times. Applications for systems consisting of multiple computers that communicate with one another are based on a so-called loosely coupled architecture. In English, this term means that the various components of the system are connected to one another, but they operate independently of one another and they may disconnect at any time. This architecture requires a secure, reliable mechanism for the system's parts to exchange information. This mechanism is provided by the Microsoft Message Queuing (MSMQ) component, which comes with both Windows 2000 and Windows XP, but it's an optional component that you will have to install through the Add/Remove Windows Components tool. The basic functionality of MSMQ is to set up queues, send messages to these queues, and receive messages from the same queues. Your application will create messages and send them to a specific queue. After that, MSMQ takes over and makes sure that the message is delivered to the destination queue. If the message can't be

delivered, MSMQ can generate an acknowledgment message to indicate that the delivery of the original message failed.

Consider an application that runs on a salesperson's portable machine. The salesperson should be able to record orders on the go, but can't assume a connection to a server at the company's headquarters. The orders should be uploaded to the company's server as soon as possible, so that they can be processed in a timely fashion. What we just described here is a loosely coupled system: the order-taking application on the portable computer and the server at the company work together, but they can't be connected at all times. Data is stored in the portable machine and is uploaded when the two computers are connected. One of the basic requirements of a loosely coupled system is that information be safely stored locally until the two systems are connected and can exchange information. Messaging is an excellent mechanism for moving information from one machine to another.

As you know, messages are an ideal mechanism for passing information between remote computers. You can also send messages to a queue on the same computer. A simple technique to develop multithreaded applications is to create messages that represent specific tasks and leave them on a queue, rather than process each task. Another application can retrieve the messages from the queue and perform the task described by each message. This type of application isn't really a multithreaded application, but it's an efficient mechanism for running tasks in the background while the front end is free to interact with the user, as long as the tasks need not communicate with one another. If the number of tasks exceeds the capacity of a single workstation, you can have multiple workstations process the messages in the queue.

Queues and Messages

A message queue is a structure for storing messages, much like a first in, first out (FIFO) queue. Messages are stored in the queue according to their priority and the time they arrived. Messages with the same priority are stored in the queue in chronological order (the order in which they're received) and are read in the same order. When you read a message, the oldest message in the queue will be returned. You can change the default order by setting the priority of the messages you write to the queue. Messages with higher priorities are read before messages with lower priorities, even if they haven't been in the queue as long. The order of the messages in the queue is determined by MSMQ and you can't change the order of the messages in a queue after their arrival.

The messages themselves are serialized objects. Although we can create simple text messages, we rarely do. We create one or more custom classes that represent physical entities (such as orders, products customers, and so on) and our messages are instances of these custom classes. The custom classes must be marked as serializable, so that MSMQ can serialize the corresponding objects either in binary or XML format. At the receiving end we read a Message object from the queue, cast it to the appropriate type, and process it in our application.

Unlike mail messages, queue messages have a well defined structure, which must be known to both the sending and receiving end. In effect, they're equivalent to the messages we exchange through our mail software, but meant to be understood and processed by applications, not humans. In the same sense, MSMQ is equivalent to a mail client that can send and receive messages. The queue, finally, is equivalent to a message store.

The process of passing messages between two machines (whether they're mail messages or MSMQ messages) is inherently asynchronous: the sending machine isn't blocked until the message

[Team Fly](#)

 Previous

Next 

This form of the Send method sends a simple message to the queue. The first argument is the message's body and the second argument is the message's label (a string that will be displayed on the MSMQ snap-in under the Label heading).

To read the message from the queue, call the MessageQueue1 component's Receive method. This method returns a Message object, which exposes many properties. The Body and Label properties return the message's body and label, respectively. Enter the following statements in another button's Click event handler to retrieve the message and display its label and body on a message box:

```
Private Sub Button2_Click(ByVal sender As System.Object, _  
                          ByVal e As System.EventArgs) _  
                          Handles Button2.Click  
    Dim msg As Message  
    msg = MessageQueue1.Receive  
    MsgBox(msg.Label & vbCrLf & msg.Body)  
End Sub
```

The Receive method retrieves the first message in the queue. Usually this is the oldest message in the queue, unless the messages have different priorities. Messages with higher priority are stored ahead of messages with lower priority. You can't set the priority (or many other properties of a message) with the simple form of the Send method shown here. As you will see shortly, you must create a Message object, then set its properties, and finally send it with the Send method. We rarely use this simple form of the Send command; we've only shown it here to simplify the example.

Even simple text messages are first serialized into XML format before they're written to a queue. You can read the XML-serialized description of a message with the BodyStream property of the Message object. The following statements read the XML description of a text message:

```
msg = MessageQueue1.Receive  
Dim buffer(msg.BodyStream.Length - 1) As Byte  
msg.BodyStream.Read(buffer, 0, msg.BodyStream.Length)  
Console.WriteLine(System.Text.Encoding.UTF7.GetString(buffer))
```

The Receive method won't return without reading a message. If the queue is currently empty, the Receive method will wait until a new message arrives. We describe how the Receive method is used in the section "The Message Class," later in this chapter.

The simple code examples we use in the following sections are parts of the SimpleQueue sample application. This application assumes that the ToolkitQueue private queue exists on the local machine. Follow the steps outlined in the section "Creating New Queues" to create the ToolkitQueue on your machine (or edit the properties of the MessageQueue1 object in the application).

The MessageQueue Class

Once you know how to reference queues, you can use the MessageQueue class's methods to create new queues, delete existing ones, and find out whether a specific queue exists or not from within your code. The Exists method of the MessageQueue class returns True if the queue specified with the argument to the method exists, False otherwise. To create a new queue, pass its path or format name as argument to the Create method of the MessageQueue class. The Create method has a second overloaded form

TABLE 14.1: THE PROPERTIES OF THE MESSAGEQUEUECRITERIA CLASS

FILTER NAME	DESCRIPTION
<code>Category</code>	The category of the desired queue(s). A queue's category need not be unique.
<code>CreatedAfter</code>	A date that filters out the queues that were created before the specified date.
<code>CreatedBefore</code>	A date that filters out the queues that were created after the specified date.
<code>Label</code>	A string that specifies the queue's label. MachineName The computer name on which the desired queue(s) reside.
<code>ModifiedAfter</code>	A date that filters out the queues that have not been modified after the specified date.
<code>ModifiedBefore</code>	A date that filters out the queues that have not been modified before the specified date.

Another method to retrieve the queues on a computer is to use the `MessageQueueEnumerator` class. The `GetMessageQueueEnumerator` method of the `MessageQueue` class returns a `MessageQueueEnumerator` object, which you can use to iterate through all public queues in the network. This is a typical enumerator that exposes the `MoveNext` method, which moves to the next queue, and the `Current` property, which references the current queue

The `MessageQueue` classes expose members to send messages as well as retrieve messages from queues. These methods make use of the `Message` class, which represents queue messages. In the following section we'll discuss the `Message` class and then look at the methods of the `MessageQueue` class for manipulating messages.

The Message Class

The other major class of the `Messaging` namespace is the `Message` class, which represents MSMQ messages. You've seen how to send simple messages with the `Send` method of the `MessageQueue` class, but applications exchange information in the form of objects, not strings. Moreover, the simple form of the `Send` method we used in our earlier example doesn't allow you to set the properties of the message: You can't assign a label to the message (a string that describes the message while it resides in the queue), nor can you change the message's priority. You should always create `Message` objects and pass them to the `Send` method, even if these objects are strings.

The `Message` class exposes only properties and no methods. The purpose of this class is to enable us to manipulate the properties of the messages before sending them to a queue, or read the properties of messages retrieved from a queue. To send a `Message` object you must create an instance of the `Message` class, populate its properties, and then pass it as argument to the `Send` method of the `MessageQueue` class. The `Message` class exposes two types of properties: read-only properties designed to work with incoming messages and read/write properties designed to work with outgoing messages. To read a message off a queue, you must create a

new instance of the Message class and assign to this object the value returned by one of the methods that reads messages from a queue.

[Team Fly](#)

 Previous

Next 

specified ID. If no related messages exist in the queue, the `PeekByCorrelationID` method returns a `Nothing` value.

There's one simple operation that's fairly expensive in terms of resources, and this is the counting of messages in a queue. There's no method to return the number of messages in a queue. You must call the `GetAllMessages` method, which accepts no arguments and returns an array with all the messages in the queue, and then examine the array's `Length` property (or call its `GetUpperBound` method). This is a fairly expensive operation and its value is questionable, because the number of messages in the queue varies all the time. As with all classes that implement the `IEnumerable` interface, you should use the appropriate `Enumerator` to actually process the messages. If you want to look at the messages and then decide which one to process, you can call the `GetAllMessages` method to retrieve a snapshot of the queue and then retrieve each message you want to process with the `ReceiveByID` method, or process one or more of the messages returned by the `GetAllMessages` method and then remove it from the queue with the `ReceiveByID` method. We'll use the `GetAllMessages` method in the `OrdersServer` project later in this chapter to copy all the messages in the queue and allow the user to view them, move back and forth through them, and select which ones to process. The processed messages will be removed from the queue with the `RemoveAt` method.

Acknowledgments and Time-Outs

Sending a message to a remote queue doesn't mean that the message will actually arrive at its destination. The destination queue may not exist, the computer on which the queue resides may be disconnected, or the application that sent the message may not have the rights to write to the specific queue. Even if a message is delivered to the destination queue, we can't be sure that the message will actually be retrieved from the queue and processed. Sometimes it's critical that a message is processed within a predefined time interval. If the messages represent orders, for example, your application should eventually find out which orders were processed and which not, and take the appropriate action. Delaying the processing of an order for a week is simply unacceptable.

A robust distributed application should be able to know whether a message has reached the destination queue and whether it was retrieved from the queue and processed. MSMQ provides a confirmation mechanism based on acknowledgments. Acknowledgments are messages that are generated automatically by MSMQ in response to events, such as the arrival of a message to a queue or the retrieval of a message from a queue. Acknowledgments are not generated by default; you must request that specific acknowledgments be generated and forwarded to a specific queue. You can also request positive acknowledgments (signifying the successful completion of an operation) or negative acknowledgments (signifying the failure of an operation).

Requesting Message Acknowledgment

To request the acknowledgment of a message, you must first create a queue that will receive the acknowledgment messages. The queue that will accept the acknowledgment messages is a regular queue; there's nothing special about messages sent to this queue, except for the fact that they're generated automatically. To use the acknowledgment queue, you must set certain properties of the message before you send it. These properties are the `AdministrationQueue` property, which determines where the acknowledgment messages are sent, and the `AcknowledgeTypes` property, which indicates what

As you can understand, not all operations can be placed into a transaction. If you're sending a message to a remote queue, you probably don't want to wait for the message to arrive at the destination queue (let alone be processed) before you commit the transaction. This is simply a limitation of loosely coupled systems. Message transactions are not the same as database transactions. However, there are mechanisms to ensure the integrity of messages, such as acknowledgments. In the following section we're going to learn yet another mechanism for message integrity—how to keep copies of the sent messages and process them again if they can't be delivered.

AUDITING MESSAGES

Transactions are invaluable in designing robust systems. Another technique, perhaps not as valuable but very useful, is the logging of messages. In addition to sending a message, you can create a copy of it and send it to a designated queue. If everything else fails, you can at least recover the message that wasn't delivered (or processed) and repeat it. Under each queue in the MSMQ snap-in there are two items: the Queue Messages item, where incoming messages are stored, and the Journal Queue, where MSMQ keeps copies of the outgoing messages. In effect, journal queues are equivalent to acknowledgment queues, in the sense that they keep track of the movement of the messages in a specific queue.

Messages are not copied to the corresponding journal queue by default. To create copies of the messages sent to a specific queue, set the `UseJournalQueue` property of the `MessageQueue` object that represents the queue to `True`. Every message sent to this queue will be copied into its journal queue. Earlier in the chapter we showed you how to retrieve acknowledgment messages from a queue and take appropriate action. A more robust technique is based on a combination of acknowledgment and journal queues. Let's assume that certain messages, such as messages about orders, must be processed within an interval of a few hours to a few days. If the message regarding an order isn't retrieved from its queue within the specified interval, a negative acknowledgment is sent to an acknowledgment queue (the acknowledgment message won't be sent unless the sending application has requested message acknowledgment). In addition, the sending application should also send copies of the messages into a journal queue. Another application can continuously remove messages from the acknowledgment queue (presumably, there won't be many messages in this queue) and retrieve their IDs. For every message that failed to be delivered (or retrieved from its queue), the application can retrieve the associated message from the journal queue and resend it. Alternatively, you can send it to a different queue, or log an error message. You can even send a mail message to an operator at the site that fails to read the messages. The most common scenario for message delivery failures is that the destination queue has been moved to another computer, or the user privileges on the remote computer have been altered.

MSMQ won't remove messages from the journal queues. Instead, you must write code to retrieve the messages of a journal queue using the techniques discussed earlier in this chapter.

Processing Orders with Messages

As you have realized by now, working with queues is fairly straightforward. In this section we'll put together all the information presented so far in the chapter to build a practical application that uses the MSMQ component. This section's example consists of a client application that takes orders and submits them to a specific queue and a server application that retrieves orders from the queue and

Message Queuing Triggers

One might expect that the MessageQueue class would support events. If a MessageQueue object could fire an event every time a new message arrives, we'd be able to write simple code to process messages as soon as they arrive and we wouldn't have to use asynchronous techniques to read messages from a queue. MSMQ doesn't support events, but you can use the Message Queuing Triggering service to process messages as soon as they arrive at a specific queue. This service comes as a component of Windows XP. If you're using Windows 2000, you can download it from the following URL:

<http://www.microsoft.com/windows2000/technologies/communications/msmq/default.asp>

The setup dialog boxes for the version of Message Queuing Triggering for Windows 2000 are a little different than the ones for the XP version, and in this section we'll describe the XP version of the component. You shouldn't have any problem applying your knowledge to Windows 2000, but bear in mind that the various dialog boxes you'll see while setting up triggers are different than the ones shown in the figures of this section.


Setting up a trigger for a specific queue is straightforward and you need not write any code. Actually, you can't set up triggers from within your code, but you should expect a new class in the next version of the .NET Framework that exposes the functionality of the Message Queuing Triggering component. The process of setting up a trigger for a queue involves two items:

Rules A rule determines the conditions under which the trigger will be fired. By default, triggers are fired every time a new message arrives at the queue to which the trigger is attached, but you can request that triggers are fired only when certain conditions are met. For example, you can request that a trigger is fired only if the message's label contains (or doesn't contain) a string, the message's priority is greater (or smaller) than a specific value, and so on.

Actions Actions are programs that process the message that caused the trigger to be fired. There are two types of actions, or equivalently two ways to process a message. You can either start an executable (an EXE file) or a call a method of a COM object (a DLL file). The program is started automatically by the trigger every time the conditions specified by the corresponding rule are met. It's also possible to pass one or more arguments to the program that services the trigger, such as the message's ID or label, the date and time it was sent or has arrived at the queue, and so on.

Once you've created a set of rules and their corresponding actions, you can create triggers. Each trigger is unique to a specific queue, but it can deploy any of the existing rules. New triggers are created by combining one or more rules and assigning the trigger to a specific queue. The same rules can be reused in as many triggers as you need.

Defining Rules

The first step in creating a trigger is to define the rule(s) that will be used by the trigger. Expand the Message Queuing Triggers item in the MSMQ snap-in, right-click the Rules item and select New  Rule. You'll be prompted to enter a name and a description for the rule, shown in Figure 14.8. Let's call our trigger **NWOrderRule**. We'll create a trigger to signal the arrival of a new message at the NWOrders queue.

[Team Fly](#)

 Previous

Next 

To test the application, you must first create the OrderTrigger trigger. This trigger is attached to the NWOrders queue and its "Message processing type" setting should be Peeking. Its rule is the NWOrderRule, which has no conditions (it's fired every time a new message arrives at the NWOrders queue) and its action is the ProcessOrders.exe application. Check the box "Interact with the desktop" on the Rule Action tab of the NWOrderRule rule's property pages, so that the console window will appear on your desktop and you'll be able to interact with it. Figure 14.12 shows the Rule Action tab for the NWOrderRule, as well as the Parameters window.

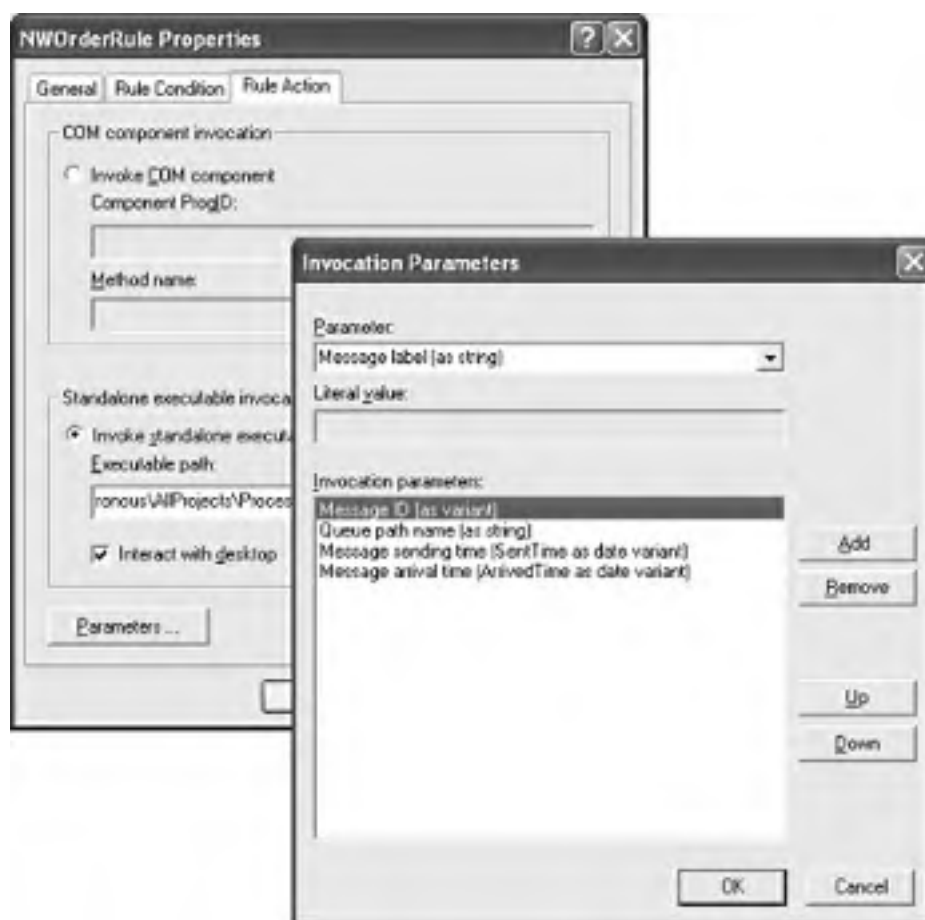


FIGURE 14.12 Setting up the NWOrderRule rule for the OrderTrigger trigger

Assuming that you've set up the OrderTrigger trigger in the MSMQ snap-in, switch to the ProcessOrders project and build it. This project isn't meant to run on its own, because it should be called by a trigger, which is also responsible for passing the appropriate arguments to the application as command-line arguments. Switch to the DisconnectedOrders project and create a new order. As soon as the new order message is written to the NWOrders queue, the OrderTrigger will be fired, which in turn will invoke the ProcessOrders console application. You will see a console window that looks like the window of Figure 14.13. The data shown in this window correspond to those in the order shown on the form of the DisconnectedOrders project of Figure 14.14.

Summary

In this chapter you learned how to build applications running on loosely coupled systems. Large systems may involve computers that are not part of the same local area network, and we can't assume that all the computers we need to access are always online. To pass information from one system to another in a reliable, fail-safe way, we use messages.

[Team Fly](#)

 Previous

Next 

Chapter 15

Practical ADO.NET

ADO.NET, MICROSOFT'S LATEST DATA access technology, is the evolution of ActiveX Data Objects (ADO). ADO.NET was designed from the ground up for distributed architectures, so that it can fit nicely in a networked world. The basic premise is that clients are not constantly connected to a data source. There has to be a convenient mechanism to store the data at the client, process them locally, and submit the updates to the database. This mechanism is the DataSet, which is something like a lightweight, in-memory database. A DataSet is a data structure for storing related tables, and it offers developers a relational view of the data.

ADO.NET is ideal for distributed, disconnected applications, but it's just as good for client/server applications and multi-tier connected applications. A thorough explanation of the architecture of ADO.NET and its classes would require another book, and there are many books on the topic. In this chapter you'll find an overview of the basic classes of ADO.NET and explanations of the techniques we'll use to build a few practical applications in Chapter 18. Our goal is to show you how to write practical data-driven Windows applications with functional, user-friendly interfaces.

As you might expect, Visual Studio .NET supports two approaches for building data-driven applications: a visual approach, which relies heavily on data-binding and wizards, and the programmatic approach. We'll focus on programming the ADO.NET classes. Data binding is convenient for building prototypes, but you can't expect to build professional data-driven applications with data-binding and point-and-click operations.

Accessing Databases

The basic tasks in working with databases are to establish a connection to a database, execute commands against the database, and move data to the client, where they'll be processed. The commands we execute may update some tables in the database and not return any data to the client (except for the number of rows that were affected by the command), or retrieve data and move them to the client. The ADO.NET architecture is based on a few fundamental classes that encapsulate these actions. The Connection class provides the functionality to establish a connection to a database, the Command class provides the functionality to execute a command against the database, and the DataSet class provides a convenient mechanism for storing data at the client. The

The Command Class

The Command class allows you to specify the command you want to execute against the database, set its parameters (if any), and finally execute the command. So far you've seen how the DataAdapter interacts with the database with the commands that were generated by the wizard. The advantage of the DataAdapter is that it knows how to prepare the parameters of each command, execute it against the database, and populate a DataSet with the results. In this section you'll learn how to set up custom commands from within your code and execute them outside the context of the DataAdapter. The commands you can execute against a database are SQL statements (queries) and stored procedures. The CommandText property stores the SQL statement, or the name of the stored procedure, that will be executed against the database. The CommandType property specifies the type of the command, and its value is one of the members of the CommandType enumeration:

StoredProcedure The command is the name of a stored procedure.

Text The command is an SQL statement.

TableDirect The command is the name of a table. When the command is executed, it will retrieve all rows and all columns of the specified table. This member can be used only with the OLE DB data provider. You can retrieve the join of multiple tables by setting the CommandText property to a comma-delimited list of table names.

Once the Command object is configured, you can call one of the following methods to execute the command against the database:

ExecuteNonQuery This method is used to execute action queries; it returns the number of rows affected by the query.

ExecuteScalar This method is used to execute a query and returns the first row of the first column in the resultset. This value is returned by a selection query and it's usually an aggregate value, or the result of some calculations. Note that the method doesn't return the return value of a stored procedure. To read the return value, you must set up an output parameter for the stored procedure's return value.

ExecuteReader This method executes a selection query and returns a DataReader object. This object is similar to a StreamReader, in that you can use its methods to read consecutive rows in the resultset and their columns.

ExecuteXmlReader This method executes a selection query and returns an XmlReader object. The query should return its data in XML format, by using the FOR XML clause of the SELECT statement. The XmlReader allows you to read the elements and attributes of the XML document returned by SQL Server.

As you can see, the Command object doesn't provide any methods for filling DataSets. We use the Command object to execute action queries against the database and retrieve the number of rows that were affected. You can also use its ExecuteReader method to retrieve the results of a selection query and use them to populate a custom structure at the client. Strictly speaking, it's possible to use the ExecuteReader method to populate a DataTable manually, but there's no

reason on earth to do it. The DataAdapter is much more efficient in filling DataTables. The use of the DataReader class with the ExecuteReader method is demonstrated in the section "Using the DataReader," later in this chapter.

This query will return two cursors, one with the selected rows of the Products table and another one with the selected rows of the Suppliers table. While you can't use this batch query to populate a DataSet with two tables (you must set up two different DataAdapters, one for each table, and call their Fill method), you can retrieve the rows from both tables with a DataReader object. You start reading rows as usual. When you reach the end of the first cursor, call the DataReader's NextResult method to move to the following cursor:

```
Do
    While RDR.Read()
        ' statements to process the rows of the current cursor
    End While
While RDR.NextResult() ' skip to next cursor, if there is one
```

The first time through the outer loop, the inner loop reads the rows of the Products table. When the NextResult method is called, you're into the second cursor and the inner loop is executed again, this time reading the rows of the Suppliers table.

Working with DataSets

Now that you know how to contact a database and execute commands against it, we can examine the DataSet object in detail. This is the main object of ADO.NET and this is where most applications store data at the client. The DataSet is made up of DataTable objects; there's one DataTable for each one of the tables involved in the query. The DataTable objects are made up of DataColumn and DataRow objects. The DataColumn objects specify the structure of the table, and the DataRow objects contain the rows of the table. You can also establish relations between the tables in the DataSet; these relations are represented with DataRelation objects.

As you already know, we use DataAdapters to fill DataSets. It's possible to create a DataSet and also fill it from within your code, but this isn't common. Most often, DataSets are generated at design-time and populated at runtime with the DataAdapter's Fill method. These DataSets are called typed DataSets, because they know about the structure of the data they store. Notice that only DataSets created at design time are typed. This happens because the IDE generates a class behind your back to encapsulate the data and the basic operations you can perform on the data. A DataSet generated at runtime won't expose the names of its tables as properties and it won't provide methods that are specific to your data (such as the FindByCustomerID method, or the IsUnitPriceNull property) for example).

By the way, if you want to see the code of the class that implements a typed DataSet, click the Show All Files button at the top of the Solution Explorer window and then expand the file named after the DataSet (its extension is XSD). Under this file you'll see two more files, which are normally hidden, and they're both named after the DataSet. Double-click the file with extension VB and you'll see the code of the class that implements the typed DataSet. Figure 15.9 shows a section of the DSPProducts.vb class, which is part of the NWProducts project. As you can see, there's nothing complicated or magic about typed DataSets. They're implemented with code that some of us might have written to simplify the process of coding large projects, only this code was generated by a wizard. Normally, you'll never have to see the auto-

generated code, unless you want to add a few members that are specific to an application. Even so, you should implement these members in a separate class, because every time you redesign the DataAdapter and re-generate the DataSet, this class is auto-generated.

[Team Fly](#)

 Previous

Next 

Insert and Update Operations

One of the most important topics in database programming is the commitment of the changes made at the client back to the database. The changes involve edited rows, which must update the underlying rows in the table, new rows, which must be inserted into the underlying table, and deletions, which must remove the corresponding rows from the underlying table. There are basically two modes of operation: single updates and multiple updates. A client application running on a local area network as the database server can (and should) submit changes as soon as they occur. If the client application is not connected to the database server at all times, then changes may accumulate at the client and be submitted in batch mode when a connection to the server is available.

From a developer's point of view, the difference between the two modes is how you'll handle update errors. If you submit individual rows to the database and the update operation fails, you can display a warning and let the user edit the data again. You can write code to restore the row to its original state, or not. In any case, it's fairly easy to handle isolated errors. If the application submits a few dozen rows to the database, several of these rows may fail to update the underlying table and you'll have to handle the update errors from within your code. At the very least, you must validate the data as best as you can at the client before submitting them to the database. No matter how thoroughly you validate your data, you can't be sure that they will be inserted into the database successfully.

Another factor you should consider is the nature of the data you work with. Let's consider an application that maintains a database of books and an application that takes orders. The book maintenance application handles publishers, authors, translators, and other data. All users who are entering and correcting titles are working with the same table of authors. If you allow them to work in disconnected mode, the same author name may be entered several times, as no user can see the changes made by any other user. The result is that several rows in the Authors table refer to the same author. This application should be connected: every time a user adds a new author, the table with the author names in the database must be updated, so that other users can see the new author. The same goes for publishers, translators, topics, and so on.

The order-taking application can safely work in a disconnected mode, because orders entered by one user are not aware of, and they don't interfere with, the orders entered by another user. You can install the client application on the notebooks of several salespersons so they can take orders on the go and upload them when they establish a connection between their notebook and the database server (which may happen when they return to the company's offices). There's a small implication here, namely the stock. If you can't make a sale unless the items are in stock, things get quite complicated; the order-taking application can't run in disconnected mode. Incidentally, this is one of the most complicated types of projects you may run into and we will not discuss it in this book. The solution is dictated by the business rules and, in most cases, it's non-trivial.

The order-taking application can be used in a disconnected mode, because each order contains existing products and there will be no update errors. The worst that can happen is that a product's price will change. In this case, a business rule determines whether the sale is made with the old price, or whether the customer should be contacted and confirm the revised price.

Updating the Database with the DataAdapter

The simplest method of submitting changes to the database is to use each DataAdapter's Update method. At the beginning of the chapter we discussed how to submit changes to the database and

Of course, a real application shouldn't allow the user to enter invalid data in the first place. We're going to build a practical interface for entering orders and invoices in Chapter 18. The example we just finished was merely meant to demonstrate the basic principles of performing multiple updates in the context of a transaction using a DataAdapter.

Summary

In this chapter, which is one of the longest ones on the book, we've explored the basic objects of ADO.NET. The Connection object establishes a connection to the database, through which we can submit and execute commands against the database. The commands to be executed against the database are represented by Command objects. A Command object is assigned the SQL query to execute against the database, as well as the necessary parameters. To actually execute the query, you must call one of the Execute methods of the Command object. Action queries return a single value, which is the number of rows affected by the query. Selection queries return a DataReader object, which you can use to read the values retrieved from the database serially.

You can also use DataAdapter objects, which move data into a client DataSet. Most of the applications you'll write will make use of the DataSet object, which can store sections of database tables and maintain relations between them. The DataSet knows how to submit changes to the database, and you can use it at the client as an in-memory database. As you have seen, DataAdapters and DataSets are classes generated for you at design time. These two classes expose most of the functionality you need for typical business applications. You have also seen how to use these classes to perform the basic data operations, from retrieving a table's rows to performing transactional updates.

In this chapter we used the DataGrid control to view and edit our data. Practical applications aren't built around the DataGrid control, however. They use interfaces based on regular Windows controls and they contain quite a bit of code. In Chapter 18 you're going to see several examples of practical user interfaces, which are based on Windows controls and make use of the objects discussed in this chapter.

Chapter 16

Building Middle-Tier Components

IN THIS CHAPTER WE'LL discuss the concepts of distributed architecture and the role of components and multiple tiers in developing data-driven applications. We'll focus on the drift from client/server architectures to multi-tier architectures and we'll present a few simple examples to demonstrate the principles of middle-tier components and how to deploy them on a remote server.

We'll also discuss how to use existing COM components (including ActiveX controls) and COM+ applications with .NET clients. Every corporation has made an investment in COM components, which you aren't going to throw away. Whether these components you've developed as recently as a year ago can be called "legacy" components is a different story.

Finally, we'll show you how to deploy .NET components on remote servers and allow clients to request their services over the network, or the Web. We'll touch the subjects of Web services and remoting, which are central in deploying business components in a distributed environment.

From Client/Server to Multiple Tiers

The dominant architectural model for data-driven applications today is the client/server model. Most applications written today in small business environments are based on the client/server model and most VB6 developers are quite familiar with it. The client/server model distributes the processing on two layers: the database, which is a powerful machine running the database management system, and the clients (Figure 16.1). The program running on the client is responsible for interacting with the user: it accepts user input, validates it, and makes requests to the server. When the server sends the data, the client application presents it to the user, using a rich Windows interface.

The advantage of this architecture is that the workload is distributed on two different layers. The database server is optimized for performing queries against a database. The basic requirements of the database server are very fast disk systems (usually RAID systems) and large (to enormous) amounts of memory. The clients need not be nearly as powerful: a regular workstation will do. The application running on the client is either a Windows application (in which case the client is called rich client, to indicate that the client application can exploit all the resources of the client

You may notice that all the work is done by the `GetItemDiscount` stored procedure. This means that we can change our discount policy by editing the stored procedure at the database server, without even looking at the middle-tier component's code. This is an added bonus for the specific application, but you can't count on this. A complicated business rule may require quite a bit of code in the middle tier, and you can't always implement business rules at the database level. When this is possible, you can simplify deployment even further (no need to touch the application's code, just change the stored procedure at the database). However, you shouldn't place an additional burden on the server just to avoid the deployment of a new component. In our example, the rule requires the execution of a non-trivial query against the database and we can't avoid it.

Remoting the Business Logic

As we mentioned, one of the benefits of building components, especially in large applications, is that we can deploy them on a single server (or a small number of servers) and service a large number of clients. The servers on which the components reside are called application servers and they're fast machines, usually connected to a database server through a high-speed link. The application server is where the business logic is executed. If your corporation changes a business rule, you can revise one or more components and install them on the application server, and all clients will see the new component the next time they request it. It's not uncommon for the application and database servers to be hosted on the same machine. When the system is overloaded, you can add application and database servers as needed. It's not a trivial task, but this is the way to build a highly-scalable application. Scalable applications are written so that they can be spread over multiple servers. You can add multiple application servers to scale out the business components, as well as database servers to scale out the database. With some form of load-balancing software, client requests are directed to the least loaded server at the time.

Another good reason for building components that can be executed remotely is that not all components may reside on your corporation's servers. Today's applications may need to work with resources outside a corporation's environment. Consider a Web application that accepts orders and calculates shipping costs. To calculate the shipping cost, you may have to connect to the database of the shipping company and interact with a component that can calculate the cost of a shipment, given its source and destination, and the weight of the goods to be shipped. You may also wish to display the progress of the shipping, if this information is available from the shipping company. Many online stores display the progress of the shipment online. This information comes from the shipping company, not from the merchant's database. The merchant's system requests this information from the shipper and displays it for its own customers.

Given the need for remoting components and different systems to talk to one another, we'll explore .NET's techniques for invoking components on remote systems. The two techniques are Web services and remoting. Web services are simple to set up; remoting can be substantially more difficult, but ultimately faster and more flexible. Web services are actually based on remoting, but they hide many of the underlying details.

First, we're going to host the business component on the web server and expose it as a Web service. All the clients on the network will call the Web service to retrieve the discount. Then we'll use remoting to access the middle-tier component on an application server.

[Team Fly](#)

 Previous

Next 

This is how you can use a COM component in your .NET applications, regardless of whether it's a legacy component you developed with VB6 or a third-party control. The source code is not required and you don't have to create the interoperability layer yourself; instead, the IDE will create it as needed and will also copy it to the project's output directory. This means that interop assemblies will be distributed with the project's setup application.

***TIP** The CLR can determine the dependencies of your code on COM components and include them automatically in the project's output when you build the application's setup program (the MSI package, as discussed in Chapter 10). However, it can't determine the dependencies of the COM component. If the COM component has dependencies of its own, you must add them to the project's output manually.*

Using COM+ Applications in .NET

In many situations, middle-tier components are distributed to the production environment as COM+ applications. A COM+ application runs as a Component Service: that is, the component is hosted by the Component Services on an application server and any number of client machines can contact the server and request an instance of the component. There are many benefits to this approach, especially for large-scale applications. COM+ applications can be used to create objects that run on the application server, not on the client. They can participate in transactions and, most importantly, you can maintain a pool of objects on the server to service a large number of clients.

Let's start by reviewing the process of creating a COM+ application with VB6. First, you create a class as usual. The class's compiled code, which is a DLL, can be installed on an application server as a COM+ application. This is done with the help of the Component Services Explorer (or the Component Services console). You simply create a new COM+ application and add the new component (the DLL) to the application. The COM+ application can be configured easily from within the Component Services Explorer. You can specify whether the component will run at the server or the client, whether the component will participate in transactions, and whether the component can be pooled. You can also determine who can access the component. All these operations take place through a point-and-click interface, and you don't have to modify the DLL's code or the application that uses it.

Once the COM+ application has been installed and configured, you create a proxy for the clients. The Component Services Explorer will export the application into an MSI package, which you distribute and install at the clients. The MSI package will install a proxy for the actual COM+ component at the client. The proxy appears at the client as a new COM+ application and any application on the client can contact it. However, you can't set the component's properties through the proxy.

What does it take to use an existing COM+ application in a .NET client application? Basically, once the proxy has been installed on the client machine, you can add a reference to the COM+ proxy as we demonstrated in the preceding section. The IDE will create an interop assembly, which will be included in the .NET project. Let's look at the process by building a simple application. For the example of the following section we'll assume that you have VB6 installed on your system, which you'll use to create a COM component. If not, you can copy the sample DLL that's included in the zip file with this chapter's projects.

Run the application and click the button. Ten seconds later you will see a message box with your computer's name. After that, the message box will pop up within a second after you click the same button. The object is pooled and need not be created again. If you remove the ObjectPooling attribute from the class's definition, recompile the component, and run the test project again, every time you click the button you'll wait for 10 seconds before seeing the message box on your desktop. Notice also that regardless of whether the component runs as a server or library application, the objects you create are pooled automatically.

Summary

In this chapter we discussed the importance of the middle-tier component in designing scalable applications. Middle-tier components are used commonly with large applications, because they simplify the tasks of maintaining and deploying the revised applications. Reinstalling a client application to a large number of workstations can be quite a task and any technique to simplify this task is welcome.

The maintenance of a large application is also greatly simplified if the application is built with components, because the middle tier can be revised independently of the presentation tier and be deployed on selected servers rather than on every client. If you choose to post the middle tier on a web server and expose its functionality through Web services, you can build both Windows and web clients that exploit the functionality of the middle tier to access the database, regardless of the actual location of the database. The clients will never interact with the database directly, and the objects of the middle-tier component abstract the view of the database at the presentation tier's level.

Finally, you learned how to use existing COM components with .NET clients, as well as how to develop serviced components in .NET. Serviced components provide the functionality of COM objects, such as automatic transactions, role-based security, and object pooling, which is new to VB developers.

Chapter 17

Exploring XML Techniques

IT'S SAID THAT .NET rests on XML, meaning that in the .NET world, XML is the data storage and transmission technology of choice. The XML classes built into the .NET Framework are gathered into these primary categories:

XmlDocument for editing XML (part of Microsoft's implementation of DOM)

XmlReader for reading and searching (part of Microsoft's SAX)

XmlWriter for saving

XmlSchema creating and managing XSD schemas

XmlValidatingReader validation

XmlTransform executing XSL transformations

XpathNavigator applying Xpath queries

There are several auxiliary technologies that expand and assist XML. For instance, XML, like HTML, can use Cascading Style Sheets, or the even more advanced styles technology called XSL, which can reorder, or add and delete, tags and attributes.

XML rests on two main APIs:

- ◆ DOM (Document Object Model)
- ◆ SAX (Simple API for XML)

Each has its uses. DOM needs an entire XML document to sit in memory while DOM processes it (DOM is therefore capable of random-access processing). DOM is preferred for editing XML. SAX works serially on an XML stream, and is preferred for reading or searching.

This chapter covers a variety of XML tools and features that every .NET programmer needs to understand, exploring the first four of the primary categories listed at the opening of this chapter.

Choosing SAX

DOM can be used right along with SAX, if you wish, but normally you select the set of tools appropriate to the job at hand. SAX is best for searching, particularly simple searches or when large documents are involved. DOM is best for tasks involving document modification, or when the task is complex (for instance, when you are dealing with internal cross-reference structures such as ID and IDREF).

DOM builds a verbose, navigable tree structure in memory—it even adds a type description for each node. If you're merely interested in reading through a large XML document, you normally wouldn't want to hand it over to DOM. But do remember that you can use the technologies together. You could emit a SAX stream from a DOM tree, or ask SAX to build a DOM tree.

Let's take a look at how SAX works first because it's the simpler of the two technologies.

Copying the Sample File

Before going further, you should now copy a sample XML file to your C:\ drive. Several examples in this chapter require this XML document.

As with the famous Pubs and Northwind sample databases, Microsoft offers a sample XML file for you to experiment with. Included with Visual Studio is a file in the Help system you should now save to your hard drive. Save it in a file named `books.xml` on your C:\ drive. Use the VB.NET Help Search feature (click the Search tab at the bottom of the Help window) and search for:

```
<author>Gambardella, Matthew</author>
```

You see an entry titled Sample HTML File For XML Data Islands. Copy it to Notepad. The data you want begins with the usual ?:

```
<?xml version='1.0'?>  
<catalog>
```

and it ends like this:

```
</book>  
</catalog>
```

Using SAX

To see how to access a SAX stream in .NET, start a new VB.NET Windows-style project, then type in the code in Listing 17.1.

LISTING 17.1: ACCESSING A SAX STREAM

```
Imports System.Xml
```

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

```
    Dim Xreader = New XmlTextReader("c:\books.xml")  
    Dim ele, att As Integer, m As String
```

```
    While Xreader.Read()
```

[Team Fly](#)

 Previous

Next 

NodeType The type of the current node (using the `XmlNodeType` constants —see "Xml-NodeType enumeration" in Visual Studio Help).

Prefix The namespace prefix of the node.

Value The text value of the node.

Note that the `XmlTextReader`'s `Read` method maintains a pointer within the streaming document, keeping track as the nodes flow by, leaping from node to node. At each leap, you have the opportunity to query the current node, using the Reader's properties and methods, such as the `HasAttributes` property used in this example.

You can stream XML in from a variety of sources, including a URL, like this:

```
Dim Xreader = New XmlTextReader("http://www.myplace.xml")
```

If you are one of those who are simply determined to avoid Microsoft technology, you will want to find and use other versions of SAX (and what are you doing reading this book?).

A primary distinction between classic SAX parsing and the Microsoft `XMLReader` is that SAX *pushes* the events into your source code, meaning that you are notified each time a node is read by the parser. The `XMLReader` *pulls* the XML in, offering you a bit more flexibility. For one thing, with the `XMLReader` you can rather painlessly access multiple input streams. Another signal advantage of the XML Reader is that it includes `Skip` and `MoveToContent` methods, so you can locate nodes of interest to you more quickly. It's similar to random-access, albeit forward-only.

Deeper into DOM

The DOM is a way for programs to read and write to XML, adjusting the style, content, and structure of the XML file.

The original DOM specification is not itself a library of functions. It's merely a collection of *interfaces*. Interfaces are often used when a committee is concerned that a set of class and member names be standardized and enforced. XML DOM is a list of words, and you (or any other programmer) can create the source code that actually makes the interfaces do their jobs. However, I suspect you'll simply want to join the crowd and use Microsoft's version of the DOM.

The Microsoft .NET implementation of the DOM specification closely follows the official W3C DOM interfaces. The .NET `XmlNodeList` and `XMLDocument` classes—and related classes—offer both the fundamental and extended technologies specified by W3C. If you've worked with W3C DOM, you'll find its .NET implementation very familiar and easy to use. What's more, you will surely appreciate the additional features available in .NET that make working with XML both easier and less error-prone.

The DOM can be viewed as an interface to the many proprietary APIs and XML data structures, making it possible for a programmer to work with standard DOM interfaces rather than having to study proprietary APIs.

For example, Ford and GM may use different APIs to handle their XML needs, but with DOM, a programmer can move from GM to Ford and still count on a known, abstract interface that will work with either the Ford or GM APIs. In other words, DOM is a linguistic convention.

Also supporting XML are XML schemas, which assist programmers in defining their own, proprietary XML structures. Schemas, including one proposed by Microsoft, ultimately go beyond

Using Namespaces in XML

XML namespaces help prevent "collisions" that can happen when attribute names or tags are identical if a document contains multiple markup vocabularies (more than one namespace). They are similar to .NET namespaces. This works in XML because each namespace is given a unique number. Commonly, a different URL (Uniform Resource Identifier) is assigned to each namespace. By definition and design URLs are unique—there's only one possible number for each URL anywhere in the world of the Internet. Sometimes a URN (Uniform Resource Number) is used instead. In either case, the number is unique and prevents collisions of the names you use in different vocabularies.

You can either come right out and explicitly name an XML namespace within your XML code, or you can allow the parser to assume the namespace implicitly, by omitting it from your code.

Explicit Declaration

Just as with variables in VB6 and earlier, you can either explicitly declare an XML namespace or let it happen *implicitly*. Explicit declarations keep things straight if your node contains elements from more than one namespace. You use a shorthand name for the namespace that you use as a prefix (like an alias) to specify which namespace an element belongs to.

```
<mo:film xmlns:mo="urn:FilmSociety.com:FilmData"
  xmlns:directors="urn:CinemaHistory.com:Directors">
  <mo:name>Annie Hall</mo:name>
  <directors:director>Woody Allen</directors:director>
</mo:film>
```

`xmlns` is the attribute used to declare a namespace and at the same time to specify a prefix that represents the namespace. In the example above, we defined a prefix (*mo*) that represents the namespace identified by the unique value of "urn:FilmSociety.com:FilmData" and also declared a second namespace ("urn:CinemaHistory.com:Directors") and assigned the word *directors* as its prefix. Then the *mo* prefix indicates that the *name* element belongs to the "urn:FilmSociety.com:FilmData" namespace.

Next the *directors* prefix specifies that the *director* element belongs to the "urn:CinemaHistory.com:Directors" namespace. In this way, you can freely employ elements from different namespaces, and not have to worry that you'll run into duplicate (therefore ambiguous) element names. With the prefixes, there will never be confusion if more than one element has the same name.

Implicit Declaration

Implicit declaration means that all the elements inside the element's scope belong to the same namespace (so a prefix is not needed). You accomplish explicit declaration by simply leaving out the prefix when you declare the namespace, like this:

```
<film xmlns="urn:FilmSociety.com:FilmData">  
  <name>Annie Hall</name>  
  <star>Diane Keaton</star>  
</film>
```

[Team Fly](#)

 Previous

Next 

The Explosion of Schemes

As you probably guessed, the extensibility of XML is a two-edged sword. Allowing everyone to create their own tag vocabulary (element and attribute names) and data structures has resulted in many thousands of unique, proprietary XML vocabularies.

In the early 80s, an intriguing language named Forth fascinated many programmers. It permitted a crude form of inheritance and polymorphism.

Using Forth was similar to Lego and transformer toys where a robot can be changed into a truck, and a truck can be built up until it becomes a city. Essentially, the Forth language was open and protean: You took the core language and modified it until it transformed into an application. Each Forth application was merely the core language itself, but renovated and expanded until it became functional, specialized, and unique.

The problem was that each application contained many unique statements that only the programmer could understand (if even he or she could figure it out after a few weeks passed). Also, programmers tended to quickly customize the language in other ways, creating their own personal (and incompatible) version of string manipulation, data shorthand, and other language components.

Linux aficionados call this effect *forking*. By this they mean that an IT department can lose control of a Linux-based project because it's all too easy to create forks in the code base. Precisely because the central source code is open to anyone's fiddling, the fundamental core (code base) of Linux can divide into incompatible code bases which cannot ever be reconciled.

This effect is not accidental or rare. Forking becomes a tree of forks rather rapidly. Indeed, forking always seems to happen to languages such as Forth, and operating systems such as BSD (now we have multiple forks: NetBSD, FreeBSD, OpenBSD and so on). XML (supposedly *standardized*) itself is forking rapidly into incompatible versions for bankers, bakers, and candlestick makers. Efforts are made to enforce conformity on extensible languages (there are XML standards committees; Linus Torvald's and Alan Cox's attempt to act as a central authority for adding and accessing the Linux kernel, and so on). Nonetheless, these efforts at keeping *open* source languages and platforms *closed* are oxymoronic. They have always failed in the past. Despite heavy breathing on the part of the techie crowd, and quite a bit of positive publicity in the press, Forth rapidly disappeared.

Every organization left to build its own set of XML structures and tags generates a new XML language, unique to itself. All these new languages share the XML punctuation and syntax rules (in effect, they share the XML interface), but the actual vocabulary is special to each implementation.

How you navigate unique XML structures, what the tags mean, the hierarchy, the relationships, the diction—all this can differ among the many thousands of versions of XML schema currently being invented by disparate organizations.

Microsoft, and others, have proposed sets of rules, schemata. One such initiative is Microsoft's *biztalk*, a site that attempts to gather information about XML, XSL and other data models used by all those thousands of organizations. See www.biztalk.org for further details.

Now let's turn our attention to XSD, Microsoft's choice for the building blocks for schemas. As you'll see, XSD is uniquely suited to representing data sets, and to translating database tables into XML and vice versa.

Understanding XSD

Visual Studio.NET focuses on XSD rather than DTD or other alternatives. So we'll take a brief look at what you can do with XSD.

[Team Fly](#)

 Previous

Next 

Programmatic XML

In .NET, an XML document can be loaded using the Load method (you pass an argument describing the source, which can be a disk file, a stream, an XMLReader, or a TextReader object). Or you can use the LoadXML method to load a literal string, or string variable, into your document.

Start a new VB.NET Windows project, and add these namespaces:

```
Imports System.Xml
Imports System.Xml.Xsl
```

To see how to create an XML document, then load a literal string into it, type in the code in Listing 17.3.

LISTING 17:3 LOADING A LITERAL STRING INTO AN XML DOCUMENT

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

    Dim XMLdoc As XmlDocument

    Try

        XMLdoc = New XmlDocument
        XMLdoc.LoadXml('<' <Cookie>
<Name>Francine Cerance</Name></Cookie>")
        Console.WriteLine(XMLdoc.DocumentElement.OuterXml)
        Console.WriteLine(XMLdoc.DocumentElement.InnerXml)

    Catch ex As Exception
        MsgBox(ex.Message)
    End Try

End Sub
```

Here's the result when this code is executed:

```
<Cookie><Name>Francine Cerance</Name></Cookie>
<Name>Francine Cerance</Name>
```

To see how to load XML from a file, make the following change to the previous example. Change LoadXml to Load, and replace the string argument with the path to an XML file:

```
XMLdoc.Load("c:\books.xml")
```

Press F5 and you'll see two long lines of data in the Output window. The only difference between these lines is that the first line includes the <catalog> tags because it displays the outer XML.



FIGURE 17.1 The program atomizes an XML document into its smallest components.

XML and DataSets

Among the most useful aspects of XML in .NET is its interchangeability with DataSets. In this section you see how to create an XML schema that becomes a DataSet, then connect it to a DataGrid, and save or load this schema and the associated data.

Start a new VB.NET Windows-style project. Choose Project ➤ Add New Item, then double-click the XML Schema icon in the dialog box. The Toolbox is now filled with the Tinkertoys you can use to build a schema. Double-click the Element icon in the Toolbox. A graphic appears in the design window, ready for you to define the structure (add attributes, for example). Each element in this kind of XML schema is the equivalent of a table in a database or DataSet.

Add several attributes to your element by dragging attribute icons from the Toolbox and dropping them into the element box graphic. You can adjust the data type in the right column of the element box by clicking, then dropping a list of available types. You can rename the attributes by clicking them, then typing.

Right-click the background of the design window and choose Generate DataSet from the context menu. Now click the Form1.vb [Design] tab to display your project's form. Click the Data tab on the Toolbox and double-click the DataSet icon. The Add DataSet dialog box opens with the name of your schema already displayed in the dialog box by default.

Click the Windows Forms tab on the Toolbox and add a DataGrid by double-clicking its icon. If you haven't experimented with it, the DataGrid is an excellent, flexible user-interface device for database work. In the Properties window, set the DataGrid's DataSource property to XmlSchema11.element1.

Now add some source code to permit you to persist and retrieve DataSets stored as XML files. Add two buttons to your form, then type Listing 17.7 in.

[Team Fly](#)

 Previous

Next 

Then click the Save button and take a look at the file that VB.NET has saved:

```
<XMLSchema1 xmlns=''http://tempuri.org/XMLSchema1.xsd">
  <element1 FirstName="Danny" attribute2= "Prior" attribute3= "12" attribute4=
  <element1 FirstName="Hoda"
attribute2="Macksoof" attribute3= "55" attribute4="-4" />
  <element1 FirstName="Soledaa" attribute2="Nussy"
attribute3="-2" attribute4="-5" />
</XMLSchema1>
```

This isn't your father's database. Stop the program, then run it again and click the Load button to populate the DataGrid.

Persisting with SOAP

Persisting arrays and collections (and arraylists, hashtables, what have you)—as well as objects, structures and so on—can be conveniently handled by an XML daughter technology, SOAP. You use the SoapFormatter to create or load an XML file that contains both the structure of the collection and its data. This technique draws upon the .NET serialization capability and is illustrated in Listing 17.8.

***TIP** XML serialization ignores any private fields (binary serialization saves both private and public fields).*

***WARNING** To use XML serialization, you must choose Project ➤ Add Reference, then scroll down the list of "components" and add System.Runtime.Serialization.Formatter.S Soap to your project. Oddly, some namespaces (mercifully only a few) must be added in this way—as a "reference"—to a project rather than employing the usual Imports statement. The distinction between which namespaces are imported and which assemblies must be added as "references" escapes me. It perhaps reveals which technologies were added to .NET late in the game. After you finish this step, you'll see the System. Runtime.Serialization.Formatter.S Soap reference in the Solution Explorer, and you're ready to roll.*

LISTING 17.8: PERSISTING TYPES VIA XML SERIALIZATION

```
Imports System.IO
Imports System.Runtime.Serialization.Formatter
Imports System.Runtime.Serialization.Formatter.Binary

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

    Dim Arr(2), ArrNew(2) As String

    Arr(0) = "This test"
    Arr(1) = "continues until"
    Arr(2) = "it finishes."
```



```
        Console.WriteLine(c1.FirstName)
        Console.WriteLine(c1.LastName)
    End Sub
    Sub deser_UnknownElement(ByVal sender As Object, ByVal e As XmlElementEventArgs
        MsgBox(e.Element.Name & ' ' is not recognized. Deal with it. The prob
    in line number " & _
    e.LineNumber & " in the XML source file.")
    End Sub
```

Notice that when you execute this program, the Output window displays the LastName, but not the FirstName (which was ignored in the input stream). Also, you can use the ObjectBeing-Deserialized property of the XmlElementEventArgs object to identify which object instantiated by your application is having the problem.

More Interchangeability

As you see in various examples in this chapter, XML is pervasive in .NET. You can quickly design an XML schema, then transform it into an ADO.NET DataSet structure with a click of the mouse (see the section titled "XML and DataSets" earlier in this chapter).

In various other contexts in .NET you also see XML popping up here and there—and often you need not do anything yourself because .NET creates the XML or schemas all by itself.

How easy is it to transform XML data (as opposed to a schema) into a database-style data table? As easy as clicking a Data tab in the VB.NET IDE. Try it.

Choose File ➤ Open ➤ File in VB.NET. Locate and load the sample XML file, `c:\books.xml`. You see a formatted view of the XML. Now click the Data tab at the bottom of the .NET IDE window. The XML is automatically translated into an editable data table, as shown in Figure 17.3:





FIGURE 17.3 .NET transforms XML into a DataSet with the click of a button.

Finally, you learned how to employ serialization to persist data two ways: via SOAP and via simpler, more streamlined, XMLSerializer techniques. And you saw the uses, and limitations, of self-description during deserialization.

[Team Fly](#)

 Previous

Next 

Chapter 18

Designing Data-Driven Windows Applications

IN CHAPTER 15 WE discussed the architecture of ADO.NET and the classes that make up ADO.NET. So far we've shown you simple examples to demonstrate the basic operations you can perform with the ADO.NET objects, and the interfaces of these sample applications were based on the DataGrid control. The DataGrid control is not the be-all-and-end-all of your data display requirements. For one thing, it's almost impossible to edit a DataGrid control without reaching for the mouse, and a basic requirement for many applications is that they be used with the keyboard only. Another limitation of the DataGrid control is that it doesn't support the functionality of the ComboBox control, which is frequently used as a lookup tool. If you search the Internet for tips on using Windows controls, you'll find numerous resources on adding functionality to the DataGrid control. Our tip is to not use a control for a purpose for which it wasn't designed.

Real-world applications are based on interfaces built with regular Windows controls—data-binding is not a developer's first choice. In this section you'll learn how to build functional, userfriendly applications with regular Windows controls. You'll also learn how to build navigational tools that allow users to quickly locate the desired rows. A navigational tool based on a couple of buttons that take the user to the next or previous row is simply unacceptable. In this chapter we'll describe a couple of functional navigational models that you can use with your applications.

We'll start this chapter with a quick overview of data-binding and then we'll present a few typical data-driven applications. The applications of this chapter contain quite a bit of code, but these aren't simple applications. We'll explain their architecture and then we'll look at the code. We suggest that you download the projects from the book's website and open them in the Visual Studio IDE. The projects are well documented and you'll find it easy to understand their code.

Data-Binding

Data-binding is a mechanism for mapping selected columns of a DataTable to a control property, which is usually the Text property. When a control is bound to a column, the column's value is displayed automatically on the control. As you move through the rows of the DataTable, the property changes value to reflect the value of the bound column in the current row. If the control's text is

edited, the new value replaces the column's value in the DataSet. A data-bound control is in effect a window for viewing and editing a specific column in the DataTable.

In most cases we bind the control's Text property, but there are other properties you can bind to a data source, such as the Tag property. To set the data-bound properties of a control, expand its Data-Bindings section in the Property Browser, select one of the data-bound properties, and set it to the appropriate column name. You will notice that the properties listed in the DataBindings section do not have unique names; you will see a Text property, a Tag property, and so on. In the DataBindings section, you bind the values of these properties to a data field. The properties by the same name that appear outside the DataBindings section can be set to static values as usual.

Some of the Windows controls can be bound to columns and display all the rows in the table. The ListBox and ComboBox controls, for example, can be populated with the rows of a table and display a specific column. These controls are used almost exclusively as lookup tools on data entry forms, and you'll see several examples of this technique in the following sections. It's possible to populate a ComboBox control with the rows of the Categories table, for example and bind the control to the CategoryID field of the Products table. As a result, every time you move to another row in the Products table, the current product's category name will appear on the control. To change the category of the current row, you simply select another category name on the control.

Data-binding is not new to ADO.NET. Data-binding was available with earlier versions of ADO, but it has never been a real developer's tool. You will see how to use data-binding to build a functional viewing and editing interface for the Products table, but most of the examples aren't based on data-binding. We'll discuss the relevant topics as we go through the examples.

The NWProducts Application

Our first example is a typical application for viewing and editing a table with products. You've seen how to display the Northwind Products table's rows on a DataGrid control and how to submit to the database the changes made to these rows by the user with the DataAdapter object. In this chapter we're going to build a functional and intuitive interface for viewing and editing the rows of the Products table.

The basic requirement of this application is that we shouldn't have to download the entire table to the client: in a production database the Products table is quite large. This application will be used by multiple users on a local area network, and they should be able to see each other's changes. If we give each user a copy of the table, multiple users may edit the same line. Moreover, there will be a lot of conflicts we'd have to reconcile as we submit the changes to the database. We'll build a connected application that submits the changes to the database as soon as they occur.

Another important aspect of the application is the navigational model. Even if the rows of the Products table are copied to a DataSet at the client, we should provide a mechanism for users to locate a desired product quickly and conveniently. Dumping all the product names on a ListBox control may work with a small table, but it's a totally impractical approach for a production database with many thousands of products. Moreover, we'll allow users to locate a product with several criteria, and not just the product's name. We'll allow users to select a product by its name, its category, or its supplier, on a separate form.

The Products table is related to the Categories and Suppliers tables. Obviously, we can't display category and supplier IDs on our interface; we must retrieve and display each product's category and

supplier name on the form. When users edit a product, they should be able to select a category and a supplier by name from a list. To facilitate this operation, we'll download the tables with the categories and suppliers to the client. We're assuming that these two tables aren't edited frequently and users will almost always find the desired category and supplier in a DataSet at the local machine.

The Application's Interface

The application's main form is shown in Figure 18.1. Users can select the product to view or edit on the form shown in Figure 18.2. This form is invoked when users click the button with the question mark on the application's main form. On the Product Search Form users can select a product by name, by category, or by supplier. To view the products in a given category, or the products by a given supplier, they can simply select the desired category or supplier on the appropriate ComboBox control. Every time users select another item on either list, the corresponding products are displayed on a ListView control at the bottom of the form. To select products by name they must enter part of the product's name in the top TextBox control and press Enter.

The matching products are displayed on a ListView control at the lower half of the form, where users can double-click a product's name, or press Enter, to view the selected product's details on the program's main form. The auxiliary form will close and the selected product's fields will be displayed on the main form. While viewing products, the controls on the main form are locked. The Edit button unlocks the controls for editing and it also changes the background color of the various controls on the form to indicate that the controls can be edited. The editing process must end with the OK or Cancel button. While editing, users are not allowed to select another product.



FIGURE 18.1 Editing the Products table with the NWProducts application



FIGURE 18.2 Locating the product to view or edit with the NWProducts application

first instance of the application and try to retrieve the new product. You can search for it by name, category, or supplier. If the new category doesn't appear in the ComboBox of the auxiliary form, close the auxiliary form and open it again. The application will fail to load the selected row, because it violates the DataSet's referential integrity. This will activate the Catch clause of the structured exception handler shown in Listing 18.3, and the application will silently reload the Categories and Suppliers DataTables.

You can't delete any products from the database, because they're all referenced by one or more rows of the Order Details table. You will be able to delete a row in the DataSet, but the update operation will fail. Start two instances of the application and add a new product using one of two windows on the desktop. Then select this product in both instances of the application. Delete the product in one instance of the application, and then edit it in another instance of the application. The deletion will succeed, but the edit operation will fail, because the application can't find the row in the Products table and update it. When you click the OK button for the first time, the changes will be submitted to the database. When you click the OK button of the other instance of the application, the update operation will fail. You must cancel the edits and reload the same product row to see the current values of the row in the database. This happens because the DataAdapters were configured for optimistic concurrency. If you turn off optimistic concurrency, then all changes will be written to the database, overwriting changes made by other users.

An Invoicing Application

A very common task in business applications is a program for entering orders and invoices. All invoicing applications are based on a grid control, where the user can enter, as well as edit, the items. If you attempt to build an interface for an invoicing application with the DataGrid control, you'll end up writing a lot of code to add functionality that's not natively supported by the control. A basic requirement of an invoicing application is that it should provide full keyboard support. Can you imagine a cashier at the Wal-Mart using the mouse? Of course, Wal-Mart doesn't use Windows workstations at their registers, and there's a very good reason for this: they want the lines to move fast. Editing the contents of a DataGrid control is practically impossible without the mouse. However, we've seen many similar applications that allow users to enter orders/invoices by editing a grid control. To better understand the requirements of such an application, consider how cashiers at large department stores work: most of the time they scan barcodes. They don't touch their keyboards, except to print the receipt. When the scanner fails to read, the cashier enters the barcode manually. If the barcode can't be read at all, they can search by a product code that's printed on the label.

The Application's Interface

Short of building a custom control or buying a third-party control, your best bet for building an invoicing application is to base it on a ListView control. The ListView is a non-editable grid; the editing of the data should take place on a few controls outside the grid, and the ListView control should be used for displaying the invoice's rows. The form shown in Figure 18.3 is our idea of a functional invoicing interface.

[Team Fly](#)

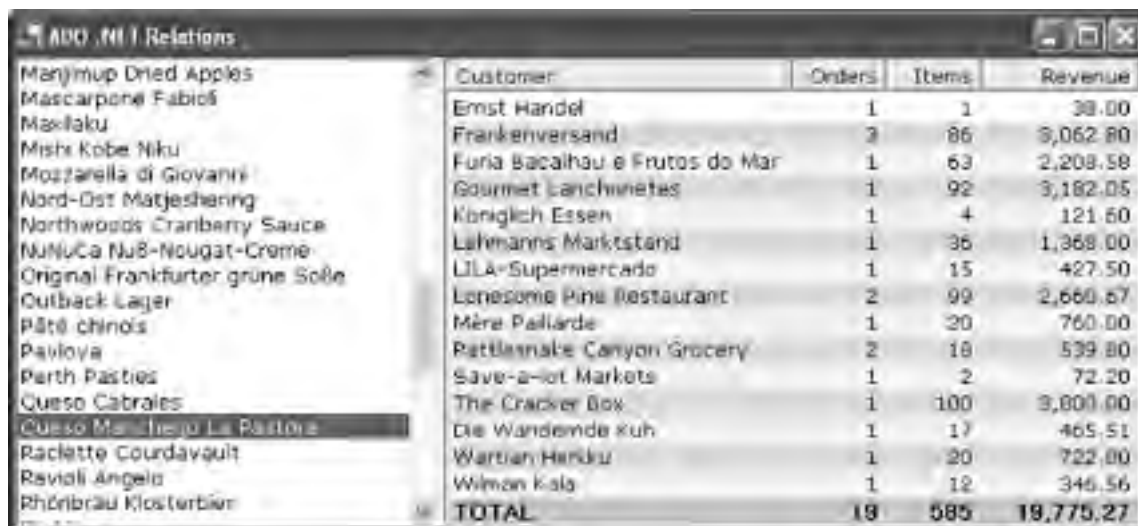
 Previous

Next 

The automated discount calculations impose another limitation on the design of the application's interface. The original NWOrders application allows you to switch tabs and select another customer even after adding detail lines to the order. We can't have this flexibility when the discount policy is based on the customer. To prevent users from selecting a new customer after having entered detail lines with discounts for another customer, we disable the Order Header tab. Another approach would be to allow users to select another customer and recalculate the discounts for the detail lines on the Order Detail tab.

The Relations Application

The Relations project demonstrates how to present related data on a Windows form. As you should guess, we're not going to use the DataGrid control, despite the fact that it's been designed to display related tables. The major disadvantage of the DataGrid as a data presentation tool is that it doesn't allow users to view the hierarchy of the data. The DataGrid control displays one level of data at any one time. Besides, users must select the relation they want to view on the control—certainly not the friendliest approach. We must give credit to the designers of the controls for the fact that the DataGrid can display any DataSet. A well-crafted application is very specific as to the data it handles and you can't expect a general tool to accommodate your needs as nicely as a custom solution. And this is what we'll do in this project: we'll write an interface that allows users to select a product from a list and see the customers who purchased the specific product, in how many of their orders it has appeared, and the total number of items of the same product each customer has ordered. The application's form is shown in Figure 18.6.



	Customer	Orders	Items	Revenue
Manjmur Dried Apples	Ernst Handel	1	1	39.00
Mascarpone Fabbio	Frankenversand	3	86	9,062.80
Maxilaku	Furia Bacalhau e Frutos do Mar	1	63	2,209.58
Mishi Kobe Niku	Gourmet Lanchonetes	1	92	3,182.05
Mozzarella di Giovanni	Königlich Essen	1	4	121.60
Nord-Ost Matjeshering	Lahmanns Marktstand	1	36	1,368.00
Northwoods Cranberry Sauce	LIJA-Supermercado	1	15	427.50
NuNuCa Nuß-Nougat-Creme	Lonesome Pine Restaurant	2	99	2,660.67
Original Frankfurter grüne Soße	Mère Pâliarde	1	20	760.00
Outback Lager	Rattlesnake Canyon Grocery	2	18	539.80
Pâté chinois	Save-a-lot Markets	1	2	72.20
Pavlova	The Cracker Box	1	100	3,800.00
Perth Pasties	Die Wandende Kuh	1	17	465.51
Queso Cabrales	Warrian Herku	1	20	722.00
Queso Manchego La Pastora	Wilman Kals	1	12	346.56
Raclette Courdavault				
Ravioli Angelo				
Rheinbrau Klosterbier				
	TOTAL	19	585	19,775.27

FIGURE 18.6 The Relations project displays sales data about each product.

The Application's Architecture

When the application's form is loaded, all the data are loaded into the ProductSales DataSet. In a real application you should provide an interface that enables users to limit the selection. For example, select orders placed in a time interval, the orders of customers from a specific country, and so on. The tables of the Northwind database are very small and we've chosen to download all their rows to the client.

[Team Fly](#)

 Previous

Next 

The listing is a bit lengthy because it calculates totals, formats the cells of the ListView control, and so on. We'll focus on the statements that navigate through the hierarchy of the DataSet's rows. When the user selects an item in the list, the following actions are performed from within the control's SelectedIndexChanged event handler:

1. The detail lines that refer to the selected product are copied from the Order Details DataTable into an array of DataRow objects with the DataTable's Select method:

```
DetailRows = _  
    ProductSales1.Order_Details.Select('ProductID = ' & productID)
```

The DetailRows array contains all the rows of the Order Details DataTable that refer to the product whose ID we passed to the Select method as argument.

2. The program creates a new DataTable, the OrdersTable, with the same structure as the original Orders DataTable of the DataSet. This DataTable will store all the rows of the Orders DataTable that correspond to the details selected in step 1. The following statements iterate through the rows of the DetailRows DataTable, retrieve the order to which the detail belongs, and add it to the OrdersTable DataTable. The GetParentRow method accepts as argument the name of the relation between the Order Details and Orders tables.

```
For Each DetailRow In DetailRows  
    OrderRow = DetailRow.GetParentRow("OrdersOrder_Details")  
    OrdersTable.Rows.Add(OrderRow.ItemArray)  
Next
```

3. The OrdersTable DataTable now contains the orders that include the product selected on the ListBox control. Another loop iterates through the rows of this DataTable and displays them on the ListView control. In addition, it keeps track of the number of orders placed by each customer and the total amount spent by each customer for the selected product.
4. To retrieve the customer name from each order, the program calls the FindByCustomerID method of the Customers DataTable, passing as argument the customer's ID with the following statements. The CustomerRow variable is of the ProductSales.CustomersRow type.

```
CustomerID = OrderRow.Item("CustomerID")  
CustomerRow = _  
    ProductSales1.Customers.FindByCustomerID(CustomerID)
```

The remaining statements populate the ListView control, calculate the totals, and perform other straightforward tasks.

The Relations project demonstrates an interesting alternative to the DataGrid control for building interfaces that display related data. It involves quite a bit of code, as opposed to the DataGrid, but you have absolute control over the appearance of the data and you can display all the levels of your data hierarchy.

The Relations1 Project

Figure 18.7 shows an application that maps more complicated relations on a ListView control. The Relations1 project maps the publishers of the Pubs database, along with their titles and each title's

[Team Fly](#)

 Previous

Next 

```
        LI = New ListViewItem()  
        LI.Text = ' ' "  
    Next  
Next  
End Sub
```

The outer loop adds a new item for each publisher in the publishers table. Then it retrieves all the books under the current publisher by calling the `GetChildRows` method of the `DataRow` object that represents the current publisher. The selected titles are stored in an array of typed `DataRow` objects.

The first nested loop iterates through the titles and retrieves each title's entries in the `titleauthor` `DataTable`. These entries, which are pairs of title/author IDs and correspond to the authors of the current title, are stored in the `TitleAuthorRows` array. The last nested loop goes through these rows and retrieves the authors of the current title by calling the `GetParentRows` method of the current `TitleAuthorRow` object.

The code also keeps track of the changes in the publisher name and title, so that it can add each publisher's first title next to the publisher name, but not repeat the same publisher name until it runs into a new publisher. The same is true for titles: titles with multiple authors appear only once, but each author is added to the `ListView` control as a separate item. You can open the `Relations1` project in the Visual Studio IDE and go through its code, which contains quite a few comments, which are not shown in Listing 18.23.

Summary

In this chapter we've shown a few practical data-driven applications. The `DataGrid` control is not the ultimate tool for displaying data—in fact it's not even the most appropriate control for building data-driven interfaces. Typical business applications use the same controls used to build any other type of interface.

Some of the more advanced and richer Windows controls, such as the `ListView` and `TreeView` controls are not even data-bound. However, it's fairly straightforward to populate them with data from a `DataSet`, as you have seen in the examples of this chapter. The `DataGrid` control is a very convenient tool for developers, because it allows you to quickly view the contents of a `DataSet`, but it's not the most suitable control for building intuitive interfaces for end users. The `ListView` control is ideal for displaying data. In Chapter 7 we developed a custom control that inherits from the `ListView` control and can also print its contents. You can combine this custom control with the techniques of this chapter to build even more functional interfaces.

You've also seen an example of a data-driven application that makes use of a middle tier component to simplify the deployment of the application. The middle tier component was implemented as part of the application, but we discussed in Chapter 14 how to deploy middle tier components as COM+ applications, web services or remotable components.

Chapter 19

Working with Regular Expressions

THIS CHAPTER DEALS WITH a classic and very popular topic in computer science, *regular expressions*. Regular expressions are supported in the .NET Framework by the `System.Text.RegularExpressions`, which we'll discuss here, along with a few practical examples. Regular expressions are strings that match patterns of text. They consist of characters and digits, some of which have special meaning. The asterisk, for example, means any number of characters and the period means any single character. The expression `"."` (without the quotes, of course) means any character, any number of times. Regular expressions are not the same as the wildcard characters you use to match file patterns. The expression `"*"` has a totally different meaning as a regular expression than it has as a file-matching specification. As a regular expression, it matches an entire line of text, or the entire text.

A regular expression allows you to search for general text patterns, instead of literals. The `IndexOf` method of the `String` class searches for a specific string in a longer one. The `IndexOf` method (the `InStr()` function works the same way), locates exact instances of the string you specify as argument. When you use regular expressions, you can specify a pattern such as all e-mail addresses or all dollar amounts in the text. If the text contains product codes that have a specific pattern, like `XX-NNN-X`, where `Xs` are uppercase letters and `Ns` are digits, you can locate all product codes in a single sweep through the text with the help of the appropriate regular expression.

Regular expressions are an extremely powerful tool in text processing. The `System.Text.RegularExpressions` class abstracts a very powerful engine for matching regular expressions against arbitrary text. All you have to do is specify the regular expression and the text to be searched and then call a method to retrieve the matches. The `Regex` class makes it very easy to locate any pattern in a text, as long as you can construct the appropriate regular expression. Writing the correct regular expression is not trivial, but this is something you get used to. Besides, a simple search on the Internet will return many regular expressions for common patterns such as e-mail addresses, IP addresses, phone numbers, and so on.

Writing Regular Expressions

Before examining the methods of the `RegularExpressions` class and how to program it, we'll review the process of building regular expressions. To experiment with regular expressions as you read through the material of this tutorial, use the `RegExEditor` application, which is described later in this chapter. This application, shown in Figure 19.1, is a simple text editor that supports the usual editing operations (copy/cut/paste); its Find command allows you to search with regular expressions as well as literals. The regular expression in the Search For box of the Find & Replace dialog box locates words that begin with "t," followed by three characters (any three characters) and ending with the character "e." The `\w` construct in the regular expression is a so-called metacharacter that matches a word character (everything except spaces and punctuation symbols). This metacharacter is followed by a count in curly brackets. The subexpression `\w{3}` stands for three consecutive word characters. The `\b` construct is another metacharacter that matches word boundaries. By placing the `\b` metacharacter at the beginning and the end of the regular expression, we specify that the matches should be complete words. If you're totally unfamiliar with regular expressions you may find them quite odd, but don't give up yet. We'll explore all metacharacters in the following sections, starting with the simpler ones and progressing to more complicated ones.

You can also use the `RegularExpressions` project to experiment with regular expressions. This project demonstrates a few of the more advanced topics, such as using groups in the regular expressions and performing replace operations using regular expressions. You can also use it with simple regular expressions such as the ones we'll build in the following sections. Enter the regular expression to match against the text in the upper `TextBox` control (Search Pattern box) and the text to be searched in the large `TextBox` control (Text box), then click the Find First Match button to locate the first match. After that, keep clicking the Find Next Match button to locate the next match in the text. Each time a match is found, the matched text will be highlighted on the control with the text. For the time being, you can ignore the Replace Pattern box and the Replace Matches button, as well as the Groups box at the bottom of the form. We'll look at the function of these two buttons later in the chapter, when we'll discuss replacement operations with regular expressions. The Groups box contains the matches of complicated regular expressions that contain groups, which we'll cover later. Simple regular expressions, like the ones we're going to discuss in the introductory sections, contain a single group and a single capture.

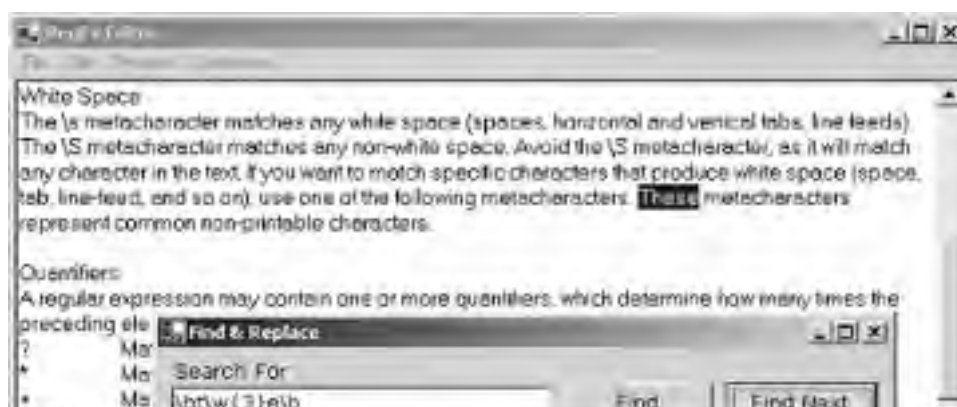




FIGURE 19.1 Matching regular expressions with the RegExEditor project

If the letters (or numbers) are consecutive, you can use the range operator and specify the first and last letter in the range. The following expression matches all uppercase characters:

```
[A-Z]
```

while the following matches all numeric digits:

```
[0-9]
```

The expression `[1357]` means any of the digits "1", "3" or "5" or "7."

ISBNs are made up of 9 numeric digits followed by a check digit, which can be either a numeric digit or the character X. The following expression locates ISBN values in the text (but it doesn't validate their check digit, of course):

```
[0-9]{9}[0-9X]
```

This pattern instructs the regular expression engine to locate 9 numeric digits followed by another numeric digit, or the character X. The expression `[0-9]{9}` means 9 digits. The next character can be either a digit or the letter X.

Notice the content of the second pair of square brackets: it matches a character in the range 0–9, or the character "X." Digits are so common that there's a special metacharacter for them. This is the `\d` metacharacter. The following regular expression will also locate ISBNs in a text:

```
\d{9}[\dX]
```

This expression will match all runs of 10 digits in the text, even if they're part of a very large, unformatted number. If the text contains the number 390102188541, then the previous regular expression will report the first 10 digits as a match. To avoid erroneous matches, we must also use the `\b` metacharacter, which specifies the beginning or the end of a word. Our final regular expression for locating ISBN values is:

```
\b\d{9}[\dX]\b
```

There's quite a bit about regular expressions and we'll return to the topic of building regular expressions shortly, but first we'll take a closer look at the `RegularExpressions` class. This class belongs to the `System.Text` namespace and it exposes all the functionality you'll need to use regular expressions in your .NET applications.

The RegularExpressions Class

Now that you have a general idea of what regular expressions are and how to locate general patterns of text, we can explore the basic functionality of the `RegularExpressions` class. To exploit the functionality of regular expressions in your code, you must import the `System.Text.RegularExpressions` class to your project:

```
Imports System.Text.RegularExpressions
```

and then create an object of the RegEx type:

```
Dim RX As Regex
```

[Team Fly](#)

 Previous

Next 

The Elements of a Regular Expression

In this section we'll go through the various metacharacters you can use in building regular expressions and look at numerous examples. This is a more formal treatment of regular expressions and, unlike in the introduction of the chapter, we've organized the metacharacters according to their function.

Characters and Metacharacters

Regular expressions are made up of regular characters (they match the same characters in the text), metacharacters, and special symbols. Metacharacters are regular characters prefixed by the slash character. The character "w" in a regular expression will match the same character in the text. If you prefix it with a slash, you turn it into a metacharacter: the `\w` metacharacter will match any word character. The "d" character will match the same character in the text, but the `\d` metacharacter will match a digit. The `\W` and `\D` metacharacters will match any non-word character and any non-digit character, respectively. Some symbols also have special meaning in a regular expression. The period matches a single character (any character, including the space) in the text and the square brackets are used to declare a range of characters. To match any of these symbols in the text, you must prefix them with the slash.

The simplest regular expression you can build is a regular string. The `Match` method will locate all the instances of the regular expression in the text, as if you were using the `InStr()` function, or the `IndexOf` method of the `String` class. If you use the string "Basic" as a regular expression, you will locate all instances of the word "Basic" in the text. By default, the search is case-sensitive. If you turn on the `IgnoreCase` option, you will also locate all instances of the words "BASIC," "basic," and so on.

To specify a more general pattern, you must include one or more metacharacters in the regular expression. One of the most common metacharacters of regular expressions is the period, which matches any character. The asterisk is another metacharacter that matches the preceding pattern any number of times. The expression `.*` will locate entire sentences in the text, because the period doesn't match the newline character.

If you want to treat any of the metacharacters in the regular expression as regular characters, you must "escape" them with a slash. To locate a period followed by an asterisk in the text, use the following regular expression:

```
\. \*
```

In the following sections you will find descriptions of all metacharacters used in building regular expressions, and examples to demonstrate their usage.

Single Character Metacharacters

A very common metacharacter in building regular expressions is the `\w` symbol, which means

a "word character." Use this metacharacter to specify a character in a word and exclude spaces and punctuation. The period metacharacter matches any character, including spaces and punctuation symbols. The `\w` metacharacter matches word characters only. The pattern

`...ce`

[Team Fly](#)

 Previous

Next 

LISTING 19.8: THE REPLACE ALL BUTTON'S CODE

```
Private Sub btnReplace_Click(ByVal sender As System.Object, _  
                             ByVal e As System.EventArgs) _  
    Handles btnReplace.Click  
    If chkRegex.Checked Then  
        Dim searchOptions As RegexOptions  
        searchOptions = RegexOptions.Multiline  
        If Not chkCase.Checked Then  
            searchOptions = searchOptions Or RegexOptions.IgnoreCase  
        End If  
        Regex = New System.Text.RegularExpressions.Regex( _  
            searchWord.Text, searchOptions)  
        Dim selStart As Integer = EditorForm.txtBox.SelectionStart  
        Dim replacementText As String  
        replacementText = Regex.Replace( _  
            EditorForm.txtBox.SelectedText, _  
            replaceWord.Text, _  
            System.Text.RegularExpressions._  
            RegexOptions.Multiline)  
        EditorForm.txtBox.SelectedText = replacementText  
        EditorForm.txtBox.Select(selStart, replacementText.Length)  
        EditorForm.txtBox.ScrollToCaret()  
    Else  
        If EditorForm.txtBox.SelectedText <> "" Then  
            EditorForm.txtBox.SelectedText = replaceWord.Text  
        End If  
    End If  
    btnFindNext.PerformClick()  
End Sub
```

You can experiment with the RegExEditor project, or even use it as a starting point for a highly specialized editor. Let's move on to some more advanced topics in regular expressions.

Advanced Topics in Regular Expressions

So far you've learned the basics of regular expressions. The metacharacters and symbols you've seen so far are adequate for many practical applications, but there are more topics to explore in regular expressions.

First, we'll examine the grouping of matches in a regular expression. A lengthy regular expression can be broken into simpler ones, which you can refer to later in the same regular expression. Being able to refer to previous matches allows you to perform very powerful searches (such as locating repeated words in a text).

In the replacement string we'll make use of the two grouped subexpressions in the regular expression. The following replacement pattern will place each key and each value in square brackets and each pair on a separate line:

```
[$1] : [$2]
```

You must also press the Enter key once at the end of the replacement string. If not, each key/value pair won't be printed on a separate line. You must also make sure that there's no newline character at the end of the search pattern. The text after replacing all instances of the search pattern with the replacement string is shown next. The text contains the same data as the original document, but it appears in a nicer format (see Figure 19.7):

```
[value1] : [34]
[value2] : [405]
[value3] : [4534]
[value4] : [45]
[value5] : [3334]
[value10] : [-4554]
[value11] : [3904]
[value12] : [456]
[value13] : [5564]
```

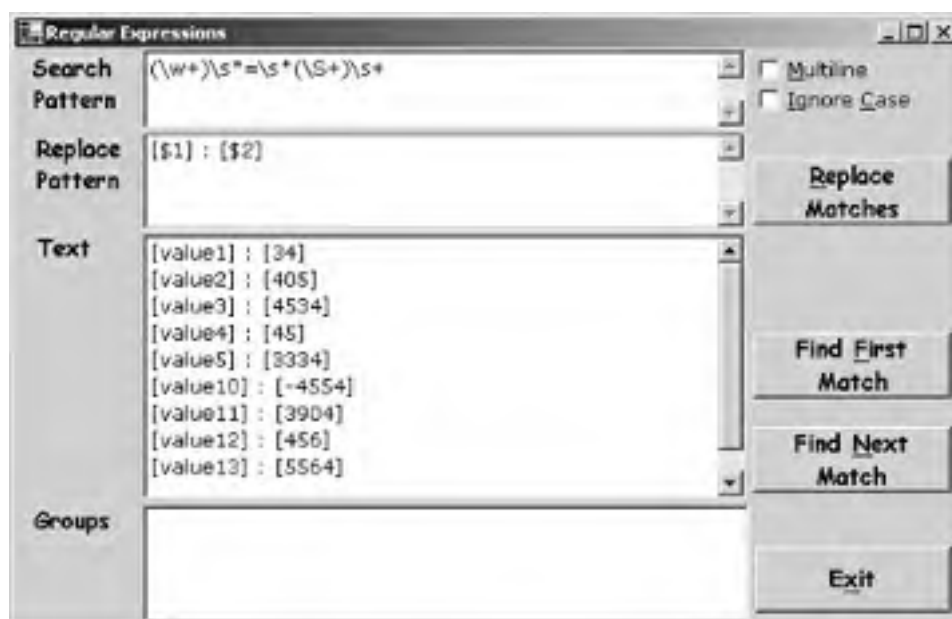


FIGURE 19.7 Cleaning a data file with a regular expression that captures all the instances of the same pattern in the text

The RegularExpressions Project

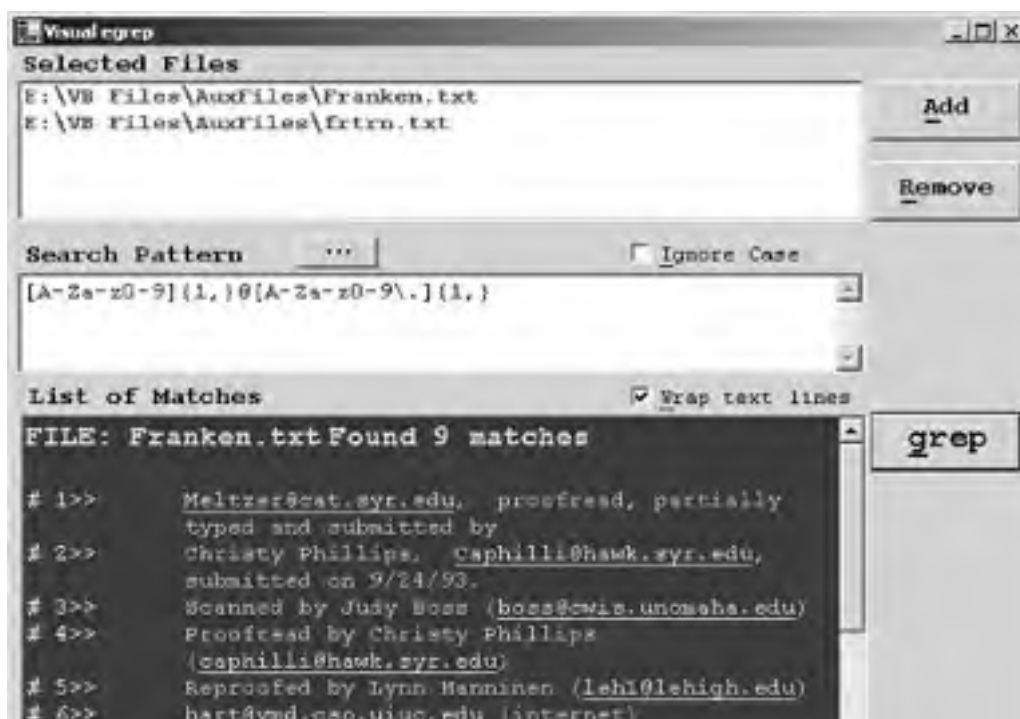
Now we can examine the code of the RegularExpressions project, which you can use to experiment with regular expressions, or process long text files using regular expressions. In the Search Pattern box you enter a regular expression that determines the matches you want to locate in the text, which is entered in the Text box (just copy the text you want to search and paste it in the Text box). Then you can click the Find First Match and Find Next Match buttons to locate the matches in the text.

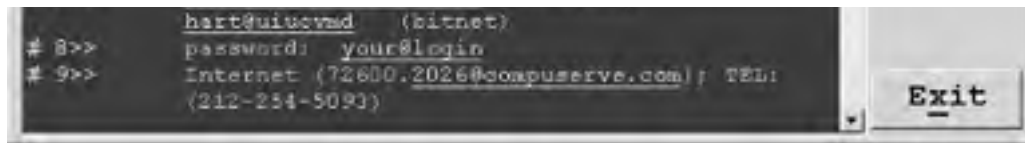
```
End Try
If txtReplace.Text.Trim <> "" Then
    txtText.Text = RX.Replace(txtText.Text, txtReplace.Text)
Else
    Dim MatchEval As New MatchEvaluator(AddressOf UCaseEvaluator)
    txtText.Text = _
        RX.Replace(txtText.Text, txtPattern.Text, MatchEval)
End If
End Sub
```

The Visual grep Project

One of the classic (and most popular) tools of the Unix operating system is the grep utility, which searches text files with regular expressions. Even though it's spelled in lowercase, it stands for General Regular Expression Parser. The application's interface is shown in Figure 19.8. I've actually tried to emulate the look of the old monitors by using shades of green on the visual interface of the application, but you may find them objectionable.

The Visual grep project is a visual adaptation of the grep utility. It allows you to select any number of text files on your drive and apply a regular expression against them. The names of the selected files appear in the ListBox control at the top of the form and you can add/remove files with the Add and Remove buttons. Once you've selected the files you want to process, you can type a regular expression, or select one of the predefined regular expressions by clicking the button with the ellipses. A dialog box, shown in Figure 19.9, will pop up; here you can select a regular expression by its description and click the OK button to paste the regular expression that corresponds to the selected description onto the appropriate box on the application's main form.





```
hart@uiucvmd (bitnet)
# 8>> password: your@login
# 9>> Internet: (72600.2026@soopuserve.com); TEL:
      (212-234-5093)
```

FIGURE 19.8 Using the Visual grep project to locate e-mail addresses with a regular expression

Summary

This concludes our overview of regular expressions. You have enough information about building regular expressions and using them to perform powerful searches in large text files, but there's more to regular expressions. The details can get hairy, but, fortunately, for most practical applications you won't need extreme knowledge of regular expressions. Regular expressions are an inherently difficult topic that is very popular among computer science majors and Unix programmers. However, they are very interesting and should be explored further. We should mention here that Perl (Practical Extraction and Report Language) is based on regular expressions and was designed around them. With Perl you can embed regular expressions in your code just like variables. The If statements of Perl are very compact, but really awkward to understand. If there's a write-only language, this should be Perl. Yet it's quite popular.

You should perform some searches on very large text files (using either of the applications discussed in this chapter) to get an idea of how efficient the RegularExpressions class's code is. Of course, regular expressions aren't the bread and butter of a typical developer, but some tasks can be simplified enormously with the functionality of the RegularExpressions class. You will find this class especially useful if you're interested in language statistics (distribution of word count versus their length, words that contain specific letter combinations, and so on). Project Gutenberg at www.gutenberg.net is a great resource for text files representative of the English language: It provides thousands of free electronic versions of classic literature.

Chapter 20

Advanced Graphics

ONE OF THE MOST interesting aspects of a programming language is graphics. The graphics engine of .NET is the Graphics Device Interface (GDI+). GDI+ is part of the Windows XP operating system that provides support for two-dimensional vector graphics, imaging, and typography. GDI+ is the successor to the Windows Graphics Device Interface (GDI), but it's more than an improved version of GDI; it's a new optimized graphics engine with many new features. We looked at GDI+ in Chapter 7, where we discussed the new printing techniques of .NET. In Chapter 7 we focused on a few methods we use to generate business graphics: how to print text, how to create reports with tabular data, how to draw lines and frames. In this chapter, we'll explore methods for manipulating individual pixels on a bitmap.

To demonstrate the advanced graphics methods of this chapter, we'll build two very different applications, one for plotting functions and one for generating fractals. The first application is a custom control that will plot any user-supplied two-dimensional function. In the process you'll also learn how to evaluate math expressions at runtime. The PlotControl custom control, which you'll build in the following section, allows you to set the properties of the plot (the range of values over which the function will be plotted, the axis titles, the style and color of the plot, and more) and you can incorporate it into your applications to give them plotting capabilities.

PERSISTENT DRAWING

According to the documentation, you should insert your graphics statements into the OnPaint event, which is fired every time a form (or control) must be redrawn by Windows. In effect, this technique redraws your graphics elements every time a segment of the window is uncovered, or when the window is resized. The graphics you generate from within the OnPaint event are not persistent: they're redrawn on the form or control as needed.

A complicated drawing, such as the drawings we'll develop in this chapter, may involve a large number of calculations. If you create your graphics from within the OnPaint event handler, the calculations will be repeated every time the form is refreshed. As a result, the refresh operation won't be instant. To avoid this unnecessary delay, you can create persistent graphics by drawing on a bitmap object and then displaying this bitmap. The bitmap need not be refreshed and the form that contains it is redrawn instantly. In the examples of this chapter we'll create persistent graphics by drawing on a bitmap, which is the background image of a Form, or PictureBox control.

The second application is a fractal generator. Fractals are a special type of function plotting; instead of a simple curve, fractals fill the space with intricate patterns of startling beauty. The fractal generator is not a practical application in a strict sense, but it's a fun application you can use to experiment with fractals.

The PlotControl

Our first sample application is a custom control for plotting two-dimensional functions, as shown in Figure 20.1. Figure 20.1 shows a test form that uses the PlotControl. The control, which takes up the upper part of the form, displays the plot of the two functions specified in the TextBox controls near the bottom of the form. To create a plot, you set the control's properties and then call the Plot method.

Create a new solution in Visual Studio and add two projects to it: a Windows Control Library (the PlotControl project) and a Windows project (the TestProject). The Windows project is the test project for the control. The PlotControl is a compound custom control that contains a PictureBox control, where the function plots and the grid are drawn. The titles and the axis numbers are printed on the control itself. The PictureBox control is anchored at all four sides and the bands around it, where the elements of the plot are drawn, have fixed sizes. In our code we set up two Graphics objects for drawing. The `G` Graphics object represents the surface of the control. This is where we'll draw the axis titles and numbers, and the `G` object is created with the following statements:

```
Dim bmp As Bitmap
bmp = New Bitmap(Me.Width, Me.Height)
Me.BackgroundImage = bmp
Dim G As Graphics
G = Graphics.FromImage(bmp)
```

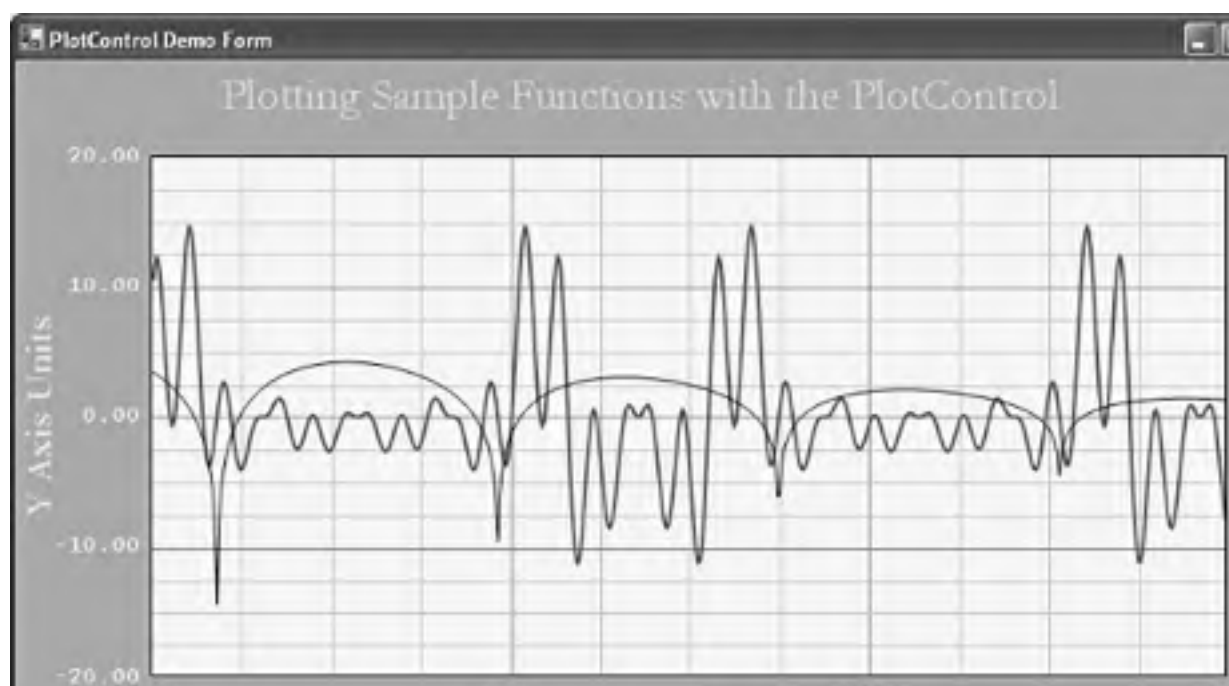




FIGURE 20.1 Using the PlotControl in a test application

DEALING WITH SINGULARITIES

One issue that deserves attention is the handling of singularities. Singularities are points at which a function can't be calculated. Consider the function $\text{Cos}(X) / X$. This function can be plotted in any range that doesn't contain the point 0. If you attempt to calculate the function at $X = 0$, the result is an undefined number (NaN). If you run into a singularity, you can either ignore it and continue, or abort the process and inform the user that the function can't be plotted. In our code, we abort the process.

However, it's possible to skip the singularity, even though it's included in the range of X values. If the size of the PictureBox control is 200 pixels and you're plotting a function in the range from -2 to 2 , you'll be calculating the function at increments of $4 / 200$, which is 0.02 . The points at which the function is calculated are $-2, -1.98, -1.96$, and so on up to $-0.02, 0, 0.02$. When you attempt to calculate the function at point 0, you'll run into a singularity. If the PictureBox control's width were 285 pixels, however, the step along the X axis would be $4 / 285$, or 0.014035087 . The function will be evaluated at the point -0.00701754 and then at 0.00701754 . The function can be evaluated at both points and the singularity has been skipped with no special effort on our part. Usually, it's the responsibility of the user to avoid singularities in the range of X values and specify a meaningful X range for the plot.

In the second half of this chapter we're going to look at fractals. Fractals are special plots that aren't plotted with curves; instead, they fill the space with intricate patterns.

A Fractal Generator

People who start playing around with fractals sometimes get hooked. Fractals are like alien worlds—obviously different from things we see in nature, yet also somehow familiar. You can zoom into a fractal endlessly, producing fascinating variations of color, texture, and shape. And what you see as you take this tour somehow looks not only natural, like a cabbage or a tree, but also mysterious enough to earn fractals their reputation as the most complex objects in all math.

In this section we're going to demonstrate how you can generate fractals in VB. Beyond that, we'll also attempt to explain to nonmathematicians the strange numbers and odd dimensions that produce fractals. Mathematicians like fractals because they produce images of often startling beauty. Most mathematical formulae, when plotted, result in wave-like lines, arcs, and other visually simple—really rather boring—geometric designs. Fractals, by contrast, yield extremely complex, lacy, colorful patterns that hover just beyond symmetry. You never really see the same thing twice, though at first you might think so. Fractals often imitate the patterns found in nature—those produced, say, when a coastline erodes, or when an octopus grows a tentacle.

What Is a Fractal?

One way of describing a fractal is *the adventures of a small number on the complex plane*. A

fractal is a peculiar and very dense "graph" generated by a mathematical process. Although the resulting images are literally infinitely complex, the underlying algorithms are short and rather simple. When you want to see relationships between numbers—to see mathematical expressions—you can put them into a kind of grid called a plot. The coordinates of this space are arbitrary—that is, you can set up the marks to be large enough to embrace whatever expression you are trying to make visual. You saw how to scale the plot of an arbitrary function to fill a given area in the example of the first part of the chapter.

experiment with the sample applications and discover other values of the Cx and Cy parameters that yield rich, colorful Julia shapes). Note, however, that most numbers you enter randomly will produce uninteresting fractals, and that your numbers must be between -2 and +2. Also remember that often the most elegant pictures result from zooming into the initial Julia fractal six or eight times.

- | | | |
|-----|--------------|-------------|
| 1. | Cx=-0.754 | Cy=0.049 |
| 2. | Cx=-0.744 | Cy=0.097 |
| 3. | Cx=-0.736 | Cy=0.097 |
| 4. | Cx=-0.756 | Cy=0.097 |
| 5. | Cx=-0.743 | Cy=0.097 |
| 6. | Cx=-0.766227 | Cy=0.096990 |
| 7. | Cx=-0.9 | Cy=0.12 |
| 8. | Cx=-0.745429 | Cy=0.113008 |
| 9. | Cx=-1.0300 | Cy=-0.9200 |
| 10. | Cx=0.320 | Cy=0.043 |
| 11. | Cx=0.3080 | Cy=0.46 |
| 12. | Cx=-1.330 | Cy=0.043 |
| 13. | Cx=-0.16 | Cy=1.32 |
| 14. | Cx=-1.8 | Cy=-1.67 |

Complex Number Operations

Most readers are not likely interested in the details about addition and multiplication of complex numbers. So, we left this discussion for the end of the chapter. For the intrepid, here are the three basic complex number operations: addition, subtraction, and multiplication. Complex numbers are actually pairs of numbers (the real and imaginary parts) that are handled separately. The sum of two complex numbers is another complex number, whose real number is the sum of the real parts and whose imaginary part is the sum of the imaginary parts of the operands.

Adding and Subtracting Complex Numbers Here are the formulae for adding and subtracting complex numbers:

$$(a + ib) + (c + id) = (a + c) + i(b + d)$$
$$(a + ib) - (c + id) = (a - c) + i(b - d)$$

or

$$(a, b) + (c, d) = (a + c, b + d)$$
$$(a, b) - (c, d) = (a - c, b - d)$$

And here are some examples of addition and subtraction of complex numbers:

$$(3 + i7) + (-2 + i2) = (1 + i9) \text{ also: } (3, 7) + (-2, 2) = (1, 9)$$
$$(3 + i7) - (-2 + i2) = (5 + i5) \text{ also: } (3, 7) - (-2, 2) = (5, 5)$$

Multiplying Complex Numbers To multiply two complex numbers, we form all four products:

$$(a + ib) * (c + id) = a*c + ib*c + ia*d + ib*id$$

Summary

In this chapter we discussed some advanced graphics topics by means of two demonstration applications: a practical application for plotting 2-dimensional functions and a "fun" application that generates fractals. While building the plotting application you learned how to build GraphicsPath objects and how to apply transformations to graphics elements before rendering them in the drawing surface. You also learned how to calculate arbitrary math expressions at runtime with the MSScript ActiveX control.

The second sample application of this chapter was a simple fractal generator that produces startling fractal images. This application generates the fractals by painting one pixel at a time. The calculation of each pixel's color involves some math, which isn't beyond the grasp of the average developer. You can enhance the fractal generator in many ways and the most challenging aspect of the application is the design of a palette for coloring the fractals.

Chapter 21

Designing the User Interface

IF YOU'RE LIKE MANY programmers, you ask your spouse or a friend if this tie goes with that suit. You leave it up to someone else to select colors, patterns, and designs. In other words, you don't have much experience with visual design.

Fear not. In this chapter you'll find some guidelines for good Windows design. Talented designers at various software companies have spent lots of time developing these concepts in the past decade. You can see the results by comparing the uninviting flat gray appearance of early Windows applications with the sleek, sculpted, dimensional look that has become the standard in recent versions of Windows, especially XP.

Many studies have demonstrated that how a program looks influences how it's used, and also how it's rated. The most obvious example is grouping a set of related radio buttons into a single GroupBox. This cues the user that these buttons are mutually exclusive choices, making the user's life easier. There are many other examples. User-interface design is a surprisingly highly developed set of techniques and suggestions, some rather subtle. Surprising, given that it's only a decade old (nobody counts pre-Windows DOS UI theory as significant anymore—for the same reason that automakers no longer take into account the features of the horse-drawn carriage).

Making Applications Look Reliable

We'll explore a variety of techniques you can use to make your VB programs look better. At first we'll work with tools that VB provides—the BackgroundImage property of forms and of a few controls. Then we'll go beyond what VB provides, demonstrating how to do pretty much anything visual that you want to do. You've seen commercial software with slick lighting effects, fade transitions, embossed and shadowed text, sliding panels, opacity, and all the rest. In this chapter you'll see how to accomplish those tricks and other effects. But before going further, we've got to finish dealing with the fundamental question some of you are doubtless asking: Why bother?

A pretty form is more than merely a desirable luxury. If your work looks coordinated, polished, and professional on its surface, people will think it is equally solid on the inside. They will trust it more. Study after study has demonstrated that handsome men or beautiful women are far more likely to be believed than plain people. That's why grifters, lounge lizards, and con artists of all stripes are usually physically attractive. It's also why so many companies pay huge sums to improve their logo, and millions to get movie stars to recommend their products.

USING FOCUS GROUPS TO PRODUCE VISUAL GUIDELINES

Microsoft and others have conducted many focus groups, testing tens of thousands of people, and have found that certain design elements result in the most efficient form organization and most visually appealing "looks." One finding that might surprise you: Choose icons with relatively subtle coloring when designing your application. Too many bright tiny icons are annoying and clutter the screen.

If you've never studied, or even thought about, pictorial design, here's your chance. VB, and Windows, offer rich graphics possibilities. Computing is becoming increasingly, even relentlessly, visual. (This trend will not stop until computer programs are photo-realistic, until a telephone icon looks like a 3D hologram of a telephone, and until multimedia is so common that the distinction between computers and television disappears. So prepare yourself. Programming now requires that at least *someone* involved has a visual sense.)

Design is not just a matter of making things look better—it's also a matter of user-comfort, efficiency, and, ultimately, a quality that distinguishes professional from amateur programming. How things look and feel is a big part of how easily they are used. Ergonomics matters. And ergonomics is, in part, visual.

Visual design and decoration have not traditionally been part of a programmer's job description. But computing is increasingly graphical, and will never revert to the text-based interface typified by the beloved but infamous black DOS screen with its white words. The computer console is as dead as the floor-standing radio.

These days you must communicate with the user via graphics—even sometimes via video—as well as with text. Fortunately there are guidelines and conventions you can learn. Explaining these conventions, and providing hints about design, is the purpose of this chapter.

In the best-designed applications, some visual conventions include placement of the Close or Exit buttons in the lower right, a gray (or at least not white) background, related controls grouped into zones separated by frames, and so on. If one of your Visual Basic forms has its BackColor set to white, is unzoned, and locates the Exit Button on the left, your application will slow users down. It will confuse them because it's both homely and, in a bad sense, unique. Users are simply not familiar with odd design elements. There are conventions to Windows (form) design. Users might not know why, but they will be uncomfortable using your program.

Windows Conventions

There are several graphics conventions to which virtually all Windows programs now submit. Your programs should too. The most important of those are explained in the following sections.

The Metallic Look

First, many Windows programs still aspire to look "metallic," though various "themes" are less severe and involve earth tones and other color schemes. You can achieve the metallic look by building highlights and shadows into controls, such as buttons, and by using a metallic gradient for backgrounds to your forms and other elements. (For an explanation demonstrating how to create gradients, see the section titled "Metallic Shading" later in this chapter.)

Metallic Shading

One of the best ways to avoid dull-looking forms is to use gradient metallic shading. It's subtle and conservative enough for any business application, yet considerably more attractive than plain gray.

You can create gradients with Adobe's Photoshop, Corel's Picture Publisher, or most any photo-retouching program. Here's how to do it.

The best metallic gradient is a gradual shift between two shades: white and the typical Windows gray (the light gray often used as shading on windows and controls).

You can use the code in Listing 21.4 to add the top-left to lower-right metallic gradient to a form, as shown in Figure 21.10.

LISTING 21.4: CREATING METALLIC CRADIENTS

```
Imports System.Drawing
Imports System.Drawing.Drawing2D

Private Sub Form1_Paint(ByVal sender As Object, _
    ByVal e As System.Windows.Forms.PaintEventArgs) Handles MyBase.Paint

    Dim g As Graphics = Me.CreateGraphics

    'coordinates for both the gradient and the fillRectangle routi
    Dim x As Integer = Me.Width
    Dim y As Integer = Me.Height

    Dim lgBrush As New LinearGradientBrush( _
        New Point(0, 0), New Point(x, y), _
        Color.White, Color.FromArgb(190, 190, 190))

    'linGrBrush.GammaCorrection = True 'smooth the transition
    g.FillRectangle(lgBrush, 0, 0, x, y)

End Sub
```

To draw gradients on buttons and such, you either can create a gradient the button's size in a graphics program (including caption), then import the graphic into the button's Image property, or you can use this code:

```
Dim g As Graphics = Button1.CreateGraphics

    Dim x As Integer = Button1.Width - 3
    Dim y As Integer = Button1.Height - 3
```

```
Dim lgBrush As New LinearGradientBrush( _  
    New Point(1, 1), New Point(x, y), _  
    Color.White, Color.FromArgb(190, 190, 190))  
  
g.FillRectangle(lgBrush, 0, 0, x, y)
```

Notice the adjustments, in boldface (`-3` and so on), that permit the button's frame to show. You want to put the gradient only on the button surface, not cover up the shading around its edges.

***TIP** You may want to add additional special effects to your forms, such as custom buttons (perhaps round), lights that dim and fade, drop shadowing, neon, 3D, animation, sculpted labels, and other effects. Use graphics programs to create these effects, then import them into the form's and controls' `BackgroundImage` or `Image` properties. Use a `Timer`, for example, to occasionally display a reflection like those in Figure 21.10, then hide it by setting the `PictureBox` containing it to `Visible = False`.*



FIGURE 21.10 Using a variety of different gradient effects add solidity and sophistication to your applications. Notice that the background on this form is, itself, a gradient.

Sliding and Fading Transitions

Applications with multiple forms benefit from animated transitions. Just as it can be annoying if a movie jumps abruptly from one scene to another—simply slapping a new scene on top of the current one—so too will your projects look more professional if there's a visual transition between your forms.

When a user clicks a button to bring up Form2, you can slide it over from the side, or down from the top, like a garage door. Or you can use the classic fade: the new form gradually appears as if from out of the mist, replacing the current form.

In the past, computer video was too slow for fades (though it could cope with slides). Also, there was no built-in facility for adjusting opacity as there is now, the form's Opacity property (alas, there's no such property yet for individual controls—they all fade along with their parent form).

```
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    Timer1.Enabled = True
End Sub

Private Sub Timer1_Tick(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Timer1.Tick

    Panel1.Left += 2

    'stop when it hits the correct position
    If Panel1.Left >= X Then Timer1.Enabled = False

End Sub
```

When you run this, you see the panel slide in and take its correct position. Adjust the step size (+2) to fiddle with the granularity; it's unlikely that you can lower the timer's Interval enough to speed things up as much as you'd like to, so you have to accept some granularity. However, a step of 4 isn't noticeable, and 12 is probably too fast for you.

Try dropping panels or other controls down from the top; slipping them in, then back out when the user is finished with them; sliding more than one control at a time; and so on.

Summary

This chapter covers one of the most overlooked aspects of program design—the *design* itself, properly so called; the actual *look* of the finished application. Among the issues considered in this chapter are metallic surfaces, fonts, layering, light sources, depth, framing, shading, gradients, and transitions.

If you think the design job is best left to the art department, either you work for a very large company (with enough cash to hire application-design specialists with Windows ergonomics experience) or you're trying to avoid what is partly a programmer's responsibility. *You*, the programmer, are often the best person to advise which controls should be grouped together, and how the various windows in your application interact (and therefore what kinds of transitions should link them), and other graphics issues.

At the very least, the programmer or programming team should participate in meetings with the art department—if there is one—to ensure that purely visual considerations aren't overriding logical groupings or Windows conventions. And if there is no art department, that's all the more reason for programmers to refer to this chapter's suggestions before considering an application finished. Oh, and one more thing: Take a good look at Word or the Visual Studio IDE and make sure that your Cancel, OK, and other buttons are in the same location as theirs on your forms; make sure that your toolbars look like theirs; ensure that your menus are located where theirs are, and so on. Microsoft has spent loads of money testing their designs, so if you follow their conventions you won't go far wrong.

Chapter 22

Using the .NET Compact Framework and Its Emerging Technologies

PEOPLE WANT TO REMAIN connected to the Internet—and to their computer applications and files—no matter where they are. The current buzzword is *mobile* computing. This phrase has several meanings, but the one we're focusing on in this chapter is: How to squeeze programming and I/O into the highly restrictive platform of small, portable devices like PDAs and cell phones.

This chapter offers you an overview of the technologies and tools available to the VB.NET programmer who wants to extend their programming skills into the mobile arena. We'll explore the ways that .NET addresses the needs of users and programmers within the limitations imposed by mobile devices: as you'll see, this is truly computing lite.

Writing programs that will work on a cell phone is a bit like traveling back in time about twenty-five years to the day when personal computing was just getting started. Processor speeds were much slower than today's desktop machines enjoy. You had very little memory to work in, so you had to be careful and conserve this precious resource by avoiding such memory-hungry luxuries as graphics. And there was another reason to avoid graphics: You had to deal with I/O constraints such as low-resolution, black and white, text-based screens, and little, if any, mouse capability. Although most early computers were more restrictive (32K RAM was considered a lot of memory), today's mobile devices are nonetheless significantly less powerful than today's personal computers. The mobile platform demands a different kind of communication with the user.

The .NET Framework is designed for desktop computing, and to run on servers managing Internet sites. The .NET Framework runtime is large; it simply cannot fit within the memory available to mobile devices. The solution: a new framework, the .NET Compact Framework, a condensed, stripped-down version.

What's Eliminated?

To get a sense of the limitations you must work within when programming under the Compact Framework (CF), consider the Button control. If you look in Help under ButtonBase members, you see the properties and methods available to the base class from which various button-like controls inherit (RadioButton, CheckBox, Button). In the list of members, you see 68 total properties, but only 25 of them are described as "Supported by the .NET Compact Framework."

You find that properties such as AllowDrop are unavailable (dragging and dropping cause difficulties, serious bandwidth problems, if the mobile client expects to see an animated illustration as they drag). Also unavailable are TabIndex, Image, Dock, FlatStyle, and others. Here's a list of the properties supported for Button controls running on the CF:

BackColor	BindingContext	Bottom
Bounds	Capture	ClientRectangle
ClientSize	ContextMenu	Controls
DataBindings	Enabled	Focused
Font	ForeColor	Height
Left	Location	Parent
Right	Size	Text
Top	TopLevelControl	Visible
Width		

Output Lite

Writing for mobile devices presents several problems to a programmer. For one thing, PDAs and phones vary widely in their screen resolutions, color capabilities, input technology, and so on. Some mobile device programming platforms have "solved" this incompatibility issue by stripping down the features to the lowest common denominator. But clearly someone owning a nice color PDA doesn't want their applications in black and white.

Microsoft addresses this issue by promising to provide class libraries for the .NET Compact Framework that will target the capabilities of types of devices as well as individual models. There's only so much you can do with this approach, however. For example, when you're creating a word processor application for a color-capable screen, you make design decisions differently than you would for a black and white screen. You employ different visual cues than you do with the more restrictive environment of a black and white screen.

Solving the Connectivity Problem

With mobile computing there's another major issue a programmer faces: where should the actual programming execution take place? In the mobile device, in the server communicating with it, or shared between the two locations? And, if shared, should the client mobile device remain in continuous contact with the server during application execution? This is one aspect of what's called *the connectivity problem*.

The connectivity problem was solved by ASP.NET and ADO.NET—information to be detached from databases, for instance, and sent as an HTML package to users' browsers. This way, a continuous

[Team Fly](#)

 Previous

Next 

connection between client browser and server (or database) was not necessary. This decoupling of the remote user from the server is also a feature that distinguishes the CF from other mobile-computing platform initiatives, such as WAP (Wireless Application Protocol, a UNIX-derived standard for Internet communications and telephony on pagers, PDAs, two-way radios, cell phones, and other wireless devices). With the CF, you, the programmer, can balance the advantages, and the mix, of server-side versus client-side code execution. Also, one doesn't foresee the CF attempting to service pagers or two-way radios. Their I/O limitations are just too severe.

Security is also superior on the CF because it's managed code, and it has access to the various Code Access Security strategies described in Chapter 5.

When you program for the CF you benefit from many of the familiar and useful tools in the justly praised .NET IDE. You can also write your code in VB.NET or C# (the two languages currently supported in the CF). On the minus side, because the CF is a subset of the full .NET Framework, many tools and many language features are available to you, but some are not. If you've ever programmed with VBScript or other script versions of languages, you understand that limitations are built in.

The single most significant advantage of using the .NET CF, however, is that you can leverage your knowledge of how to write VB.NET programs, and how to use the .NET IDE tools, when programming for mobile devices. Most of the familiar features and techniques—ADO.NET, ASP.NET, security classes, and so on—are right there at your disposal, ready to be applied to the new and sometimes challenging task of creating mobile applications.

Using the Simulator

Within VB.NET 2003 is a facility for designing and testing CF applications. It roughly simulates the user experience of running your application, similar to the way that ASP.NET applications can be previewed in the Internet Explorer browser during development by pressing F5 to execute them. This simulation is a way for you to test the functionality of your application, but it cannot replicate how your application will actually look on mobile devices. In fact, it will look different on different devices. In any case, you can get the I/O sketched in, and the logic working and tested. Then you can switch to emulation, or actual testing on target devices for any final tweaking.

Try it out. Choose File ➤ New ➤ Project then double-click the ASP.NET Mobile Web Application icon in the New Project dialog box, as shown in Figure 22.1:

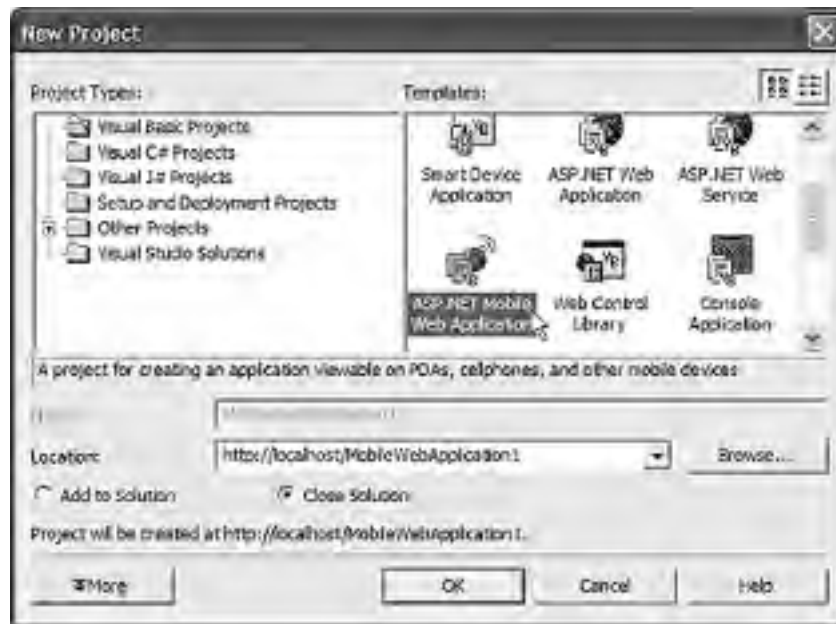


FIGURE 22.1 This icon is your gateway to building and modeling a PDA or cell phone application.

Use the Toolbox to add a second form to your page. Note that forms are *controls on the toolbox* in CF applications. The traditional ASP.NET form (which represents a browser window), and the traditional Windows form (which represents a window) are not used here. The form in CF is merely a unit of organization, a way of grouping controls, a container for code. Forms reside within a Page object.

Add a Link control to Form1 and change its NavigateURL property to #Form2. You'll find this option and any other application targets listed in the dropdown list in the Properties window.

Now put a Label control on Form2 and change its Text property to "You've arrived!" Press F5 and click the Link in Form1. You should see the *You've arrived!* message appear, demonstrating that you indeed navigated to Form2.

CF Forms offer several novel properties you should know about. There's a Method property; isn't it wonderful that a *property* is named *method*? Programmers have been driven barking mad by less. Anyway, the "Method property" describes the HTTP request—either Get or, the default, Post. The Action method can be an absolute or relative URL to which the form must submit a Get or Post, but it defaults to an empty string causing it to post back to the URL from which the form itself came. The PageCount tells you how many pages the form has when paginated, and the CurrentPage gives you the index of the current page. The PageStyle property includes a set of other properties such as color, text font, alignment, and so on, which you can use to customize the defaults used for pagination (the styles used for NextPage Text properties and such).

A Form's Deactivate event triggers when a new form becomes active via programming, or when the user navigates to a different form via a link.

More New Features

Now look at the Toolbox. You'll see some interesting controls available in no other Toolbox but the CF's: Form, PhoneCall, List, SelectionList, Object List, DeviceSpecific, and StyleSheet.

The new TextView control offers automatic pagination (within the text it displays). However the Form's Paginate property must be set to True for this to work.

The Image control has a tricky job because of the wide range of graphics capabilities of the various mobile devices. If you want to ensure the best experience for your users, you should choose images conservatively (make them rather simple and recognizable). Remember you're likely up against screens as small as 94×72 pixels in cell phones. Also you must tailor your image formats specifically to the various devices you're targeting using the device filters. Some devices will want .jpg and others some different graphics file format. .JPG is requested by Pocket Internet Explorer, for example, but WAP devices want a format you've perhaps never heard of called .wbmp. WAP also supports a .png format (Portable Network Graphics). .GIF is also popular. You just have to use a format translator (found in many graphics

programs) to create the various files in the various formats. PaintShop Pro, for example, supports .png, but I couldn't find any support for .wbmp.

Just as e-mail and instant messaging users resort to emoticons and other crude symbols instead of graphics, mobile devices often include a set of symbols, clipart, icons, cartoons, or glyphs representing common graphics ideas such as lightning storms. You can employ these to liven up your output. Resources are available online and you can locate them by searching for the device you're targeting. You use the ImageURL property and specify `symbol:nnnnn` with `nnnnn` being a code or code word, such as `symbol:cloudy`.

[Team Fly](#)

 Previous

Next 



FIGURE 22.5 Bind mobile List controls to arrays or other data sources.

Mobile Security

As you can imagine, sending information over the air is even less secure than sending it over a LAN, cable, or phone line. When you're exchanging messages wirelessly using cell phones or PDAs, you might as well consider yourself a radio station.

You can, of course, encrypt sensitive data, and you should. As described in Chapter 6, .NET includes several powerful encryption routines that can easily be added to your programs to ensure your privacy. As for authenticating callers, you'll find the following authentication section in the `Web.config` file:

```
<!-- AUTHENTICATION
    This section sets the authentication policies of the application.
    Possible modes are 'Windows','Forms','Passport' and 'None'
    "None" No authentication is performed.
    "Windows" IIS performs authentication (Basic, Digest, or Integrated
    Windows) according to its settings for the application. Anonymous access must b
    disabled in IIS.
    "Forms" You provide a custom form (Web page) for users to enter their
    credentials, and then you authenticate them in your application. A user credent
    token is stored in a cookie.
    "Passport" Authentication is performed via a centralized authenticati
    service provided by Microsoft that offers a single logon and core profile servi
    for member sites.
-->
```

ASP.NET relies on cookies when performing form-based authentication, so you should probably avoid this approach (so many mobile devices don't support cookies). For details about the features of Windows-based authentication, see Chapter 5.

You can specify user names, roles (such as administrator), and other modes of access such as passport. IIS stands guard in front of the localhost (Web simulator) or the Web itself in a deployed mobile application. You can use IIS's Internet Services Manager in Control Panel's Administrative Tools folder to modify security policies rules.

[Team Ely](#)

 Previous

Next 

Debugging via Tracing

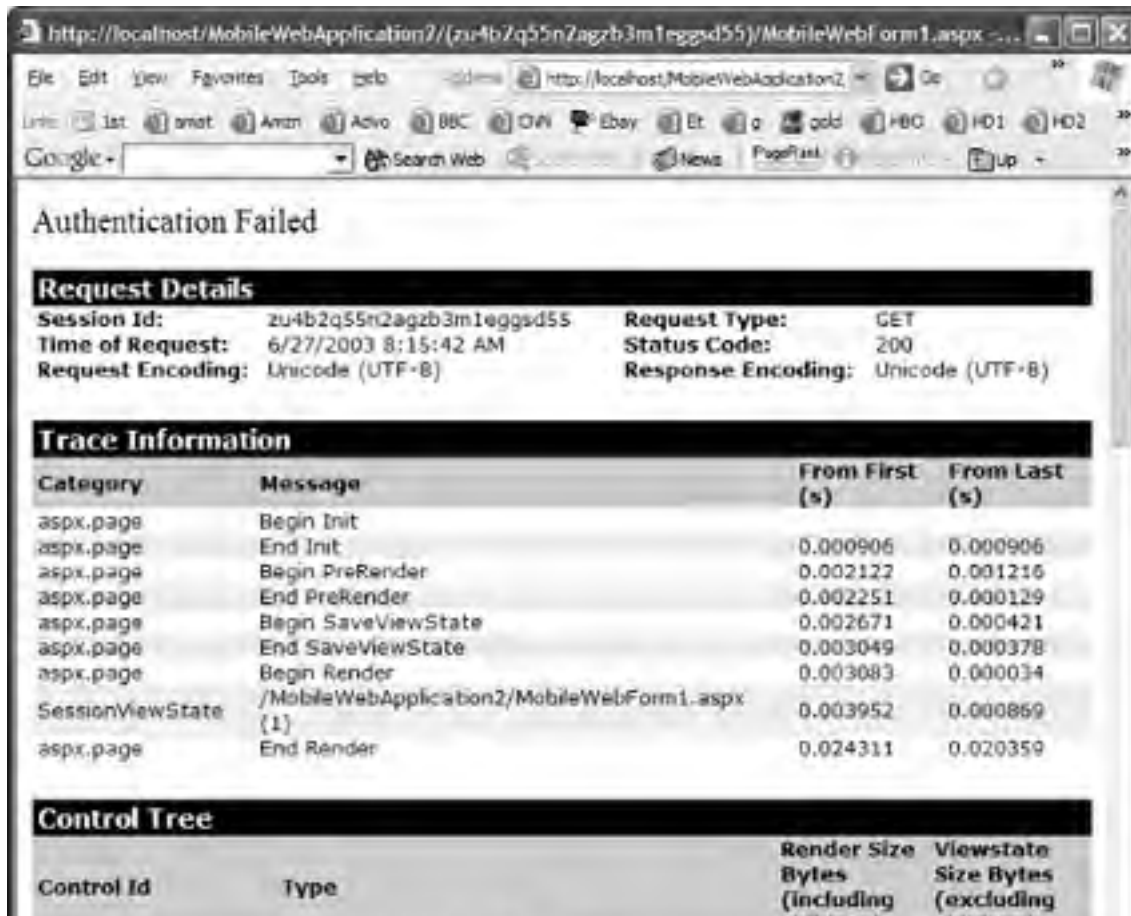
Debugging a mobile application is similar to the techniques you can use in traditional Windows applications, and is particularly like debugging ASP.NET applications. You'll be able to resort to many of your usual practices and strategies, and many of the usual VB.NET debugging tools. See Chapter 9 for an in-depth discussion of various approaches to debugging.

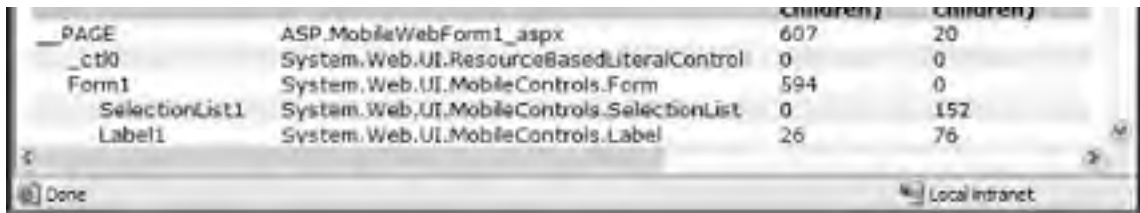
However, mobile and ASP.NET applications do offer their own peculiar challenges. In this section, we'll take a close look at the tracing feature, which can be especially useful in a distributed programming context such as mobile applications where you have to deal with execution shared between a server and client.

To set a trace, adjust the line in the `Web.config` file so it sets the `Trace` to `True` and `pageOutput` to `True` (so the trace won't be sent into a file, a log in your mobile application's root directory named `trace.axd`).

```
<trace enabled='true' requestLimit="10"  
pageOutput="true" traceMode="SortByTime" localOnly="true" />
```

The `pageOutput` trace will be appended to your output page in Internet Explorer, as shown in Figure 22.8:





		Children	Children
_PAGE	ASP.MobileWebForm1.aspx	607	20
_ct0	System.Web.UI.ResourceBasedLiteralControl	0	0
Form1	System.Web.UI.MobileControls.Form	594	0
SelectionList1	System.Web.UI.MobileControls.SelectionList	0	152
Label1	System.Web.UI.MobileControls.Label	26	76

FIGURE 22.8 For the most complete report on your application's behavior, request a trace.

If you want your custom message to really stand out in the trace listing, substitute `Trace.Warn` for `Trace.Write`. Tracing can be a special tool when debugging because of the wealth of information it provides, and because it shows you a complete list of all the steps that took place during execution, their order, and their duration.

Tracing tells you these primary facts:

- ◆ How long each step takes in milliseconds (you can quickly see if there are any significant delays, or use this information to optimize your application)
- ◆ Which processes executed
- ◆ Any error messages, and specific details about these errors
- ◆ Custom trace messages you insert, along with variable values if you wish to add them (illustrated above)
- ◆ Details about variables used in the project
- ◆ Specifications about the containers and controls on each page
- ◆ Specifications about when requests happened

Trace Information Sections

When you request a trace you see six major divisions, but the Trace Information section is most often the most useful information. The Request Details section simply identifies the HTTP request type and other data about the request such as the type of character encoding (usually Unicode), when the request was made, and so on. The Control Tree lists server controls that may be on your page, and also lists child controls. If you're tracing an ordinary ASP.NET page, you would see a Cookies Collection section, but this isn't displayed for a mobile application. Instead, you see a Session State section that displays the Session ID. The Headers Collection zone lists the HTTP headers that your server sent to the client device. In ordinary ASP.NET applications, this section also includes cookie information. Finally, the Server Variables section describes your server in considerable detail, identifies quite a bit about the state of your server—its URL, connection type, and so on.

Providing Friendly Error Messages

You always want to avoid frightening your users with lengthy, technical error messages such as Object Not Found or Stack Collapse. They'll think their PDA is broken, or their phone is about to explode. Never underestimate the confusion and fear that the average person experiences after punching some buttons on a high-tech device. If anything unusual happens, many of them think they must have accidentally entered the *activation code*.

In ordinary Windows applications, you can intercept error messages and, instead of letting users see them, substitute your own user-friendly descriptions displayed in message boxes. `MsgBox` doesn't work in ASP.NET applications, though. You have to employ a different tactic. Find this section in `Web.config` :

<!-- CUSTOM ERROR MESSAGES

[Team Fly](#)

 Previous

Next 

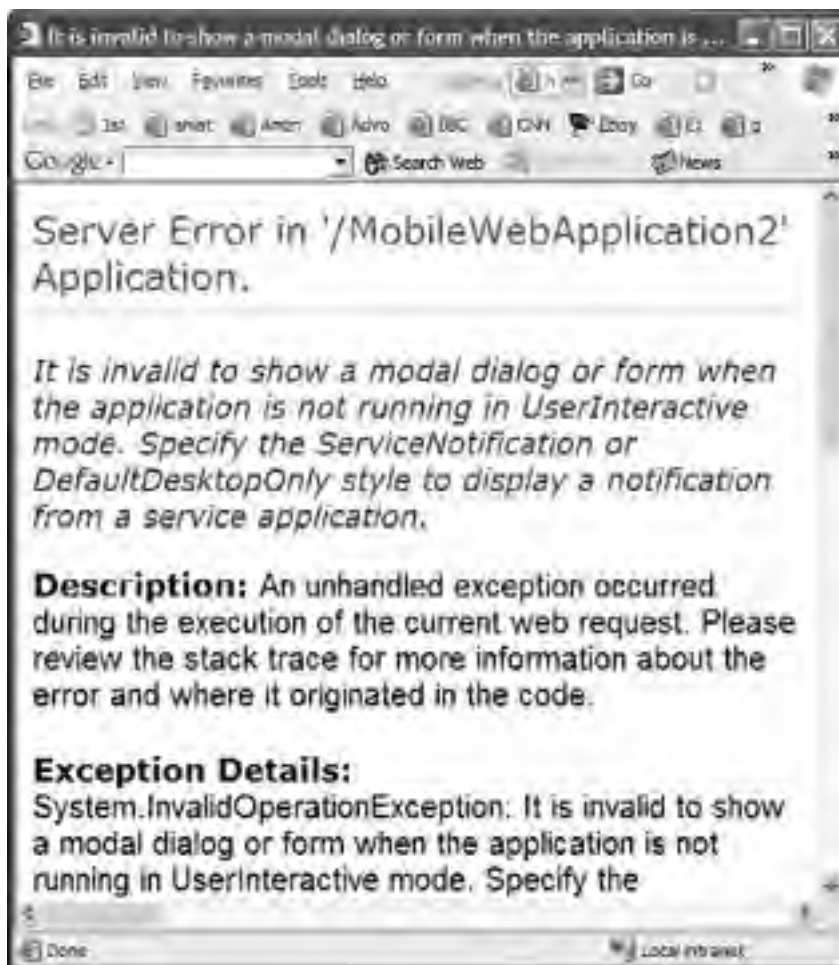


FIGURE 22.9 Some users would panic if this appeared on their cell phone.



FIGURE 22.10 Rather than the default messages, intercept errors and show users something they can understand.

Press F5 and VB.NET will choke on the MsgBox, causing an error and displaying your custom message page rather than the default page.

Device Specificity

Although word wrapping, color rendering and other features of display devices are generally automatically handled for you on most mobile devices, you must sometimes perform the old *is it Internet Explorer or is it Netscape?* branching within your code. Of course Netscape is all but dead now, but varying kinds of mobile operating systems—and varying display capabilities, input keys, and hardware features—are still alive and well. The differences between Palm and Pocket PC systems aren't trivial, not to mention the variations of output you can expect between phones and PDAs.

Similarly, some devices offer more room for text than others, and so on. You sometimes want to be able to send different sources to different devices or models. You accomplish this by defining device filters, then doing a kind of elaborate, roundabout `Select Case` or `If Then` comparison. Why this simple, common computing act of branching has to be made elaborate and unique for mobile computing programming I can't say. My guess is that this new technique probably fits in better with the invisible structures deep within .NET, and it also allows for protocols such as querying devices, asking them to tell you details about their capabilities. You can also deal with multiple devices by setting up a filter system. My only problem with this is that we all know how to use `Select Case` and `If Then`, so why set up method testing? It's sure inconvenient for us programmers up on the higher levels where everyday coding goes on.

And, if you're not yet convinced that new technologies often hand us totally unnecessary extra debugging worries, note that in VB.NET variable names, property names, and values are *not* case-sensitive. We VB.NET programmers are proud that our language doesn't introduce burdens like case-sensitivity into the language. But, alas, when you compare property values in these mobile-project filters (unless they're Boolean where *True* and *true* do match), the values *are* case-sensitive. Here's yet another exception to the traditional rules for you to memorize, or suffer later from confusing bugs in your code.

Using Emulators

You can roughly design your mobile application's user interface, and write your code-behind programming to get the kinks worked out, all as described above using Internet Explorer as the target "device." However, before deploying a mobile application, you'll doubtless want to test it with real PDAs or cell phones. But do you have to buy dozens of cell phones and PDAs? One easy way to test your mobile applications is to use emulators that, via software, mimic the I/O facilities and behaviors of a particular device.

Custom Device Emulators

You can search the Internet for devices and their emulators. Most device manufacturers make emulators available for your use. Microsoft, for example, offers an emulator for the Pocket PC from:

<http://microsoft.com/downloads/details.aspx?FamilyId=9996B314-0364-4623-9EDE-0B5FBB133652&displaylang=en>

The entire download is an SDK with which you can create and test applications using the "eMbedded" C variations or VB.NET in Visual Studio .NET 2003. After you compile your mobile application, you type your application's start page URL into the emulator.

You can find some of the most popular emulators at these locations:

YoSpace This is the place to start: www.yospace.com. They offer multiple, simultaneous WAP

emulations, including a variety of models and manufacturers. You can emulate Sony, Nokia, Motorola, Siemens, and others all at once without having to switch among various emulation environments.

Ericsson <http://www.ericsson.com/mobilityworld/sub/open/index.html>

Go.America (includes BlackBerry devices)
<http://www.goamerica.net/partners/developers/index.html>

[Team Fly](#)

 Previous

Next 

Firewall feature. If you still cannot connect to the emulator for testing, try the usual new-technology problem-solving tactics: Search Google Groups or MSDN, post a message on .NET user groups, or throw up your hands.

Summary

This chapter begins by exploring the limitations you'll face when writing programs for mobile devices—primarily PDAs and cell phones today, but who knows what tomorrow has to offer? The .NET Compact Framework is a condensed, stripped-down version of the familiar .NET Framework you're used to. You'll face memory and processor speed restrictions that you don't face when writing VB.NET applications for full-size desktops and portables.

But the most severe challenge is I/O. The keyboard on portable devices (if any) is pretty difficult to type on (so you'll want to help your users by avoiding typed input whenever possible). And the screen is very, very small. It may not even have color. How do you handle the screen compatibility problem? Programs are designed differently for black and white screens versus color.

You also saw that quite a few control properties aren't available in the Compact .NET Framework. Microsoft isn't, however, blind to these problems. For one thing, they've designed libraries specific to individual brands, and even individual models of mobile devices.

The solutions to the connectivity problem (should the connection persist throughout the session?) and security issues were discussed. Then you saw how to use the built-in mobile device simulator, how to navigate between forms, and some new features on the Toolbox when you're working with the .NET IDE within the template that Microsoft has named the ASP.NET Mobile Web Application.

You saw how to write source code for a mobile application, how to display lists in various ways, how to employ tracing during debugging, how to intercept error messages and replace them with your own, user-friendly versions, and how to use emulators so you don't have to go to the trouble of connecting to an actual mobile device during program development.

Index

Note to the reader: Throughout this index **boldfaced** page numbers indicate primary discussions of a topic. *Italicized* page numbers indicate illustrations.

Symbols

- * (asterisk), in regular expressions, [543](#), [558](#)
- . (period), in regular expressions, [543](#), [545](#), [558](#)
- ? (question mark), as metacharacter, [560](#)
- | (pipe symbol), for alternation in regular expressions, [563–564](#)

A

- Abort method, of MessageQueueTransaction class, [371–372](#)
- aborting transaction, [437](#)
- Accept method, of Socket class, [298](#)
- AcceptChanges method, of DataSet, [420](#), [425–426](#)
- AcceptTcpClient method, [302](#)
- AcknowledgeTypes property, of message, [358](#), [359](#), [360](#), [363](#)
- acknowledgments for queued messages, [358–373](#), [361](#)
 - fault tolerance and load balancing, [366–370](#)
 - processing, [361–366](#)
 - requesting, [358–361](#)
 - retrieving for specific message, [350](#)
 - timeout, [359](#)
 - transactional messages, [371–373](#)
- action queries, [407](#)
 - executing, [409](#)
- Activator class, CreateInstance method of, [211](#)
- ActiveX controls, use with .NET clients, [462–465](#)
- ActiveX Data Objects. *See* ADO.NET
- Add Dialog dialog box, [268](#), [268](#)
- Add method, of Rows collection of DataTable, [419](#)
- Add New Project dialog box (Visual Studio), [261](#), [261](#)

- Add Or Remove Programs snap-in, [259](#), [260](#)
- Add Project Output Group dialog box, [263](#)–[264](#), [264](#)
- Add Reference dialog box, [63](#)
- Add Web Reference dialog box, [456](#), [456](#)
- AddDays method, of DateTime class, [39](#)
- AddHours method, of DateTime class, [38](#)
- addition of complex numbers, [620](#)
- AddMinutes method, of DateTime class, [38](#)
- AddOrder method, in middle tier, [446](#)
- AddressFamily enumeration, [293](#)
- AddressList property, of IPHostEntry class, [291](#)
- AddValue method, of SerializationInfo class, [79](#)
- AdministrationQueue property, of message, [358](#)
- administrator, and security, [120](#), [122](#)
- ADO.NET
 - accessing databases, [391](#)–[415](#)
 - Command class, [409](#)–[415](#)
 - Connection class, [402](#)–[403](#)
 - DataAdapter class, [404](#)–[408](#)
 - Visual database tools, [392](#)–[402](#)
 - DataSets, [415](#)–[427](#)
 - accessing tables, [416](#)–[417](#)
 - adding and deleting rows, [419](#)–[420](#)
 - binding to DataGrid control, [277](#)–[278](#)
 - creation, [395](#)–[396](#)
 - DataViews, [426](#)–[427](#)
 - locating rows, [420](#)–[421](#)
 - multiple tables for, [396](#)–[400](#)
 - navigation, [421](#)–[426](#)
 - null values, [418](#)–[419](#)
 - rows, [417](#)–[418](#)
 - viewing, [396](#)
 - Insert and Update operations, [428](#)–[440](#)
 - DataAdapter for transactions, [436](#)–[440](#)
 - DataAdapter to update, [428](#)–[430](#)
 - Identity columns, [430](#)–[436](#)
- Advanced SQL Generation Options window, [407](#), [407](#)
- aesthetics, [xix](#)–[xx](#)
- alerts, suppressing, [100](#)
- Aliases property, of IPHostEntry class, [292](#)

Alignment property, for string printing, [169](#), [171](#)

All Code membership condition, [256](#)

AllDBNull property, of DataColumn class, [417](#)

AllowSelection property, of PrintDialog control, [166](#)

AllowSomePages property, of PrintDialog control, [166](#)

AlternatingBackColor property, of DataGrid control, [281](#)

- attributes in XML
 - data types for, [483](#)
 - deleting, [488](#)
- <AttributeType> element (XML), [483](#)
- auditing messages in queues, [373](#)
- authentication, [339](#)
 - in .NET Compact Framework, [652](#)
- Authenticode, [121](#)
- Author property, of Setup Project, [266](#)
- authorization, of user for database connection, [402–403](#)
- AutoGenerate Columns property, of DataGrid control, [280](#)
- AutoIncrement property, of DataColumn class, [417](#), [431](#)
- Automatic Transaction Processing service, [468](#)
- autopostback attribute, for ListBox control, [281](#)
- Autos window for debugger, [238](#), [239](#)
- AxisNumberColor property, [592](#)
- AxisNumberFont property, [592](#)
- AxisTitleColor property, [592](#)
- AxisTitleFont property, [592](#)

B

- backreferences, in regular expressions, [571–572](#)
- BaseURI property, of XMLReader object, [477](#)
- BasicSerialization project, [61–65](#), [62](#)
- batch mode for database updates, [428](#)
- batch query, [414](#)
- beep, metacharacter for, [560](#)
- Begin method, of MessageQueueTransaction class, [371–372](#)
- BeginGetResponse method, of HttpRequest object, [315](#)
- BeginReceive method, for messages in queues, [356](#)
- BeginTransaction method, of Connection object, [436](#)
- bell, metacharacter for, [560](#)
- binary file
 - for application configuration files, [76](#)
 - deserialization into ArrayList collection, [64](#)
- binary serialization, [60](#)

- of ArrayList collection, [62–64](#)
- for messages in queues, [352](#)
- and type fidelity, [65](#)
- BinaryFormatter class, [60](#), [61](#)
 - Serialize method, [63](#)
- BinaryReader, [44](#)
- BinaryWriter, [44](#)
- Bind method, of Socket class, [294](#)
- binding
 - array to SelectionList control, [651](#)
 - controls to DataTables, [509](#)
- Binding element in WSDL, [331](#)
- BindingFlags, [195–198](#)
- BitBlt GDI32 function, [186](#), [187](#)
- bitmaps
 - capturing window or desktop to, [188](#)
 - for persistent graphics, [589](#)
 - printing centered, [189](#)
- biztalk (Microsoft), [481](#)
- Blackberry devices, emulators, [660](#)
- Body property, of Message object, [347](#)
- body text, font for, [625](#)
- boldface in user interface, [627](#)
- boolean value type, [14](#)
- Bounds property, of PageSettings object, [162](#)
- braces ({}), [4](#)
 - to fill array with values, [20](#)
- Break Condition window, [238](#), [238](#)
- breakpoints for debugging, [237–238](#)
 - and display, [240](#)
- BreakPoints window, [238](#)
- Browse With dialog box, [662](#)
- business logic, [442](#), [443](#)
 - remoting, [449–461](#)
- business rules, [443–445](#)
 - adding to middle tier, [532–535](#)
- business tier. *See* middle tier component
- BusinessLayer class
 - converting to Web service, [450–457](#)
 - GetItemDiscount method, [533](#)

GetItemDiscount stored procedure, [534](#)
remoting, [458–461](#)
BusinessLayer project, [450–457](#)
buttons
 depth for, [628](#)
 etched text on, [625](#)
 in .NET Compact Framework, [644–645](#)
byte, [14](#)
byte arrays
 converting strings to, [311](#)
 converting to string, [308](#)
 for DES, [145](#)

C

- C programming language, [1](#), [2](#)
- C# programming language, translating to VB.NET, [21](#)
- Cab Project, as New project option, [261](#)–[262](#)
- cautil command, [245](#)
- Call Stack window, debugging with, [240](#), [241](#)
- CancelEdit method, [421](#)
- CanDuplex property, of PrinterSettings object, [163](#)
- Capacity property, of ArrayList, [36](#)
- capacity, vs. dimension, [28](#)
- capturing
 - errors, [217](#)
 - matches for regular expressions, [573](#)
 - multiple captures, [573](#)–[575](#)
- carriage return, metacharacter for, [560](#)
- CAS (code-access security), [124](#), [125](#)–[128](#)
 - config files, [127](#)
- case sensitivity
 - in .NET Compact Framework, [660](#)
 - in regular expressions, [545](#), [577](#)
- caspol.exe, [129](#), [252](#)
- Catch statement, [217](#), [218](#)
- Category property, of MessageQueueCriteria class, [349](#)
- cell phones. *See also* mobile computing
 - emulators, [660](#)–[663](#)
- centering when printing bitmap, [189](#)
- char value type, [14](#)
- character counting
 - to fit in rectangle, [178](#)
 - in Word document, [106](#)
- characters in regular expressions, [558](#)
 - escaping metacharacter to treat as, [563](#)
 - ranges of, [559](#)–[560](#)
- chat. *See* TcpChatClient application; TcpChatServer application
- ChatClass, [301](#), [302](#)–[303](#)

- ChildKeyConstraint property, of Relation class, [431](#)
- ciphertext, [143](#)
- classes, [14](#), [18](#)
 - in .NET Framework, XML, [475](#)
 - converting to Web service, [450](#)
 - descriptions, [5–6](#)
 - in DOM (Document Object Model), [479](#)
 - searching for members or data, [207–208](#)
 - viewer for, [20–21](#)
- ClassSerializer project, [66](#), [66–71](#)
 - Book class, [67–70](#)
 - deserializing individual objects, [71](#)
 - serializing individual objects, [70–71](#)
- Clear method, of DataAdapter, [404](#)
- CLI (Common Language Infrastructure), [214](#)
- client computer
 - installing .NET Framework on, [244](#)
 - storing data on, [412](#)
- client/server architecture, [441](#), [442](#)
- Clipboard
 - code to access, [23](#)
 - for spell-check, [95](#)
- ClipBounds property, of Graphics object, [173](#)
- closed schema model, [485](#)
- closing connection, [402](#)
- cloud services, [337](#)
- CLR. *See* Common Language Runtime (CLR)
- code
 - access to, [134](#)
 - demanding permission through, [257](#)
 - to implement typed DataSet, [415](#), [416](#)
 - reuse by components, [444](#)
- code access permissions, [251–257](#)
- code-access security, [124](#), [125–128](#)
 - config files, [127](#)
- Code Access Security Policy Tool, [129](#), [252](#)
- code-behind window, of WebForm, [273](#)
- Collate property, of PrinterSettings object, [163](#)
- collections
 - for controls, [28](#)

- zero- vs. one-based, [xxi](#)
- collisions, namespaces to avoid, [193](#), [480](#)
- color
 - for fractals, [614–615](#)
 - in user interface, [627](#)
- Color class, [18](#)
- Color property, of PageSettings object, [162](#)
- column headers, printing, [185](#)
- ColumnName property, of DataColumn class, [417](#)
- Columns collection, [417](#)
- COM+ applications, [465–473](#)
 - COMPlus component, [466–467](#)
 - exporting proxy, [467–468](#)
- COM+ Component Install Wizard, [467](#)

custom objects, in middle tier, [444](#)
CustomValidator control, [283](#)

D

Data Adapter Configuration Wizard, [394](#), [394](#)

data binding, [36–37](#), [505–506](#)

data display on WebForm, [273–283](#)

connecting to database, [273–274](#)

DataGrid control, [276–281](#)

DataList control, [275](#)

detecting postback, [281–282](#)

Repeater control, [276](#)

templates, [275](#)

Data Encryption Standard (DES), [142](#), [143](#)

encrypting and decrypting file with, [143–145](#)

data entry, [216](#)

to robust applications, [215](#)

validation, [282](#)

Data Link Properties dialog box, [392–393](#), [393](#)

data types, [11–18](#)

color as, [17–18](#)

mixing in single stream, [55–56](#)

strong typing and mismatch, [17](#)

in WSDL, [332](#)

in XML, [483–486](#)

data validation, [225](#). *See also* validation

DataAdapter class (ADO.NET), [404–408](#)

configuration, [393–394](#)

for transactions, [432–440](#)

Update method, [400](#), [404](#), [425](#), [428–430](#)

database. *See also* ADO.NET

connection, [392–393](#)

in client/server model, [442](#)

immediate update of, [508](#)

insert and update operations, [428–440](#)

DataAdapter for transactions, [436–440](#)

- DataAdapter for updating, [428–430](#)
- Identity columns, [430–436](#)
- SQL connection to, [273–274](#)
- update frequency, [408](#)
- Web services connection to, [326–330](#)
- database server, [441](#)
- DataColumn objects, [415](#), [417](#)
- Data.DataException, [227](#)
- Data.DBConcurrencyException, [227](#)
- DataGrid control, [276–281](#)
 - AlternatingBackColor property of, [281](#)
 - AlternatingItemStyle property of, [281](#)
 - appearance of, [278–280](#), [279](#)
 - AutoGenerate Columns property, [280](#)
 - binding DataSet to, [277–278](#), [396](#), [399](#)
 - detecting postback, [281–282](#)
 - formatting, [401–402](#)
 - limitations, [505](#)
 - on hierarchy display, [535](#)
 - mouse and, [516](#)
 - specifying behaviors, [278](#)
- DataList control (ASP.NET), [275](#)
- DataReader class (ADO.NET), [392](#), [412](#), [413–415](#)
- DataRelation objects, [421](#)
- DataRelations project, [422–424](#), [423](#)
- DataRow objects, [415](#), [417–418](#)
- DataRowState enumeration, [424](#), [425](#)
- DataRowVersion enumeration, [425](#)
- DataSet class (ADO.NET), [391–392](#)
- DataSets, [415–427](#)
 - accessing tables, [416–417](#)
 - adding and deleting rows, [419–420](#)
 - binding to DataGrid control, [277–278](#), [396](#)
 - converting to XML, [504](#), [504](#)
 - converting XML to, [503](#), [504](#)
 - creation, [395–396](#)
 - DataViews, [426–427](#)
 - editing with constraints, [421](#)
 - HasErrors property of, [429](#)
 - locating rows, [420–421](#)

multiple tables for, [396–400](#)
navigation, [421–426](#)
null values, [418–419](#)
to pass data between tiers, [444](#)
rows, [417–418](#)
typed vs. untyped, [416](#)
updating database from, [400–402](#)
viewing, [396](#)
and XML, [493–495](#)

DataSource property, for binding controls, [509](#)

Data.SqlClient.SqlException, [227](#)

Data.SqlTypes.SqlTypeException, [227](#)

DataTable objects, [415](#), [416–417](#)

DisconnectedOrders application, [374](#), [374–375](#)
discount policy component, in middle tier, [447–449](#)
DiscountServer project, [458–461](#)
DisplayName property, for binding controls, [509](#)
distributed applications, [2](#). *See also* middle tier components
distributed code, [125](#)
DivideByZeroException, [226](#)
division by zero, [234](#)
DLL files, and XCopy deployment, [245](#)
DnsPermission code access permission, [252](#)
documentation. *See* help
Documentation element in WSDL, [331](#)
DocumentElement property, [489](#)
documents collection, and Word object model, [104](#)
documents, loading into TextBox, [99–100](#)
DOM (Document Object Model), [475](#), [478–479](#)
 classes, [479](#)
Dotnetfx.exe setup program, [244](#)
dotted-quad notation, [290](#)
double-sided printing, printer support for, [163](#)
double value type, [14](#)
Download method, of WebClient class, [311](#)
DownloadData method, of WebClient class, [308](#), [311](#)
DownloadFile method, of WebClient class, [308](#)
downloading. *See also* Internet-based deployment process
 assemblies on demand, [258–259](#)
 documents with WebClient, [311–312](#)
DrawRectangle method, [161](#)
DrawString method, [161](#), [168–169](#)
Duplex property, of PrinterSettings object, [163](#)
duplicate entries, preventing, [16](#)
duplicate rows in invoice, combining, [531–532](#)
Dynamic Host Configuration Protocol (DHCP), [290](#)

E

e-mail addresses, regular expression for, [568](#)

- e-mail messages, getting from inbox, [109–110](#)
- ECMAScript member of RegexOptions enumeration, [549](#)
- Edit Relation dialog box, [398](#), [398](#)
- editing row in DataTable, [418](#)
- ElementName property, of XmlElement, [74](#)
- elements in XML, deleting, [488–489](#)
- <ElementType> element (XML), [484](#)
- embossed frames, [632–633](#)
- emulators in mobile computing, [660–663](#)
 - Pocket PC in Visual Studio, [661](#), [661–662](#)
 - problems, [662–663](#)
- enableSession argument, in WebMethod element, [326](#)
- encryption, [143–151](#)
 - in .NET Compact Framework, [652](#)
 - asymmetrical, [151–158](#)
 - DES (Data Encryption Standard), [143–145](#)
 - hashing with, [147–151](#)
 - initialization vectors for DES, [146–147](#)
 - key length, [147](#)
- End, [4](#)
- end of file, determining, [41–42](#)
- end of line, metacharacter for, [562](#)
- End Try statement, [217](#)
- EndGetResponse method, [315](#)
- enterprise level for security policy, [251](#)
- enterprisesec.config file, [127](#)
- EnterpriseServices class, [470](#)
- enum value type, [14](#)
- enumerations
 - AddressFamily enumeration, [293](#)
 - CommandType enumeration, [409](#)
 - DataRowState enumeration, [424](#), [425](#)
 - DataRowVersion enumeration, [425](#)
 - DataRowStates enumeration, [427](#)
 - Keys enumeration, [47](#)
 - modifying member names, [75](#)
 - reflection and, [205](#)
 - RegexOptions enumeration, [549](#)
 - SocketType enumeration, [294](#)
 - StreamingContextState enumeration, [80](#)

enumerators, [37](#)

EnvironmentPermission code access permission, [252](#)

EOF property, [41](#)

equality, [14–16](#)

 vs. identity, [13](#)

Equals method, [12–13](#)

ergonomics, [626](#)

Ericsson, emulators, [660](#)

error messages, [3](#)

 "The application attempted to perform an operation
 not allowed by the security policy", [250](#), [250](#)

 "login failed", from MSDE, [329](#)

- in .NET Compact Framework, [656–658](#), [658](#)
- from object instantiation, [8](#)
- "Object reference not set to an instance of an object", [8](#), [19](#)
- "Overload resolution failed because no accessible 'New' accepts this number of arguments", [9](#)
- "permission denied", from MSDE, [329](#)
- "An unhandled exception of type 'System.Runtime.InteropServices.COMException' ...", [95](#)
- from validation controls, [284](#)
- ValidationSummary control, [284](#)
- errors. *See also* structured exception handling
 - in database update process, [429](#)
 - preventing, [225](#)
 - in programming, [232–236](#)
- escape, metacharacter for, [560](#)
- escape velocities, and color, [614–615](#)
- escapes, from Mandelbrot Set, [608](#), [612](#)
- escaping metacharacters, in regular expressions, [563](#)
- etched frames, [632–633](#)
- EventLogPermission code access permission, [252](#)
- events
 - adding to runtime–created controls, [26–27](#)
 - handling multiple controls with single, [27–28](#)
- Everything permission set, [251](#)
- evidence, [251](#)
- Excel, [111–116](#)
 - evaluating math expressions, [111–113](#)
 - pretty printing, [113](#)
 - retrieving data, [115–116](#)
 - sending data to, formatting, calculating and saving, [113–115](#)
- Exception class, [218](#), [225–228](#)
- exception handling. *See also* structured exception handling for regular expressions, [548](#)
- Exceptions dialog box (Debugger), [230–231](#), [231](#)
- executables
 - compiled for release, [232](#)
 - partial trust of, [125](#)
- Execute method, of Command class, [410–411](#)

- ExecuteNonQuery method, of Command class, [409](#)
- ExecuteReader method of Command class, [409](#)
 - DataReader object from, [414](#)
- ExecuteScalar method, of Command class, [409](#), [412–413](#)
- ExecuteXmlReader method, of Command class, [409](#)
- executing code, discovered through reflection, [208–213](#)
- Execution permission set, [251](#)
- Exists method, of MessageQueue class, [347](#)
- explicit declaration of XML namespace, [480](#)
- Explicit option for module, [232–233](#)
- ExplicitCapture member of RegexOptions enumeration, [549](#)
- exporting
 - proxy, [467–468](#)
 - TextBox content to Clipboard, [23](#)

F

- fading transitions in user interface, [637](#), [637–640](#)
- fault tolerance, in MSMQ, [366](#)
- fax, Word's Wizard to send, [98](#)
- fields
 - in database, setting to null value, [401](#)
 - excluding from serialization, [53](#)
- file stream, for DES, [145](#)
- File System editor, [262](#), [263–266](#)
- File Types editor, [262](#)
- FileDialogPermission code access permission, [252](#)
- FileGet command, [41–42](#)
- FileIOPermission code access permission, [252](#)
- filenames, filling ListBox with, [101–102](#)
- FileNotFoundException, of Open method, [220](#)
- files
 - Directory object for getting list, [11](#)
 - finding, using Word objects, [100–102](#)
 - hashing, [142–143](#)
 - input/output, [39–44](#). *See also* streaming
 - determining end, [41–42](#)
 - reading, [40–41](#)
 - writing, [42–44](#)
 - loading assembly from, [201–203](#)

loading from assembly, [200–201](#)
reading attributes of, [224](#)
as Response object arguments, [272](#)
runtime errors when opening, reading and writing, [219–224](#)
 Open method exception handling, [219–221](#)
size in binary vs. SOAP serialization, [65](#)
tampered, [142](#)
on target computer, setup project impact on, [262](#)

- FileSearch method, [101](#)
- FileStream object
 - constructor, [9](#)
 - instantiating, [7](#)
 - Read method of, [221](#)
- Fill method, of DataAdapter, [395](#), [415](#)
- filter
 - for DataView object, [426](#)
 - for e-mail messages, [110](#)
 - for reflection results, [195–198](#)
- Finalize method, [12](#)
- Finally clause, [219](#)
- Find method, of DataView object, [427](#)
- FindBy method, of DataTable object, [420](#)
- finding files with Word objects, [100–102](#)
- FindRows method, of DataView object, [427](#)
- firewall, [339](#)
 - serialization and, [61](#)
- fixed IP address, [290](#)
- FlatStyle property, of buttons, [628](#)
- focus group, for user interface guidelines, [626](#)
- folders, adding items to user's, [267](#)
- font, for TextBox control, [174](#)
- Font property, [175](#)
- FontBold off, as user interface convention, [627](#)
- FontName property, of TextBox control, [625](#)
- For statement, declaration included within, [116](#)
- FOR XML AUTO clause, [82](#), [83](#)
- foreign keys, rules on changing, [431](#)
- forking, [332](#), [481](#)
- Form1_load event, [4](#)
- FormatFlags property, of StringFormat object, [169](#)
- FormatName property, of remote queue, [343](#)
- formatting
 - DataGrid control, [401–402](#)
 - Word for manipulating, [102–104](#)
- formfeed, metacharacter for, [560](#)

- forms, [193](#)
 - adding controls at runtime, [25–27](#)
 - collections, for controls, [28](#)
 - communication between, [45–46](#)
 - KeyPreview property of, [187](#)
 - KeyUp event handler of, [187](#)
 - for mobile computing, [646–647](#)
- Forth, [481](#)
- fractal generator, [602–620](#)
- fractals, [590](#)
 - coloring, [614–615](#)
 - explained, [602–606](#)
 - exploring, [616–620](#)
 - Julia Sets, [609](#), [612](#), [612–616](#), [613](#), [619–620](#)
 - programming, [613–614](#)
 - Mandelbrot Set, [605](#), [607](#), [607–612](#), [617–619](#)
 - programming, [608–612](#)
 - transformation process, [604–605](#)
 - complex numbers in, [605–606](#)
 - zoom operation on, [616](#)
- frames in user interface, [630–634](#)
 - multiple, [634](#), [634](#)
- Framework Configuration tool, [129–133](#), [130](#)
 - Adjust Security option, [132–133](#)
 - New and Open options, [131](#)
 - Reset All option, [132](#)
- Friend modifier, [80](#)
- FromODate method, of DateTime class, [38](#)
- FromPage property, of PrinterSettings object, [163](#)
- FromXMLString method, [157](#)
- Full Trust, [133](#)
- FullReachQueue member of AcknowledgeTypes enumeration, [359](#)
- FullReceive member of AcknowledgeTypes enumeration, [359](#)
- FullTrust permission set, [251](#)
- FunctionColors collection, in PlotControl application, [592](#)
- FunctionLineWidths collection, in PlotControl application, [592](#)
- functions
 - calculating path, [597–598](#)
 - calculating Y axis range, [594](#)
 - evaluating at runtime, [593–595](#)

singularities, [602](#)

Functions collection, in PlotControl application, [592](#)

FunctionStyles collection, in PlotControl application, [592](#)

G

GAC (global assembly cache), [243](#)

installing DLL in, [245](#)

genericerror.aspx file, for custom error messages, [657](#)

GenericPrincipal class, [122](#)
GetAllMessages method, [358](#)
GetAssembly method, [198](#)
GetAttributes method, of File class, [224](#)
GetBytes method, of System.Text.Encoding.ASCII class, [311](#)
GetChanges method, of DataTable object, [425](#)
GetChildRows method, of DataRow object, [422](#)
GetDC function, [187](#)
GetDirectories method, of DirectoryInfo class, [11](#)
GetFiles method, [11](#)
GetHashCode method, [12](#), [16](#)
GetHostByAddress method, of IPHostEntry class, [292](#)
GetHostByName method
 of Dns class, [291](#)
 of IPHostEntry class, [292](#)
GetItemDiscount method
 of BusinessLayer component, [447–448](#)
 testing, [454](#), [455](#)
GetItemDiscount stored procedure, [448–449](#)
GetLowerBound method, of Array class, [34](#)
GetMessageQueueEnumerator method, [349](#)
GetObjectData method, of ISerializable interface, [79–81](#)
GetPrivateQueuesByMachine method, [348](#)
GetProductByID method, in middle tier, [446](#)
GetProductsByName method
 in middle tier, [446](#)
 limiting number of rows returned, [456](#)
GetPublicQueues method, [348](#)
GetPublicQueuesByCategory method, [348](#)
GetPublicQueuesByLabel method, [348](#)
GetPublicQueuesByMachine method, [348](#)
GetRequestStream object, [314](#)
GetResponseStream method, of WebResponse object, [314](#)
GetString method, of System.Text.Encoding.ASCII class, [311](#)
GetType method, [12](#), [16](#), [205](#)
 to instantiate assembly, [198](#)
global assembly cache (GAC), [243](#)

- installing DLL in, [245](#)
- global variables, [15](#)
- Go.America, emulators, [660](#)
- GoTo statement, [224](#)
- gradient brush, [173](#)
- gradient metallic shading, [635](#)
- grammar, [6–11](#)
- graphics, [589](#)
 - file format for mobile computing, [647](#)
 - LoadPicture to load, [48](#)
 - sending, [286–287](#), [287](#)
- Graphics Device Interface (GDI+), [589](#)
- Graphics object
 - ClipBounds property of, [173](#)
 - PageUnit property, [168](#)
 - for printing, [168](#)
 - SetClip method of, [162](#)
- Graphics property, of PrintDocument object, [160](#)
- GraphicsPath object, [591](#)
- greedy regular expressions, [561](#)
- greedy subexpression, metacharacter for, [578](#)
- grep utility (Unix), [582](#)
- GridColumnStyle property, [401](#)
- grids, drawing on PictureBox control, [598–602](#)
- <group> element (XML), [486](#)
- grouping in regular expressions, [568–573](#), [569](#)
- groups, [122](#)
- GUIDs (globally unique identifier)
 - and Identity columns, [431](#)
 - for messages in queues, [345](#), [350](#)
- GUIDs (globally unique identifiers), [419](#)

H

- hackers, [134](#), [320](#)
- handled exceptions, [216](#)
- handles, [46–48](#)
 - key press detection, [47–48](#)
 - runtime, [47](#)
- Handles command, [27–28](#)

hardware, errors from, [216](#)
HasAttributes property, of XMLReader object, [477](#)
HasErrors property, of DataSet object, [429](#)
Hash membership condition, [256](#)
hashcodes, [16](#)
 for password, [77](#)
hashing, [140](#)
 with encrypting, [147–151](#)
 files, [142–143](#)
 passwords, [140–142](#)
hashtables, [37–38](#)
HasMorePages property, [162](#)

HasValue property, of XMLReader object, [477](#)
header of column, printing, [185](#)
headlines in applications, sans serif font for, [627](#)
help, [3–5](#), [18](#)
"high encryption pack", [140](#)
Hit Count property, for debugging, [237](#)
HKEY_LOCAL_MACHINE\Software\[Manufacturer] key, [267](#)
HostName property, of IPHostEntry class, [292](#)
hostnames, [290](#)
HTML (Hypertext Markup Language), from server controls, [272–273](#)
HTML controls, [287–288](#)
HttpRequest class, [314](#)
HttpResponse class, [314](#), [315](#)

I

Icon property, for application shortcut, [266](#)
icons, [624](#)
ICryptoTransform object, [146](#)
ID property, of Message class, [350](#)
Identity columns, [430–440](#)
 in DataSet, [419](#)
 and GUID (globally unique identifier), [431](#)
identity, vs. equality, [13](#)
idref data type in XML, [483](#)
IgnoreCase member of RegexOptions enumeration, [549](#)
IgnorePatternWhiteSpace member of RegexOptions
 enumeration, [549](#)
Image control, in .NET Compact Framework, [647](#)
Image object, copying screen onto, [186](#)
ImageUrl property, [647](#)
imaginary number, [603–604](#)
imperative code access, [124](#)
implicit declaration of XML namespace, [480](#)
importing
 API functions, [187](#)
 data from Excel, [116](#)

- namespaces, [193](#)
- Imports statement
 - for ASP.NET Web application, [273](#)
 - need for, [11](#)
 - for remoting project, [460](#)
 - for security examples, [140](#)
 - for serialization, [54](#)
 - for XML, [487](#)
- Index property
 - of control, [24](#)
 - of Match class, [550](#)
- IndexOf method
 - help sample code, [3–4](#)
 - of String class, [543](#)
- IndexOutOfRangeException, [29](#)
- infinity, [234](#)
 - testing for, [235–236](#)
- inheritance, [12](#)
- inherited controls, appearance of, [176](#)
- initialization, [30](#)
- initialization vectors for DES, [146–147](#)
- InnerException property, [225](#)
- INSERT statement (SQL), [405](#)
 - DataAdapter task for, [404](#)
- InsertAfter method
 - in Word object model, [104](#)
 - for XML elements, [489](#)
- InsertAt method, of Rows collection of DataTable, [421](#)
- InsertBefore method, for XML elements, [489](#)
- InsertCommand property, of DataAdapter, [404](#), [435](#)
- InstalledPrinters method, of PrinterSettings object, [163](#)
- installing .NET Framework runtime, [244–245](#)
- instantiation
 - of FileStream object, [7](#)
 - of objects, [8](#)
- integers, [14](#)
- IntelliSense list, [105](#)
- interfaces, [14](#), [193](#)
 - in DOM specification, [478](#)
- Intermediate Language (IL), [213](#)

Internet addressing, [289–292](#)

Internet-based deployment process, [246–259](#)

assembly download on demand, [258–259](#)

code access permissions, [251–257](#)

preparation, [247–249](#)

running application, [257–258](#)

Windows application deployment on Web server, [249–250](#)

Internet Explorer, security settings, [122](#)

Internet Information Services snap-in, [249](#)

Internet permission set, [251](#)

[Team Fly](#)

 Previous

Next 

Internet sockets, [292](#)
Internet zone, permissions for code from, [128](#)
interoperability, [93](#)
intranet zone, [128](#)
InvalidCastException, [227](#)
invoicing application, [516–535](#)
 architecture, [518–525](#)
 interface, [516–518](#), [518](#)
Invoke method, [213](#)
IOException
 of FileStream.Read method, [221](#)
 of Open method, [220](#)
IO.InternalBufferOverflowException, [227](#)
IO.IOException, [227](#)
IP addresses, [289–290](#)
 for local computer, [291](#)
IPAddress class, [291](#)
IPCONFIG utility, [290](#), [290](#)
IPEndPoint class, [291](#)
IPHostEntry class, [291](#)
Is comparison operator, [13](#)
ISBN values, regular expression for, [577](#)
IsContactNameNull method, for typed DataSet, [418](#)
IsDefault property, of XMLReader object, [477](#)
IsDefaultPrinter property, of PrinterSettings object, [163](#)
IsEmptyElement property, of XMLReader object, [477](#)
ISerializable interface, [79](#)
IsInfinity method, [235–236](#)
IsNaN method, [235–236](#)
IsNegativeInfinity method, [236](#)
IsNull method, of DataRow object, [418](#)
IsNullable property, of XmlElement, [74](#)
IsolatedStorageFilePermission code access permission, [252](#)
IsPlotter property, of PrinterSettings object, [164](#)
IsPositiveInfinity method, [236](#)
IsValid property, of PrinterSettings object, [164](#)
Item property, of DataRow object, [417](#)

iteration through rows of DataSet, [417](#)

J

journal message queues, [343](#)

referencing, [345–346](#)

Julia Sets, [609](#), [612](#), [612–616](#), [613](#), [619–620](#)

programming, [613–614](#)

Just In Time Activation (JITA), [468](#)

Just In Time (JIT) compilation, [121](#)

K

key distribution center, [152](#)

key length, [140](#)

key transfer, [152](#)

KeyChar property, [47](#)

KeyPreview property, of forms, [187](#)

Keys enumeration, [47](#)

KeyUp event handler, of forms, [187](#)

Kind property, of PaperSize object, [163](#)

L

label for queue, [345](#)

Label property

of Message class, [347](#), [350](#)

of MessageQueueCriteria class, [349](#)

landscape mode, [113](#)

Landscape property, of PageSettings object, [162](#)

LandscapeAngle property, of PrinterSettings object, [164](#)

laser printers, minimum margin, [168](#)

LastIndexOf method, of Array class, [34](#)

Launch Conditions editor, [262](#)

layering, as user interface convention, [627](#)

layers, vs. tiers, [443](#)

Length property, [29](#)

of Match class, [550](#)

library application, creating, [466](#)

License Agreement dialog box, [269](#)

lighting in user interface, [628–629](#)

LineAlignment property, for string printing, [169](#)

LineLimit property, for string printing, [171](#)

List controls, for mobile computing, [650–651](#)

ListBox control

 autopostback attribute for, [281](#)

 binding to columns, [506](#)

 filling with filenames, [101–102](#)

 SelectedIndexChanged event handler, [537–539](#)

Listen method, of Socket class, [297](#), [298](#)

ListenClass, [301](#)

ListView control

 adding print capabilities, [179](#), [179–186](#)

 handling user actions on, [527–528](#)

- for invoicing application, [516–517](#)
- relations mapped on, [539–542](#), [540](#)
- load balancing, in MSMQ, [367–370](#)
- load-balancing software, [449](#)
- LoadFrom method, of Assembly class, [203](#), [258](#)
- LoadPicture method, [48](#)
- local area network, IP address on, [290](#)
- local computer
 - IP address for, [291](#)
 - queues on, [348–349](#)
- LocalIntranet permission set, [251](#)
- LocalName property, of XMLReader object, [477](#)
- Locals window for debugger, [238](#)
- localSocket object, [297](#)
- locked rows in database
 - from pessimistic concurrency, [408](#)
 - for transactions, [436](#)
- logical errors, [231](#), [237–240](#)
- "login failed" error message, from MSDE, [329](#)
- logon dialog box, for mobile computing, [653](#), [653](#)
- lookahead assertions, [575–578](#)
 - metacharacters for, [578](#)
- lookbehind assertions, [575–576](#)
 - metacharacters for, [578](#)
- loosely coupled system, [341](#), [342](#)

M

- machine level for security policy, [251](#)
- MachineName property, of MessageQueueCriteria class, [349](#)
- macros in Word, to view VB code, [103–104](#)
- MailItem objects, [110](#)
- MajorGridWidth property, in PlotControl application, [592](#)
- MajorXTicks property, in PlotControl application, [592](#)
- MajorYTicks property, in PlotControl application, [592](#)
- managed code, [121](#), [462](#)
- Mandelbrot Set, [605](#), [607](#), [607–612](#), [617–619](#)

- programming, [608–612](#)
- ManufacturerURL property, of Setup Project, [266](#)
- MarginBounds property, of PrintDocument object, [160](#)
- margins, [161](#)
 - minimum for laser printers, [168](#)
- Margins property, of PageSettings object, [162](#)
- Match class, properties, [550](#)
- Match method, of RegEx class, [551–552](#)
- Matches method, of RegEx class, [548](#), [550–551](#)
- MatchEvaluator function, [553](#)
- MatchEvaluator project, [554–557](#), [555](#)
- math
 - Excel to evaluate expressions, [111–113](#)
 - overflow exception in calculation, [218](#)
 - and programming, [xx](#)
 - transformations, [604–605](#)
- Math object, [15](#)
- Max method, [15](#)
- MaximumCopies property, of PrinterSettings object, [164](#)
- MaximumPage property, of PrinterSettings object, [164](#)
- MaxLength property, of DataColumn class, [417](#)
- Me, [45](#)
 - Me.Controls collection, [28](#)
- MeasureString method, [169–170](#), [185](#)
- MeasureString property, [175](#)
- MemberAccessException, [227](#)
- MemberwiseClone method, [12](#)
- MemoryStream, [61](#)
- Merge Module Project, as New project option, [261](#)
- Message class, [349–358](#)
 - creating and sending messages, [352–358](#)
 - deleting messages, [357](#)
 - with MessageEnumerator class, [355](#)
 - peeking at messages, [357–358](#)
 - properties, [350–351](#)
 - reading messages, [354–355](#)
 - retrieving messages asynchronously, [356–357](#)
- Message element in WSDL, [330](#)
- Message property, of Exception class, [218](#), [225](#)
- message queues, [342–347](#)

- acknowledgments and time-outs, [358–373](#)
 - auditing messages, [373](#)
 - fault tolerance and load balancing, [366–370](#)
 - processing acknowledgment messages, [361–366](#)
 - requesting acknowledgment, [358–361](#)
 - transactional messages, [371–373](#)
- creating, [344–345](#)
- deleting, [345](#)
- processing orders with messages, [373–381](#), [374](#)
 - committing order to database, [380–381](#)
 - deleting order and related message, [381](#)

ModifiedAfter property, of MessageQueueCriteria class, [349](#)
ModifiedBefore property, of MessageQueueCriteria class, [349](#)
modules, [193](#), [205](#)
 creating, [46](#)
month, number of days in, [39](#)
mouse, and DataGrid control, [516](#)
MoveNext method, of MessageEnumerator class, [355](#)
MS Sans Serif, [627](#)
MSDE (Microsoft SQL Server 2000 Desktop Engine), potential problems, [329](#)
MsgBox command, [17](#)
MSIL (Microsoft Intermediate language), [214](#)
MSMQ. *See* Microsoft Message Queueing (MSMQ) component
MSMQLoadBalancing project, [367](#)–[370](#)
 BalancedQueue setup, [368](#)
 enumerating messages, [369](#)
 random message creation, [369](#)
MSScript control, [593](#)
multi-tier architecture, [442](#), [443](#). *See also* middle tier components
 for invoicing application, [518](#)–[519](#)
Multiline member of RegexOptions enumeration, [549](#)
multiplication, of complex numbers, [620](#)–[621](#)
My Computer zone, code executed from, [128](#)

N

Name property
 changing, [24](#)
 of XMLReader object, [477](#)
Namespace property, of XmlElement, [74](#)
namespaces, [18](#), [193](#)
 accessing compatibility, [201](#)–[202](#)
 added automatically as default, [7](#)
 adding as reference, [495](#)
 in XML, [480](#), [487](#)
NaN (not a number), [234](#), [235](#)
 testing for, [235](#)–[236](#)
navigation

- DataSets, [421–426](#)
 - in mobile computing, [646–647](#)
- negative lookahead, [575](#)
- negative lookbehind, [576](#)
- NegativeReceive member of AcknowledgeTypes enumeration, [359](#)
- nesting TextBoxes, [625](#)
- .NET applications, COM component use with, [461–467](#)
- .NET Compact Framework, [643](#)
 - case sensitivity in, [660](#)
 - code-behind programming, [648–649](#)
 - debugging via tracing, [654–656](#)
 - device specificity, [658–660](#)
 - emulators, [660–663](#)
 - friendly error messages, [656–658](#), [658](#)
 - limitations, [644–645](#)
 - List controls, [650–651](#)
 - new features, [647–651](#)
 - security, [652–653](#)
 - simulator, [645–647](#)
 - mobile form, [646](#)
 - navigation to second form, [646–647](#)
- .NET Configuration snap-in, [253](#), [253–257](#)
- .NET Framework, [1](#)
 - class descriptions, [5–6](#)
 - data types, [11–18](#)
 - exploiting, [18–20](#)
 - grammar, [6–11](#)
 - help, [3–5](#)
 - installing runtime, [244–245](#)
 - security features, [121–122](#)
 - WinCV (Windows Class Viewer), [20–21](#)
 - XML classes in, [475](#)
- .NET Framework Samples Database, installing, [327](#)
- .NET Security Policy management, [128–133](#)
- NETConfigFiles project, [76–79](#), [78](#)
- Netscape products, [273](#), [658](#)
- New keyword, [7](#)
 - need for, [8](#)
- New Project dialog box, [645](#)
- NewGuid method, of Guid class, [419](#)

newline, metacharacter for, [560](#)

Next method, of system.random object, [51–52](#)

NextDouble method, of system.random object, [51–52](#)

NextMatch method, of RegEx class, [551–552](#)

NextMatch property, of Match class, [550](#)

no-touch deployment. *See* Internet-based deployment process

[Team Fly](#)

 Previous

Next 

nodes in XML document

adding to XML document, [489–490](#)

recursive walk through, [491–492](#), [493](#)

NodeType property, of XMLReader object, [478](#)

Nokia, emulators, [661](#)

non-greedy regular expressions, [561](#)

None member

of AcknowledgeTypes enumeration, [359](#)

of RegexOptions enumeration, [549](#)

Northwind database, establishing connection, [403](#)

NoSupportedException, of FileStream.Read method, [221](#)

NotAcknowledgeReachQueue member of

AcknowledgeTypes enumeration, [359](#)

NotAcknowledgeReceive member of AcknowledgeTypes
enumeration, [359](#)

NotFiniteNumberException, [226](#)

Nothing permission set, [251](#)

NoTouchDeployment project, [248–249](#)

NotSupportedException, of Open method, [220](#)

null values

in DataAdapter, [406](#), [418–419](#)

setting database field to, [401](#)

in XML format, [85](#)

numbers

random, [49–53](#)

real and imaginary, [603–604](#)

undefined, [234](#)

testing for, [235–236](#)

NWOrders project, [81](#), [81–90](#)

AddDetailLine stored procedure, [524–525](#)

AddHeader stored procedure, [524](#)

adding business rule, [532–535](#)

code, [525–532](#)

committing order to database, [88–90](#), [530–531](#)

creating and serializing new order, [86–87](#)

deserializing XML into custom class instance, [87–88](#)

deserializing XML representing order, [90](#)

- GetProductById stored procedure, [523](#)
- GetProductsByName stored procedure, [524](#)
- OrderClass class, [519–525](#)
- ReadOrder stored procedure, [89–90](#)
- ReduceRows subroutine, [531–532](#)
- setup project for, [259–260](#)
- user interface, [445](#)
- NWProducts application, [506–516](#)
 - architecture, [508–510](#)
 - code, [510–516](#)
 - code of search form, [514–515](#)
 - concurrency handled by, [515](#)
 - interface, [507](#), [507–508](#)

O

- OAEP, [157](#)
- Object Browser, [6](#)
 - opening, [5](#)
 - translating information in, [8](#), [8](#)
- object pooling, [469](#), [469–470](#)
 - implementing, [470–473](#)
- "Object reference not set to an instance of an object" error message, [8](#), [19](#)
- object type, [16](#)
- ObjectList control, for mobile computing, [651](#)
- objects
 - accessing members, [10](#)
 - instantiation, [45](#)
 - in Microsoft Outlook Library, [109](#)
 - reflection to learn about, [191](#)
 - serialization, [59](#)
- Office applications, [109–110](#). *See also* Excel; Word
- Ole Automation, [39](#)
- OleDbDataAdapter class (ADO.NET), [392](#)
- OleDbPermission code access permission, [252](#)
- one-way functions, [154](#)
- OnError statement, Resume, [224](#)
- OnPaint event, [589](#)
- OOP (object oriented programming), [2](#)
- Opacity property, of forms, [636](#), [638](#)

Open method, exceptions, [219–220](#)
OpenRead method, of WebClient class, [309](#)
Openwave, emulators, [661](#)
OpenWrite method, of WebClient class, [309](#)
optimistic concurrency, [407](#), [516](#)
Option Base statement, [28](#), [29](#)
OR Boolean operator, for DataView object, [426](#)
order processing with messages, [373–381](#), [374](#)
 committing order to database, [380–381](#)
 deleting order and related message, [381](#)
 message retrieval from queue, [378–379](#)
 order preparation, [375–377](#)

origin of graph, relocating, [595–596](#)
origin of page, [168](#)
outgoing queues, [343–344](#)
Outlook, [109–110](#)
Output window for debugging, [239–240](#)
OutputStream property, of Response object (HTML), [286](#)
OverflowException, [226](#)
 in math calculation, [218](#)
overhead, in serializing and deserializing, [79](#)
"Overload resolution failed because no accessible 'New' accepts this number of arguments"
 error message, [9](#)

P

padding algorithm in encryption, [157](#)
page layout for printing, [168–174](#)
 DrawString method, [168–170](#)
 PrintTests project, [170](#), [170–174](#)
Page Setup dialog box, [164–165](#), [165](#)
PageSettings object, [162–163](#)
 Page Setup dialog box to display current settings, [164](#), [165](#)
PageSettings property, of PrintDocument object, [160](#)
PageUnit property, of Graphics object, [168](#)
paging, DataGrid support, [276](#)
PaintPixel() function, [615–616](#)
PaperName property, of PaperSize object, [163](#)
PaperSize property, of PageSettings object, [163](#)
PaperSizes property, of PrinterSettings object, [164](#)
PaperSource property, of PageSettings object, [163](#)
PaperSources property, of PrinterSettings object, [164](#)
paragraphs.Item collection, [106](#)
Parameter object, for SQL commands, [410](#)
parent class, [12](#)
Parse method, [213](#)
parsing, [212](#)
passwords
 hashing, [140–142](#)

- as security problem, [139–140](#)
- PathTooLongException, of Open method, [219](#)
- PDAs. *See also* mobile computing
 - emulators, [660–663](#)
- Peek method, [41](#)
 - of MessageQueue class, [357–358](#)
- PeekByCorrelationID method, [357](#)
- peer-to-peer programming, [289](#)
 - sockets, [292–300](#)
 - UDP (User Datagram Protocol), [295–297](#)
- performance
 - open transactions and, [408](#)
 - verification and, [125](#)
- PerformanceCounterPermission code access permission, [252](#)
- period (.), in regular expressions, [543](#), [545](#), [558](#)
- Perl (Practical Extraction and Report Language), and regular expressions, [587](#)
- "permission denied" error message, from MSDE, [329](#)
- permission sets
 - built-in, [251](#)
 - creating and configuring, [253–257](#)
- permissions
 - demand to test caller level, [135–136](#)
 - .NET settings, [121](#)
- persistence
 - in drawing, [589](#)
 - of object, [60](#). *See also* serialization
- pessimistic concurrency, [408](#)
- PEVerify, [121](#), [121](#)
- PictureBox control
 - axes numbering for tick marks, [599–600](#)
 - drawing grids on, [598–602](#)
 - in PlotControl application, [590](#)
 - titles, [601](#)
- plaintext, [143](#), [320](#)
 - printing, [174–178](#), [175](#)
- PlotControl application, [590](#), [590–602](#)
 - members, [591–598](#)
- PlotTitle property, in PlotControl application, [592](#)
- PlotTitleColor property, in PlotControl application, [592](#)
- PlotTitleFont property, in PlotControl application, [592](#)

Poll method, of socket, [295](#)
pool of objects for reuse, [469](#)–470
 implementing, [470](#)–473
PooledServer project, [470](#)–473
 testing, [472](#)–473
portrait mode, [113](#)
ports, [294](#)
 for UDP connection, [297](#)
portType element in WSDL, [331](#)
PortType section, of WSDL document, [335](#)–336

- enumerating messages, [369](#)
- random message creation, [369](#)
- NETConfigFiles project, [76–79](#), [78](#)
- NoTouchDeployment project, [248–249](#)
- NWOrders project, [81](#), **[81–90](#)**
 - AddDetailLine stored procedure, [524–525](#)
 - AddHeader stored procedure, [524](#)
 - adding business rule, [532–535](#)
 - code, [525–532](#)
 - committing order to database, [88–90](#), [530–531](#)
 - creating and serializing new order, [86–87](#)
 - deserializing XML into custom class instance, [87–88](#)
 - deserializing XML representing order, [90](#)
 - GetProductById stored procedure, [523](#)
 - GetProductsByName stored procedure, [524](#)
 - OrderClass class, [519–525](#)
 - ReadOrder stored procedure, [89–90](#)
 - ReduceRows subroutine, [531–532](#)
 - setup project for, [259–260](#)
 - user interface, [445](#)
- NWProducts application, [506–516](#)
 - architecture, [508–510](#)
 - code, [510–516](#)
 - code of search form, [514–515](#)
 - concurrency handled by, [515](#)
 - interface, [507](#), [507–508](#)
- PlotControl application, [590](#), **[590–602](#)**
 - members, [591–598](#)
- PooledServer project, [470–473](#)
 - testing, [472–473](#)
- PrintTests project, [170](#), [170–174](#)
- ProcessOrders console application, [385–388](#)
- ReadWriteFile project, [219–224](#)
- RegExEditor project, [564–567](#)
 - Find & Replace dialog box, [564–567](#)
- RegularExpressions project, [544](#), [545](#), **[579–582](#)**

- Relations application, [535](#), [535–539](#)
 - architecture, [535–536](#)
 - code, [536–539](#)
- Relations1 project, [539–542](#)
- RemoteOrders application, [460](#)
- SimpleQueue project, [362–363](#)
 - processing acknowledgment messages, [365–366](#)
- TcpChat application, [300](#), [300–307](#)
 - TcpChatClient application, [305–307](#)
 - MessageArrived event, [307](#)
 - TcpChatServer application, [301–305](#)
 - ChatClass, [302–303](#)
 - listening for requests on separate thread, [303](#)
- TCPServer project, [297–298](#)
- Transaction project, [430](#), [431](#), [432–436](#)
- UDPClient application, [296–297](#)
- UDPServer application, [295](#), [295–296](#)
- Visual grep project, [582](#), [582–587](#)
- properties
 - of forms, [45](#)
 - public, reflection to report on, [197](#)
 - serialization for saving, [60](#)
 - of Setup Project, [265–266](#)
- proxy
 - exporting and testing, [467–468](#)
 - between managed and unmanaged code, [462](#)
- proxy server, [290](#)
- public key encryption systems, [139–140](#), [151](#), [153](#)
 - code for encryption and decryption, [154–156](#)
 - managing keys, [158](#)
- public properties, reflection to report on, [197](#)
- public queues, [343](#)
 - referencing, [345](#)
- public variables, to reference form, [45](#)
- Publisher membership condition, [256](#)
- Pubs sample database, connection to, [327](#)
- punctuation symbols, printing, [178](#)
- purging message queues, [345](#)

Q

quantifiers, in regular expressions, [546](#), [560–562](#)
Query Analyzer window, [82](#)
Query Builder, [394](#)
query, to retrieve order information in XML format, [82–83](#)
question mark (?), as metacharacter, [560](#)
queued components, [469](#)
queues. *See* message queues
Quick Watch window for debugger, [238](#)

R

random generator seeding, [50–52](#)

- Relations application, [535](#), [535–539](#)
 - architecture, [535–536](#)
 - code, [536–539](#)
- Relations property, of DataSet, [421](#)
- Relations1 project, [539–542](#)
- relationships between tables, [397–398](#)
- Release mode, [232](#)
- ReleaseDC function, [187](#)
- remote systems
 - accessing, [289](#)
 - invoking components on, [449](#)
- RemoteOrders application, [460](#)
- remoting BusinessLayer class, [458–461](#)
- Remove method, of Rows collection of DataTable, [419](#)
- RemoveAt method, [35–36](#)
 - of Rows collection of DataTable, [419](#)
- RemovePreviousVersion property, of Setup Project, [266](#)
- RenderingOrigin property, of Graphics object, [168](#)
- repeated words, removing, [571](#)
- Repeater control, [276](#)
- Replace method, of RegEx class, [552–557](#)
 - MatchEvaluator project, [554–557](#)
- replacement
 - with regular expressions, [553](#)
 - grouped matches, [569–573](#)
 - of text, [106–107](#)
- replication of public queues, [343](#)
- RequestLimit attribute, in Web.config file, [655](#)
- RequiredFieldValidator control, [283](#)
- resolution of printer, [163](#)
- Resolve method, of IPHostEntry class, [292](#)
- Response object (HTML)
 - arguments for, entire files as, [272](#)
 - OutputStream property, [286](#)
 - Write command, [273](#)
- restricted sites zone, [128](#)
- RetrieveByCorrelationID method, [357](#)

- reverse engineering, [134](#)
- Reverse method, [34](#)
- rich client, [441–442](#)
 - vs. Web applications, [457](#)
- RichTextBox control for Visual grep project, [585](#)
 - formatted text in, [587](#)
- RightToLeft member of RegexOptions enumeration, [549](#)
- Rivest, Shamir, and Adleman. *See* RSA (Rivest, Shamir, and Adleman) encryption system
- Rnd function, [49](#)
- role-based security, [120](#), [122](#), [469](#)
- Rollback method, of Transaction object, [437](#)
- root element, changing name, [76](#)
- Row property, of DataTable object, [425](#)
- rows in database, [417–418](#)
 - adding and deleting in DataTable, [419–420](#)
 - deleting, [405](#)
 - limiting selection, [405](#)
 - search for, [420–421](#)
- RowStateFilter property, for DataView object, [427](#)
- RSA (Rivest, Shamir, and Adleman) encryption system, [151–158](#)
 - how it works, [153–156](#)
- RSACryptoServiceProvider object, [156](#)
- rules
 - grammar, [6–11](#)
 - for message queue triggers, [382–384](#), [383](#)
- runtime
 - adding controls to form at, [25–27](#)
 - DataSet generated at, [415](#)
 - evaluating functions at, [593–595](#)
- runtime-callable wrapper (RCW), [462](#)
- runtime errors, [232](#)
 - "Failed to enable constraints...", [398–399](#)
 - when opening, reading and writing files, [219–224](#)
 - Open method exception handling, [219–221](#)
- runtime handles, [47](#)
- Runtime.Serialization.SerializationException, [227](#)

S

- sa account, [403](#)

sans serif font, [625](#)
for headlines, [627](#)
Save method, of XMLConfiguration class, [76–77](#)
SaveFileDialog control, [43–44](#)
SAX (Simple API for XML), [475](#)
choosing, [476–478](#)
Scale property, of Parameter object, [410](#)
schemas in XML, [329](#), [478–479](#), [481–486](#)
data types, [483–486](#)
extending, [484–486](#)

- security, [120](#)
- serialization of data, [81–90](#)
- SqlClientPermission code access permission, [252](#)
- SqlDataAdapter class (ADO.NET), [392](#)
- SqlDbType property, of Parameter object, [410](#)
- StackOverflowException, [228](#)
- StackTrace property, of exception objects, [225](#), [226](#)
- Start method, of TCPListener class, [302](#)
- start of line, metacharacter for, [562](#)
- StartChat method, [301](#), [304](#)
- StartListening method, [301](#)
- startup object, default, in Visual Basic, [4](#)
- state, preserving in Web services, [325–326](#)
- State property, of rows in DataSet, [424](#)
- statelessness, [325](#)
- static methods, [15](#)
- Status property, of MessageQueueTransaction class, [371–372](#)
- step into when debugging, [239](#)
- step over when debugging, [239](#)
- stored procedures
 - CommandText property to store, [409](#)
 - executing, [411–412](#)
 - for item discount, [534](#)
 - in NWProducts application, [508](#)
- Stream object, ReadLine method of, [311](#)
- streaming, [1](#), [39–44](#)
 - mixing data types in same, [55–56](#)
 - to write to file, [43](#)
- StreamingContextState enumeration, [80](#)
- StreamReader object, ReadToEnd method of, [42](#)
- Strict option for module, [232–233](#)
- String class, IndexOf method of, [543](#)
- StringFormat object, [169](#), [172](#)
- strings, [14–15](#)
 - converting byte arrays to, [308](#)
 - converting to byte arrays, [311](#)
 - DrawString method for printing, [168–169](#)

- empty, in database, [401](#)
- Excel to evaluate math expressions as, [111](#)–[112](#)
- listing methods for, [19](#)
- Word for formatting, [102](#)–[104](#)
- strong name, creating, [470](#)
- Strong Name membership condition, [256](#)
- strong typing, [17](#)–[18](#), [37](#)
- structure value type, [14](#)
- structured exception handling, [63](#), [216](#)–[231](#)
 - bypassing error handlers, [230](#)–[231](#)
 - error prevention, [225](#)
 - Exception class, [225](#)–[228](#)
 - Finally clause, [219](#)
 - ReadWriteFile project, [219](#)–[224](#)
 - resuming failed statements, [224](#)–[225](#)
 - sections of code, [217](#)
 - throwing custom exceptions, [228](#)–[230](#)
- Sub. *See* constructors
- Sub Main, [4](#)
- subdirectories, process to get list, [10](#)
- subtraction of complex numbers, [620](#)
- Success property, of Match class, [550](#)
- SupportPhone property, of Setup Project, [266](#)
- SupportsColor property, of PrinterSettings object, [164](#)
- SupportUrl property, of Setup Project, [266](#)
- suppressing messages, [100](#)
- symmetric encryption routine, [142](#)
- syntax errors, [232](#), [233](#)
- System namespace, [4](#), [7](#)
- System.Data namespace, [7](#)
- System.Drawing namespace, [7](#), [18](#)
- System.IO namespace, [11](#)
- System.Messaging.MessageQueue class, [345](#)
- System.Net namespace, [289](#)
- System.Net.Dns namespace, [289](#), [291](#)
- System.Object class, [12](#)
- System.Random object, [50](#)
- System.Runtime.Serialization class, [61](#)
- System.Runtime.Serialization.Formatter.S Soap, as reference, [495](#)
- System.Security.Principal namespace, [122](#)

System.Text.Encoding class, [308](#)

System.Text.Encoding.ASCII class, [311](#)

System.Type class, [192](#)

System.Windows.Forms namespace, [7](#)

System.XML namespace, [7](#)

System.Xml.Serialization namespace, [72](#)

T

tab, metacharacter for, [560](#)

tables in database. *See also* DataTable objects

multiple, for DataSet, [396–400](#)

- TableStyles property, of DataGrid control, [401](#)
- tabular data, printing, [179–186](#)
- target computer, setup project impact on file system, [262](#)
- TargetSite property, of exception objects, [225](#)
- TCP (Transmission Control Protocol)
 - classes to exchange data, [307](#)
 - server, sending message to, [299–300](#)
 - sockets, [292](#), [294](#), [297–300](#)
- TCPChat application, [300](#), [300–307](#)
- TcpChatClient application, [305–307](#)
 - MessageArrived event, [307](#)
- TcpChatServer application, [301–305](#)
 - ChatClass, [302–303](#)
 - incoming messages, [305](#)
 - listening for requests on separate thread, [303](#)
- TCPServer project, [297–298](#)
- templates, for Webform controls, [275](#)
- testing
 - for infinity, [235–236](#)
 - proxy, [467–468](#)
 - regular expressions, [571](#)
 - transactional updates, [439–440](#)
 - for undefined numbers, [235–236](#)
 - Web services, [321](#), [337–339](#)
- Text property, of control, binding, [506](#)
- TextBox controls
 - exporting content to Clipboard, [23](#)
 - loading Word document into, [99–100](#)
 - nesting, [625](#)
 - Print method, [174–178](#), [175](#)
 - and printing, [159](#)
 - spell-checking contents, [94](#)
- TextView control, in .NET Compact Framework, [647](#)
- threads
 - background on client chat application, [305–306](#)
 - for chat programs, [301](#)

- Throw method, [229](#)
- Ticks, [38](#)
- tiers, vs. layers, [443](#)
- time, calculating elapsed, [38](#)
- Time function, [38](#)
- TimeOfDay function, [38](#)
- timeout, for queued message acknowledgment, [359](#)
- TimeSpan object, for Receive and BeginReceive methods, [356](#)
- TimeToBeReceived property, of Message class, [351](#), [360](#)
- TimeToReachQueue property, of Message class, [351](#)
- tlbimp.exe tool (Type Library Importer), [462](#)
- Today function, [38](#)
- ToDouble method, of DateTime class, [38](#)
- ToPage property, of PrinterSettings object, [163](#)
- ToString method, [12](#), [16](#), [112](#)
- trace, to debug mobile application, [654–656](#)
- Transaction object, [436](#)
- Transaction project, [430](#), [431](#), [432–436](#)
- transactional messages, [343](#), [371–373](#)
- transactions
 - DataAdapter class for, [432–440](#)
 - implementing optimistic concurrency with, [408](#)
 - testing updates, [439–440](#)
- transformation matrix, in GDI+, [596](#)
- Translate method, of world coordinate system, [595–596](#)
- Transmission Control Protocol (TCP) sockets, [292](#), [294](#)
- trap doors, [153–154](#)
- triggers for message queues, [382–388](#)
 - defining, [384–385](#)
 - ProcessOrders console application, [385–388](#)
 - rules, [382–384](#), [383](#)
- Trimming property, of StringFormat object, [178](#)
- TrimToSize method, of ArrayList, [36](#)
- TripleDES, [143](#)
- trusted sites zone, [128](#)
- Try statement, [217](#)
- type, [56](#)
- type size, user interface convention for, [627](#)
- typed DataSets, [415](#)
 - table names as properties, [416](#)

types

binary serialization and, [65](#)

direct contact with specific, [206–207](#)

and reflection, [191–192](#), [205–206](#)

Types element in WSDL, [331](#)

TypeText method, [106](#)

U

UBound function, [29](#)

UDDI Business Registry (UBR), [337](#)

UDDI (Universal Description, Discovery, and Integration), [336–337](#)

- UDP (User Datagram Protocol) sockets, [292](#), [294](#), [295–297](#)
- UDPCClient application, [296–297](#)
- UDPServer application, [295](#), [295–296](#)
- UIPermission code access permission, [252](#)
- UnauthorizedAccessException, of Open method, [220](#), [222](#)
- "An unhandled exception of type
 'System.Runtime.InteropServices.COMException' ..."
 error message, [95](#)
- unhandled exceptions, [216](#)
- Universal Description, Discovery, and Integration (UDDI), [336–337](#)
- Unix, grep utility, [582](#)
- UnknownAttribute event, when deserializing XML stream, [501–502](#)
- UnknownElement event, when deserializing XML stream, [501–502](#)
- unmanaged code, [462](#)
- Update method, of DataAdapter, [400](#), [404](#), [425](#), [428–430](#)
- UPDATE statement (SQL)
 - from DataAdapter configuration wizard, [406](#)
 - DataAdapter task for, [404](#)
- UpdateCommand property, of DataAdapter, [404](#)
- UpdateRule property, [431](#)
- upgrading applications, [244](#)
 - Internet-based deployment and, [246](#)
- UploadData method, of WebClient class, [309–310](#)
- UploadFile method, of WebClient class, [310](#)
- uploading documents, with WebClient, [312–313](#)
- UploadValues method, of WebClient class, [310](#)
- URL membership condition, [256](#)
- URLs (Uniform Resource Locators), for namespaces, [480](#)
- UseDeadLetter Queue property, of Message class, [351](#)
- User Datagram Protocol (UDP) sockets, [292](#), [294](#), [295–297](#)
- user interface
 - DataGrid control and, [396](#), [399](#)
 - fading transitions, [637](#), [637–640](#)
 - focus group for guidelines, [626](#)
 - metallic shading, [635–636](#), [636](#)
 - reliability of applications, [623–626](#), [624](#)
 - slide transitions, [640](#), [640–641](#)

- Windows conventions, [626–634](#)
 - depth, [627–628](#)
 - FontBold off, [627](#)
 - framing, [630–634](#)
 - layering, [627](#)
 - light from upper left, [628–629](#)
 - metallic look, [626–627](#)
 - sans serif font for headlines, [627](#)
 - type size, [627](#)
 - zones, [629](#)

- User Interface Editor, [262](#), [267–270](#)
- user level for security policy, [251](#)
- users, [122](#)
- User's Programs Menu, adding application to, [267](#)

V

- validation, [282–285](#)
 - in ASP.NET, [282–285](#)
 - controls, [283–285](#)
 - programmatic, [282–283](#)
 - controls, [283–285](#)
 - of data, [225](#)
 - programmatic, [282–283](#)
- ValidationSummary control, error messages, [284](#)
- Value property
 - of Match class, [550](#)
 - of Parameter object, [410](#)
 - of XMLReader object, [478](#)
- value types, [14](#)
- ValueMember property, for binding controls, [509](#)
- variables
 - displaying, [17](#)
 - forcing declaration, [232–233](#)
 - initialization, [30](#)
 - public, to reference form, [45](#)
 - scope of, [239](#)
- Variants, [17](#)
- velocity of escape, and fractal color, [614–615](#)
- verification

- of method argument lists, [121](#)
- and performance, [125](#)
- Version property, of Setup Project, [266](#)
- vertical alignment of string, [169](#)
- views, for table editing, [426](#)
- Visual Basic
 - beginnings, [1](#)
 - changes for .NET, [1-2](#)

- inflation, [192](#)
- terminology changes for VB.NET, [21–22](#)
- Word macros to view, [103–104](#)
- Visual Basic .NET, translating C# to, [21](#)
- Visual grep project, [582](#), [582–587](#)
- Visual Studio .NET
 - to create connection strings, [403](#)
 - creating Windows installer package in, [259](#), [261–262](#)
 - DataAdapter configuration, [393–394](#)
 - database connection, [392–393](#)
 - DataSet creation, [395–396](#)
 - multiple tables, [396–400](#)
 - Pocket PC emulator, [661](#), [661–662](#)
 - problems, [662–663](#)
 - updating database, [400–402](#)
 - viewing DataSet, [396](#)
- void, [15](#)

W

- WAP (Wireless Application Protocol), [645](#)
- Watch window for debugger, [238](#)
- Web applications, [246](#)
 - for database client, [442](#)
 - vs. rich client applications, [457](#)
- web page, for connecting to application, [258](#), [258](#)
- web resources, [308–317](#)
- web server
 - downloading document from, [315](#)
 - user download of application from, [246](#)
 - Windows application deployment on, [249–250](#)
- Web services, [2](#)
 - adding to project, [336](#)
 - caching data, [322–323](#)
 - characteristics, [319–320](#)
 - consuming, [323–325](#), [324](#)
 - converting BusinessLayer class to, [450–457](#)

- creating, [320–323](#)
- first line, [322](#)
- making database connection, [326–330](#)
- middle tier component as, [247](#)
- preserving state, [325–326](#)
- referencing, [456–457](#)
- security, [339](#)
- testing, [321](#), [337–339](#)
- UDDI (Universal Description, Discovery, and Integration), [336–337](#)
- XML Dataset, [327–328](#), [328](#)
- Web Services Description Language (WSDL), [330–336](#)
 - complex types, [333–335](#)
 - Enum translated into SOAP and WSDL, [334–335](#)
 - PortType section, [335–336](#)
 - Reference Map, [336](#)
 - SOAP, [333](#)
 - viewing, [331–332](#)
- Web Setup Project, as New project option, [261](#)
- web sites, C# to VB.NET translator, [21](#)
- WebClient class, [308](#)
 - to download documents, [311–312](#)
- Web.config file
 - authorization section for mobile computing, [653](#)
 - changing for remoting, [459–460](#)
 - for custom error messages, [656–657](#)
 - device specificity, [659](#)
 - RequestLimit attribute in, [655](#)
 - to set trace, [654](#)
- WebForm data display, [273–283](#)
 - connecting to database, [273–274](#)
 - DataGrid control, [276–281](#)
 - DataList control, [275](#)
 - detecting postback, [281–282](#)
 - Repeater control, [276](#)
 - templates, [275](#)
- <WebMethod> attribute, [450](#)
- WebPermission code access permission, [252](#)
- WebRequest object, [314–315](#)
- WebResponse object, [315–317](#)
- WebResponse stream, [61](#)

white space, in regular expressions, [560](#)

WinCV (Windows Class Viewer), [20–21](#)

Windows

- application deployment on Web server, [249–250](#)

- security, [120](#)

- user authentication, [403](#)

- user interface conventions, [626–634](#)

 - depth, [627–628](#)

 - FontBold off, [627](#)

 - framing, [630–634](#)

 - layering, [627](#)

 - light from upper left, [628–629](#)

- sans serif font for headlines, [627](#)
 - type size, [627](#)
 - zones, [629](#)
- Windows 98, installing .NET Framework on, [244](#)
- Windows Class Viewer. *See* WinCV (Windows Class Viewer)
- Windows controls
 - binding to columns, [506](#)
 - data to populate, [412](#)
- Windows Explorer, security settings, [122](#)
- Windows form, related data on, [535–539](#)
- Windows installer
 - creating in Visual Studio .NET, [259](#), [261–262](#)
 - deployment process with, [259–270](#), [260](#)
 - File System Editor, [263–266](#)
 - installer package creation, [261–262](#)
 - Registry Editor, [267](#)
 - shortcut creation, [266–267](#)
 - User Interface Editor, [267–270](#)
- Windows Server 2003, Software Restriction Policies, [122](#)
- Windows XP
 - default security settings, [128–129](#)
 - Graphics Device Interface (GDI+), [589](#)
 - Message Queuing Triggering service, [382](#)
 - Software Restriction Policies, [122](#)
- WindowsIdentity class, [122–123](#)
- WindowsPrincipal class, [122–123](#)
- Wireless Application Protocol (WAP), [645](#)
- WithEvents, in declaration, [28](#)
- Word
 - fax sending, [98](#)
 - feeding individual strings and specialized formatting, [102–104](#)
 - finding files, [100–102](#)
 - IntelliSense list, [105](#)
 - loading documents, [99–100](#)
 - printing features, [107–108](#)
 - replacing text, [106–107](#)

- sending text to VB.NET from, [106](#)
- spell-check, [94–98](#)
 - passing text directly, [96](#)
 - retrieving misspelled word list, [96–98](#)
 - for VB.NET TextBox, [94–95](#)
- text manipulation and insertion, [104–106](#)
- word character, metacharacter for, [544](#), [546](#), [558–559](#)
- word count, [100](#)
- Word object model, [104](#)
- WordWrap property, [175](#), [178](#), [178](#)
- WorkingFolder property, for application shortcut, [266](#)
- worksheet object in Excel, [115](#)
- workstations
 - access by multiple to same queue, [370](#)
 - authentication for database connection, [402–403](#)
- world coordinate system, Translate method of, [595–596](#)
- wrapper, [52](#)
 - COM, [94](#)
- Write command, of Response object, [273](#)
- WriteLine method, of Debug class, [240](#)
- writing files, [42–44](#)
- WSDL (Web Services Description Language), [330–336](#)

X

- XAxisTitle property, in PlotControl application, [592](#)
- XCopy method, [244](#)
- XMax property, in PlotControl application, [592](#)
- XMin property, in PlotControl application, [592](#)
- XML. *See also* SAX (Simple API for XML)
 - for application configuration files, [76](#)
 - classes in .NET Framework, [475](#)
 - controlling output, [73–76](#)
 - converting DataSet to, [504](#), [504](#)
 - converting to DataSet, [503–504](#), [504](#)
 - database results as, [329–330](#)
 - and DataSets, [493–495](#)
 - DOM (Document Object Model) and, [475](#), [478–479](#)
 - interchangeability, [503–504](#)
 - namespaces in, [480](#)

- persistence with SOAP, [495–503](#)
- persisting instance of AppConfig class in, [77](#)
- programmatic, [487–492](#)
 - edit and save, [488–490](#)
 - recursive walk through nodes, [491–492](#)
- query to retrieve order information in, [82–83](#)
- restoring instance of AppConfig class from, [77](#)
- SAX (Simple API for XML), [476–478](#)
- schemas, [481–486](#)
 - data types, [483–486](#)
 - XSD, [481–482](#)

- specifications, [332](#)
- for Web services, [320](#)
- WSDL descriptions in, [330](#)
- XML Dataset, [327–328](#), [328](#)
- XML document. *See also* elements in XML
 - adding node, [489–490](#)
 - loading literal string into, [487](#)
- XML (SOAP) serialization, [55](#), [60–61](#), [72–76](#), [495–503](#)
 - deserialization trapping, [501–503](#)
 - mixing and matching types, [497–501](#)
 - reading mixed data, [499–501](#)
 - queued message in, [353–354](#)
- XmlAnyAttribute property, [73–74](#)
- XmlAnyElements attribute, [74](#)
- XmlArray attribute, [74](#)
- XmlArrayItems attribute, [74](#)
- XmlAttribute attribute, [74](#), [75](#)
- XmlAttributes class, [73](#)
- XmlChoiceIdentifier attribute, [74](#)
- XMLConfiguration class, Save method, [76–77](#)
- XmlDefaultValue attribute, [74](#)
- XmlElement attribute, [74](#)
- XmlAttribute attribute, [74](#), [75](#)
- XmlIgnore attribute, [74](#)
- xmlns attribute, [480](#)
- XMLReader object, [477](#)
- XmlRoot attribute, [74](#)
- XmlSerializer class, [61](#), [72](#), [500–501](#)
 - deserializing XML stream, [501–502](#)
- XmlText attribute, [74](#)
- XmlType attribute, [74](#)
- XPathNavigator API, [477](#)
- XSD, [481–482](#)
 - command-line tool, [83–86](#)

Y

YAxisTitle property, in PlotControl application, [592](#)
YoSpace, [660](#)

Z

zero, division by, [234](#)
Zone membership condition, [256](#)
zones, as user interface convention, [629](#)
zoom operation, on fractals, [616](#)

Figure 1.3 illustrates how you must look in two locations in the Object Browser to find details about the process of getting a list of subdirectories:

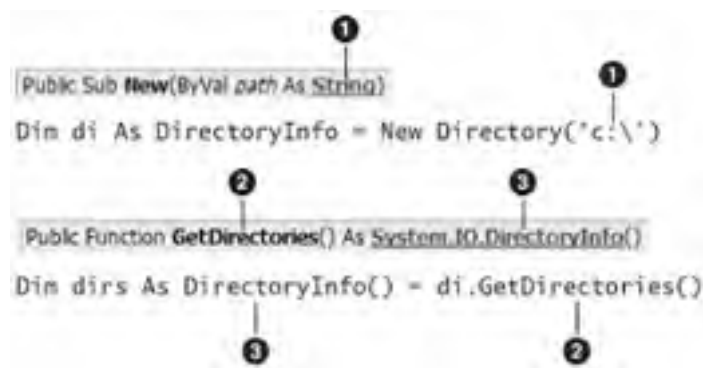


FIGURE 1.3 You must look at both the `New` and `GetDirectories` methods in the Object Browser to figure out how to employ this technique.

And even after you figure out how to use `New` to instantiate a `DirectoryInfo` object, and how to use `GetDirectories`, you still don't have enough information to know how to make these two objects (the `DirectoryInfo` object and the separate array of `DirectoryInfo` objects) work together. Thank goodness there's sample code illustrating how to get directories. By the way, to see the results, you can reuse the `DirectoryInfo` object `di` to iterate through the array:

```
For Each di In dirs
    Console.WriteLine(di)
Next
```

Why Two Ways?

Why, then, these two different ways of accessing objects' members? One requiring `New` (instantiating the object), the other not requiring instantiation?

```
Dim fstream As FileStream = File.Create("c:\myfilex.txt")
Dim dirI As DirectoryInfo = New DirectoryInfo("c:\")
```

Some objects, such as the `ArrayList`, can be instantiated *either* way:

```
Dim ara As ArrayList

or

Dim ara As New ArrayList
```

In VB6 and earlier versions, the `As New` command meant something different than it does now in .NET. In older versions of VB, when you declared an object variable using `As New`, it was "autoinstantancing"—meaning that it isn't instantiated until used later in the code somewhere. In .NET the object is instantiated as soon as the line with the `Dim` statement executes. No delayed instantiation.

About Constructors

A constructor is a Sub (a method) that executes when a new instance of its class is instantiated, hence the name of a constructor method is always New. In other words, when you instantiate a class,

[Team Fly](#)

 Previous

Next 

In this example, you use the `RecentFiles` collection of the application object to load the most recent file, then copy it to the `TextBox` via the Clipboard. In addition, you get the `Name` property of this file and display it in the form's title bar.

NOTE The Documents collection within the Word application is 1-based, so there is no `documents(0)`.

GETTING A WORD COUNT

Writers need to know many words they've written—a typical computer book page has about 175 words, so you know the size of your chapters, and the entire book, if you count words. However, you cannot use the Word Count feature in Word for text copied and pasted from a VB.NET `TextBox`—you always get too many words in the count (because of CRLF formatting, I suspect). To make this work, you'd have to strip off the formatting codes before copying the text to the Clipboard. If you're going to this much trouble, just count the words by counting the space characters in your `TextBox.Text`.

Here's code you can use, however, if you want to count words in a .doc file (not `TextBox`copied text):

```
Dim w As Object = New Word.Application
Dim d As Object = w.Documents.Add
MsgBox(''Wordcount: ' & d.Words.Count)
```

SUPPRESSING MESSAGES

If, however, this most-recent file is currently open in Word, you'll get a dialog box asking if you want to see a read-only version. You cannot suppress this dialog because it's a fundamental security alert, but most messages—modal or not—can be suppressed by setting the `WdAlertLevel` to `wdAlertsNone`, like this:

```
Dim w As Object = New Word.Application
w.DisplayAlerts = Word.WdAlertLevel.wdAlertsNone
```

The other possible settings for this property are `wdAlertsAll` (the default) and `wdAlertsMessageBox`, which displays only message boxes, not other types of alerts.

Finding Files

Here's a way to locate all files in a particular path, including subdirectories. This would be useful if you wanted to activate Word after selecting a particular .doc file to work on. If you've ever tried to code this kind of thing yourself, you probably realize that recursion is your most efficient approach, and recursion is not for the faint of heart. The technique illustrated here in Listing 4.6 works just as well, and is quite a bit easier on the programmer.

LISTING 4.6: LOCATING FILES

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

    Dim w As Object = New Word.Application
    w.visible = False

    Dim id As IDataObject

    Dim s, t As String
    Dim cr As String = ControlChars.CrLf

    Try
        With w.FileSearch
            .FileName = '*.doc'
            .LookIn = "C:\Book VB Power Toolkit"
            .SearchSubFolders = True
            .Execute()
            For Each s In .FoundFiles
                t &= s & cr
            Next
        End With

        Catch ex As Exception
            MsgBox(ex.ToString)
        End Try

        TextBox1.Text = t

    End Sub
```

Here you use the FileSearch method, filtering with the .doc extension and searching subfolders. The FileSearch returns a FoundFiles collection (of strings) through which you can iterate to build the list of filenames you display in the TextBox. You could easily replace the TextBox with a ListBox, allowing the user of your VB.NET application to choose which file to edit.

Listing 4.7 gives the changes (shown in bold) to use an ArrayList to fill a ListBox with all the .doc files.

LISTING 4.7: FILLING A LISTBOX WITH FILENAMES

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

    Dim w As Object = New Word.Application
```



```
w.visible = False

Dim s As String
Dim t As New ArrayList

Try
    With w.FileSearch
        .FileName = '*.doc'
        .LookIn = "C:\Book VB Power Toolkit"
        .SearchSubFolders = True
        .Execute()
        For Each s In .FoundFiles
            t.Add(s)
        Next
    End With

Catch ex As Exception
    MsgBox(ex.ToString)
End Try

ListBox1.Items.AddRange(t.ToArray)

End Sub
```

Feeding Individual Strings and Specialized Formatting

Although VB.NET includes a useful RichTextBox control, it's not nearly as capable as Word when you want to adjust formatting, employ templates, adjust typeface colors, and do other specialized kinds of text manipulation. Rather than reinvent the wheel when you need to manage specific aspects of your text, go ahead and dump it into a Word document, then you're free to save it as a .doc file with most imaginable kinds of formatting.

In this example (Listing 4.8), you feed some separate strings into Word from VB.NET, adjusting their formatting on-the-fly. This technique could also be a way of parsing and formatting the text in a TextBox, or you could provide a set of several TextBoxes—some for headlines, some for body text, and so on. In this way, the user could specify aspects of the formatting themselves (along with RadioButtons for whatever options you wanted to allow them to specify: color, font size, italic, and so on).

Combine this technique with the printing code (demonstrated in the next section) and you've got a pretty powerful adjunct amplification to VB.NET's intrinsic word processing capabilities.

LISTING 4.8: FORMATTING TEXT

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

    Dim w As Object = New Word.Application
```

```
w.visible = False
Dim d As Object = w.Documents.Add

With w.Selection
    .Font.Italic = True
    .Font.Size = '11" 'specify absolute size
    .font.name = "Arial"
    .TypeText("This is italic, size 11.")
    .TypeParagraph() 'carriage return

    .Font.Size = "24"
    .font.name = "Times New Roman"
    .Font.Italic = False
    .Font.Color = Word.WdColor.wdColorBlue
    .TypeText("This is quite large (24 pt. absolute), Roman, and
    .TypeParagraph()

    .Font.Color = Word.WdColor.wdColorBlack
    .Font.Underline = True
    .Font.Size = w.Selection.Font.Size - 10
    .TypeText("Now black, underlined, and decrease the point size
End With

Try
    d.SaveAs("c:\test.doc")
Catch ex As Exception
    MsgBox("Failed to save document: " & ex.Message)
End Try

w.Quit()

End Sub
```

This is entirely programming, no VB.NET controls involved. Notice that you can specify various aspects of the text formatting—color, font name, font size, underlining, and so on. In each case, you're employing the Selection object of the Word application object (the selection here is the entire document, but you can specify a selection of paragraphs, words, or individual characters if you wish). The TypeText method is used to send a string into the document and the TypeParagraph method simulates pressing the Enter key to move to a new paragraph.

If you're unsure how to write the code to invoke a style, template, or formatting command in Word's VBA language, just record a new macro and apply the styles you're interested in, using the menus and toolbars. Then choose Tools ➤ Macros ➤ Edit to see what the VBA code is. Translating VBA into VB.NET source code isn't terribly difficult. Fiddle around until it works.

Here's an example of a Word macro employing italics, a headline style, and a color applied as Word recorded the keystrokes. You can ignore the MoveLeft and other positioning commands; just look for the VBA formatting. Replace the := symbols with =, and make minor adjustments such as changing wdToggle commands to assignments (such as .Font.Italic = True), or necessary qualifications such as prepending Word.WdColor. to color specifications.

```
Sub Macro6()  
'  
' Macro6 Macro  
' Macro recorded 7/10/2003 by Richard Mansfield  
'  
    Selection.TypeText Text:='This is my text.'  
    Selection.MoveLeft Unit:=wdCharacter, Count:=4, Extend:=wdExtend  
    Selection.Font.Italic = wdToggle  
Extend:=wdExtend  
    ActiveDocument.Styles.Add Name:="Heading 3 Char" , Type:= _  
        wdStyleTypeCharacter  
    ActiveDocument.Styles("Heading 3 Char").LinkStyle = "Heading 3"  
    Selection.Style = ActiveDocument.Styles("Heading 3 Char")  
    Selection.MoveRight Unit:=wdCharacter, Count:=2  
    Selection.Font.Color = wdColorAqua  
End Sub
```

THE WORD OBJECT MODEL

If you've written or edited macros, you've had experience with the Word object model and with VBA, the version of Visual Basic designed for use with application macros. Working with this object model gives you insights into how to consume features in other Office and other Microsoft applications, such as those found in Works. At the top of the Word object hierarchy is the application object. And beneath that is the documents collection, just as if you started Word running, then opened one or more documents underneath the application.

Text Manipulation and Insertion

The Word InsertAfter method works with a selection object (or range object) in Word (see Listing 4.9). It appends text, as you might guess, and it extends the selection as well. The selection object contains a contiguous block of text, but there can be multiple range objects, specifying blocks of text here and there throughout the document. Adjusting the range doesn't affect any selection that's in effect. Other than this distinction, the range and selection objects expose much the same functionality and behave in much the same ways. There's also an InsertBefore method to prepend text.

It's useful to specify ranges or selections primarily because other objects, such as the paragraph or word, do not expose all the functionality of the range and selection objects. You can specify a range by providing a paragraph number within the document (it's not possible to adjust the text of a paragraph object to make it bold, for example—you must first identify a range).

LISTING 4.9: SPECIFYING RANGES

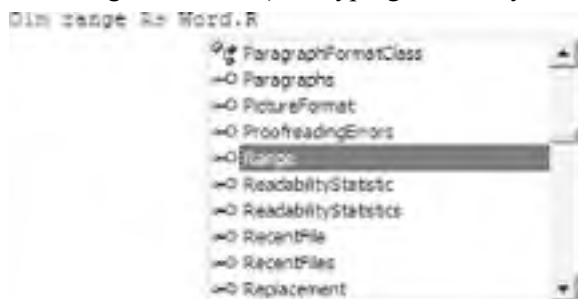
```
Private Sub Form1_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles MyBase.Load  
  
    Dim w As Object = New Word.Application  
    w.visible = True  
  
    Dim d As Object = w.Documents.Add  
  
    With w.Selection  
        .TypeText("'This is the first paragraph.")  
        .TypeParagraph()  
        .TypeText("This is the second.")  
    End With  
  
    Dim range As Word.Range = d.paragraphs(2).range  
    range.Font.Bold = True  
  
  
    w.quit()  
  
End Sub
```

USING THE INTELLISENSE LIST

When working with Word's VBA language in the VB.NET IDE, you'll likely notice that you don't have the advantage of statement completion or other useful IntelliSense features. However, if you precede a method name, for example with *Word.*, you'll then see an IntelliSense list of the possible members for the Word object. Here's an example that brings up the IntelliSense list:

```
Dim range As Word.
```

As soon as you type the period following *Word*, you'll see the IntelliSense list, as shown in the following illustration (then typing *r* moves you to that alphabetic location within the list):



You can also specify a range by starting and ending character:

```
Dim range As Word.Range = d.range(2, 13)
```

Unfortunately, character counting in a Word document is zero-based, even though the document collection and other collections are one-based. Just another of those funny little inconsistencies.

You can send text from VB.NET into a Word document by using the `TypeText` method as illustrated in the previous section. You can get text from a Word document into VB.NET by using the `document.range` method (there are other ways as well). Listing 4.10 shows how to get a substring, and a full paragraph, back from a Word document.

LISTING 4.10: SENDING TEXT FROM WORD INTO VB.NET

```
Private Sub Form1_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles MyBase.Load  
  
    Dim w As Object = New Word.Application  
    w.visible = True  
  
    Dim d As Object = w.Documents.Add  
  
    With w.Selection  
        .TypeText(''This is the first paragraph. And we want to ens  
read it completely.")  
        .TypeParagraph()  
        .TypeText("Remember the Maine!")  
    End With  
  
    Dim range As Word.Range  
  
    MsgBox(d.range(0, 12).text)  
    MsgBox(d.range.paragraphs.Item(1).range.text)  
  
    w.quit()  
  
End Sub
```

Notice that the `paragraphs.Item` collection is one-based, not zero-based.

Replacing Text

It can be useful to automate the process of searching and replacing. For example, if your company changes its name from `LocalShop` to `WorldDomination`, you could go through entire folders of `.doc` files or templates, replacing the old name with the new one throughout all the documents. This might also be useful following a divorce or other adjustments in life. (To see a technique that quickly provides

you with all the .doc files in a given path, including subdirectories, see the section earlier in this chapter titled "Finding Files.")

Assuming that you have a .doc file named `test.doc` in your `C:\` folder, and that this document includes the term *LocalShop* here and there, Listing 4.11 demonstrates how to automate the process of searching for *LocalShop* and replacing it with *WorldDomination*.

LISTING 4.11: SEARCHING AND REPLACING

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

    Dim w As Object = New Word.Application
    w.visible = False

    w.Documents.Open("c:\test.doc")

    'point the document object to the opened document
    Dim d As Word.Document = w.activedocument

    Dim r As Word.Range

    d.Content.Find.Execute(FindText:= "LocalShop", _
        ReplaceWith:= "WorldDomination",-
        Replace:= Word.WdReplace.wdReplaceAll)
    While d.Content.Find.Execute(FindText:=" ", _
        Wrap:= Word.WdFindWrap.wdFindContinue)
        d.Content.Find.Execute(FindText:= " ", _
            ReplaceWith:" ", _
            Replace:=Word.WdReplace.wdReplaceAll, _
            Wrap:=Word.WdFindWrap.wdFindContinue)
    End While
    w.Documents.Item(1).Save()
    w.Documents.Item(1).Close()
    w.quit()

End Sub
```

Borrowing Word's Printing Features

Just as the previous examples illustrate how you can considerably improve VB.NET's built-in text formatting and manipulation capabilities, you can also improve VB.NET printing by borrowing from Word's more advanced features.

Printing in Word is achieved through a document's Printout method. This method has 19 properties, all of which you can set, but most of which you can leave set to their defaults (all are optional). The properties are:

| | | |
|--------------------|---------------------|----------------------|
| Background | Append, Range | OutputFileName |
| From | To | Item |
| Copies | Pages | PageType |
| PrintToFile | Collate | FileName |
| ActivePrinterMacGX | ManualDuplexPrint | PrintZoomColumn |
| PrintZoomRow | PrintZoomPaperWidth | PrintZoomPaperHeight |

This next example illustrates how to print and set properties. In this case, we'll specify that we want printing done in the background, appending to any existing job, and specifying that we want two copies. To set a particular property, you must include all preceding properties, or at least insert a comma if you want to leave a property set to its default. In other words, we need all these commas in order to let Word know that we're specifying the eighth parameter, copies: `d.PrintOut(True, True, , , , , , 2)`.

Put a TextBox and button on a form, then type Listing 4.12 into the button's Click event.

LISTING 4.12: PRINTING AND SETTING PROPERTIES

```
Private Sub Button1_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles Button1.Click  
  
    Dim w As Object = New Word.Application  
    w.visible = False  
  
    Dim d As New Word.Document  
    d = w.documents.add  
  
    d.Range.InsertAfter(TextBox1.Text)  
  
    Try  
        d.PrintOut(True, True, , , , , , 2) 'do two copies  
    Catch ex As Exception  
        MsgBox(ex.ToString)  
    End Try  
    d.Close(Word.WdSaveOptions.wdDoNotSaveChanges)  
  
    w.quit()  
End  
  
End Sub
```

After a bizarre, though necessary, set of objects are instantiated in the first five lines, you finally get the one object you're really interested in, the `MailItem` object. With it you can access quite a bit of information about each message. In this example, you display the `SenderName`, `Subject`, and `Body`—the primary data. However, if you wish, you can also display a variety of other properties of each `MailItem`, including `Attachments`, `AutoForwarded`, `CreationTime`, `Importance`, `ReadReceiptRequested`, `Recipients`, `Size`, and `VotingOptions`. There are a couple dozen other, more arcane, properties as well. If you wish, you can use this same technique to explore and display the messages in the other folders, such as the `Outbox`.

This example displays only the first e-mail message, but it's easy to use this same code to see all `Inbox` messages. To employ this technique in a finished application, use this loop to get the entire collection of `Inbox` e-mail messages:

```
For i = 1 To it.Count
```

Then you can display the actual message (`m.Body`) for whatever message header the user clicks in the `ListBox`.

You can also use the various properties as a way of filtering the messages: show all between certain dates, show all from a certain sender, and so on. To do that, you employ the `Restrict` method of the `Inbox.Items` collection. You build a filter string used as the argument for the `Restrict` method. For example, to see only those messages from Mary Stuart, you build the string using `SenderName`, one of the properties of the `MailItem` object:

```
Dim MyFilter As String = "[SenderName]='Mary Stuart'
```

And you then use that string as an argument (here the variable `it` represents the `MAPI Items` collection—see the previous example code):

```
It = Inbox.Items.Restrict(MyFilter)
```

You can use other `MailItem` properties, such as `SentOn`. In that case, your filter argument requires the `ToShortDateString` method of the `.NET DateTime` object, like this:

```
Dim d As DateTime = Now
Dim s As String = d.ToShortDateString
Dim MyFilter As String = "[SentOn] <= 's'
```

In this case, you've defined your filter as "before now," which would not exclude any messages. However, you can combine the usual comparison operators `>`, `<`, and so on to create whatever date filter you wish. What's more, you can combine various filters, such as requesting to see all messages from Mary Stuart sent before last Christmas, or *between* last Christmas (the start date of the date range would employ the `>` operator, meaning "greater than Dec. 25, 2002) and New Year's Day (this would employ the `<` operator). Just concatenate the filter strings: `MyFilter &= "[SentOn] <= 's'`, for instance. You could permit the user to choose the date range with a `DateTime Picker Control`, then translate its `Value` property using the `ToShortDateString` method. Or let the user type in a date, as a string in the format: 7/16/2003.

[Team Fly](#)

 Previous

Next 

```
n = exl.Evaluate("COS(4.3444)")  
exl.Quit()  
MsgBox(n)
```

End Sub

Notice the important difference between these two examples: Both evaluate the expression, but one takes a string, the other a numeric variable. The Excel version accepts a *string* version of the problem, evaluates it, and returns the result as a floating-point variable. This is why you can simply put up a TextBox for the user to type in an expression, which is then sent to Excel as a string. By contrast, the pure VB.NET code cannot evaluate a string. Instead, you must submit a floating-point number to the VB.NET COS function.

Let's expand the previous example for a moment. Put two TextBoxes, two Labels, and a button on a form. Then type Listing 4.16 into the button's Click event.

LISTING 4.16: PERMITTING USERS TO INPUT EXPRESSIONS, THEN EVALUATING

```
Private Sub Button1_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles Button1.Click  
  
    Dim exl As New Excel.Application  
  
    Dim n As Double  
  
    n = exl.Evaluate(TextBox1.Text)  
  
    TextBox2.Text = n.ToString  
  
    exl.Quit()
```

End Sub

Press F5 to execute this code, then type in an expression (notice that the SIN and other methods are not case-sensitive). When you click the Evaluate button, the results are displayed. A key to understanding why all this matters is the `.ToString` method. It's easy and painless to convert a numeric variable to a string—you just attach `.ToString` as illustrated in this example. However, there is no `.ToDouble` that translates a string variable into a double-precision floating point variable. But the really difficult job—which Excel's Evaluate method nicely solves—is translating $\sin(2/1)/\cos(12.2)+(2+3.21/12)/2$ into

a format that VB.NET can analyze. Sure, the .NET Math namespace understands *sin* and *cos*. What .NET cannot do is evaluate a complicated expression such as the one shown in Figure 4.1.



FIGURE 4.1 Borrow Excel's expression evaluation abilities to permit users to enter math expressions into your VB.NET applications.

You might think that you can write a Select Case structure that could translate a complex expression from a string to a format that VB.NET would understand. I imagine it's possible, but once you start trying to deal with the complexities of nested expressions, operator precedence, and other factors, you'll be happy to just feed a string to Excel and get back the result instantly. After all, one of Excel's specializations is translating user input, so why should you tackle that gruesome job in VB.NET? The *evaluate* capability is just sitting there in Excel waiting for you to utilize it.

Pretty Printing

Formatting and printing tables of data can be another daunting task facing a VB.NET programmer. There are various controls, such as the powerful DataGrid, that display tabular information effectively on screen. But what about sending nice-looking tables to a *printer*? You can't just dump the screen contents to the printer; the aspect ratio is different. Reports are usually printed in what's called portrait mode (8 1/2×11 aspect ratio), but computer screens are nearly the opposite aspect ratio and can be called landscape mode (wider than they are high).

Creating reports on hard copy is a fairly common task. Do you want to struggle in VB.NET with this job, or just turn the task over to Excel, with its specialized formatting and printing routines? Excel won't always do the best possible job, but with some fiddling, you can often get superior printouts. To print an Excel worksheet, use this code:

```
Dim exl As New Excel.Application
Dim w As New Excel.Worksheet
w = exl.Workbooks.Add.Worksheets.Add
w.PrintOut()
```

Notice the unusual *printout* method rather than the far more common *print*.

Sending Data to Excel, Formatting, Calculating, and Saving

If you need to fill an Excel worksheet with data, format cells, force calculations on data, or save a worksheet, the following example illustrates how to accomplish all four tasks. Type Listing 4.17 into the Form_Load event.

[Team Fly](#)

 Previous

Next 

LISTING 4.17: EXPORTING DATA, FORMATTING, CALCULATING, AND SAVING USING EXCEL

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

    Dim exl As New Excel.Application
    Dim w As New Excel.Worksheet
    w = exl.Workbooks.Add.Worksheets.Add

    'select and format titles
    w.Range('A1:E1').Select()
    With exl.Selection.Font
        .underline = True
        .Size = 13
        .Name = "Arial"
    End With

    ' create data
    With w
        .Cells(1, 1).Value = "    Bob    "
        .Cells(1, 2).Value = "    Sandy   "
        .Cells(1, 3).Value = "    Jane    "
        .Cells(1, 4).Value = " Snapper (Johnson)"
        .Cells(1, 5).Value = " Sales Force Total"
        .Cells(3, 1).Value = 25000
        .Cells(3, 2).Value = 42000
        .Cells(3, 3).Value = 37000
        .Cells(3, 4).Value = 6890
    End With

    ' Have the fifth column calculate the total of the first four
    w.Cells(3, 5).Value = "=Sum(A3:D3) "

    ' Make numeric data smaller
    w.Range("A3:E3").Select()
    With exl.Selection.Font
        .Size = 10
        .Name = "Arial"
    End With

    'this next formatting should be done after the cells
    'are filled with their data:
    w.Range("A1:E1").Select()
    With exl.Selection
        .Columns.AutoFit() 'make the columns wide enough
```

```
        'center the headers
        .HorizontalAlignment = Excel.XlHAlign.xlHAlignCenter
    End With

    w.PrintOut()

    'save it to disk
    Try
        w.SaveAs(''C:\temp.xls")
    Catch ex As Exception
        MsgBox(ex.ToString)
    End Try

    exl.Workbooks.Close()

End Sub
```

Managing an Excel worksheet object is similar to the way you manage a Word doc when using the range object. After creating the Excel object, and a worksheet within it, you describe the formatting for the top row of cells (the cells that will contain your column headers). You want them underlined and rather large.

Next you specify that same row of header cells that you fill with descriptive strings. Notice that when referring to the cells within a range you use the format A1:E1 (as you would when specifying a range within Excel itself). This is the same format you use to specify calculations, such as =SUM(A3:D3). However, when referring to the cells collection when you're adding data to them, cell A1 becomes instead .Cells(1, 1).

You also add the numeric data for the row representing the sales figures, but in the final column of the sales figures, you specify a calculation that adds all the previous cells in that row: `w.Cells(3, 5).Value = "=Sum(A3:D3)"`.

After that, you specify that the numeric row is 10 point, somewhat smaller than the header row's font size. And you ensure that the headers are readable by using the AutoFit method of the Columns collection to widen the columns as necessary to display the text. Finally, you center the text within the header row using `Excel.XlHAlign.xlHAlignCenter`.

At the end, you print your data to the printer, save the file, and close the workbooks.

***WARNING** Don't use the `Quit` method—it leaves an instance of Excel running in the background.*

Retrieving Data from Excel

You may find yourself wanting to import data from an Excel worksheet into a VB.NET project. Use the previous example to create an Excel worksheet located at "C:\temp.xls" as your source of data, and type Listing 4.18 into the Form_Load event.

LISTING 4.18: IMPORTING DATA FROM AN EXCEL WORKSHEET

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

    Dim exl As New Excel.Application
    Dim w As New Excel.Worksheet
    w = exl.Workbooks.Open('C:\temp.xls'). _
Worksheets.Item(1)

    exl.Range(" A1:E3 ").Select()

    'create an object to hold the imported data
    Dim r As Excel.Range = exl.Selection

    Dim s As String
    For i As Integer = 1 To 3 'requires VB.NET 2003
        For j As Integer = 1 To 5
            s = r(i, j).value
            TextBox1.Text &= s & vbTab
        Next
        TextBox1.Text = TextBox1.Text & vbCrLf
    Next

    exl.Workbooks.Close()

    TextBox1.SelectionLength = 0 'turn off the selection

End Sub
```

After using the `Open` method to access the Excel data file, you select the range within this worksheet that contains data. The next line defines an object to hold the data in one chunk (similar to the way that an array holds tabular data). Then you assign the current selection (the range) to the range object so the data is actually now in VB.NET and available for picking off in the usual way, via a nested loop.

Notice the declaration included within the `For` statements (`For I As Integer`). This shortcut is a new feature available only in VB.NET 2003. The loops pluck each data value from the range object, one at a time, and do some simple formatting with the tab and CLRF constants. After closing Excel, you then turn off the `TextBox` selection. This strange artifact—automatically selecting text added to a `TextBox`—is an undesirable side effect in VB.NET. You just have to set the `SelectionLength` property to 0 to turn it off.

This page intentionally left blank.

One clue about why the Directory object doesn't need to be instantiated is this description in the Object Browser:

```
Public NotInheritable Class Directory
    Inherits System.Object
```

The Directory object is low-level, like the Integer object—so you can just use it without instantiation. The clue? It's that this Directory class inherits right from the essence, the very heart of the .NET Framework, the base class System.Object. By contrast, the DirectoryInfo class inherits from a derived class (System.IO.FileSystemInfo):

```
Public NotInheritable Class DirectoryInfo
    Inherits System.IO.FileSystemInfo
```

About System.Object

The term *base class* is a bit too modest for System.Object; it's actually the mother of all .NET classes. They all derive from System.Object, at least implicitly. System.Object does not inherit from any deeper object—and it's the only class in the entire measureless .NET Framework that is not inherited. It's the .NET equivalent of the big bang.

Remember that .NET, unlike various C languages, does not permit multiple inheritance. (Being able to inherit from more than one parent class has been the source of a great number of bugs for C programmers, who already have enough to worry about without adding this extra source of confusion.)

Other than System.Object, all other classes in .NET inherit from one (and only one) parent class. However, a parent class can inherit from its own parent class, and so on up the line. Ultimately, at the end of the line, sits System.Object.

This is why when you look at the methods for any object in .NET, you *always* find at least the same six methods—Equals, GetHashCode, Finalize, MemberwiseClone, GetType, and ToString—no matter what other methods a class might have. Where do they come from? System.Object, of course. What do they do?

Object types are called *self-describing*, like so many other current programming elements—notably XML and its many offspring, and .NET assemblies with their metadata. Several of the six primary object methods serve the purpose of describing the object.

MemberWiseClone

MemberWiseClone and Finalize are Protected, and can thus be accessed only from a child class. Finalize, in fact, does nothing because it's overridden when inherited.

MemberWiseClone creates a "shallow copy" of the instance (the object). That means it copies the non-static fields in the object. However, it does initialize the clone's variables (fields) and properties. This works fine if the instance contains only value types, but if you want to, you can override this method and define your own method of cloning your object.

Equals

Equals tells you whether two object variables point to the same object. The .NET compiler doesn't permit you to use = (lest you get confused and think the result means equality in the sense that there

.NET's Strong Features

Although .NET was conceived and built several years before the recent "security initiative" was launched at Microsoft—pushing security issues to the fore of the company's focus—.NET is nonetheless filled with security features built into the .NET Framework and the overall .NET design. In a sense, .NET has absorbed some security features that used to be part of the operating system. This is yet another aspect of the "platform independence" that .NET claims.

.NET offers settings of considerable specificity for various types of permissions. Administrators can define with great precision which applications can do what (an application might be permitted to delete files, for example, but not access the Internet, or vice versa). Many sensitive OS elements can be specified on an application-by-application basis. By modifying configuration files, administrators can specify exactly what a .NET application is able to do.

What's more, .NET security features are abstracted from the OS in yet another way: the .NET security model is not tied to any particular version of Windows.

Other improvements include built-in self-checking features. Each .NET component is automatically scanned to ensure that its code has not been tampered with (modified, appended to, or otherwise disturbed) in any way. Optionally, you can employ Authenticode and digital signatures with your .NET applications to provide users with a measure of comfort, knowing that you are likely who you say you are and that your .NET applications can be trusted.

Also, Just In Time (JIT) compilation cooperates with the metadata available in various ways in a .NET assembly (all the dependency files that, collectively, make up an application). This cooperation permits .NET to verify each method's argument list and ensures that no wayward memory access is taking place (the classic cause of GPFs). This verification is, however, optional. It does slow things down a bit during an application's startup, so administrators are permitted to switch it off if desired.

You, an application's designer, and administrators might find it useful to run .NET applications through a utility named PEVerify (PE for portable executable). It reports whether or not a .NET application has passed the verification test. If it does pass, administrators can flip the switch bypassing the startup verification process. Look for PEVerify in `\Program Files\Microsoft Visual Studio .NET 2003\SDK\V1.1\Bin`. The results of an executable that passes are shown in Figure 5.1.



FIGURE 5.1 Use this utility to verify that a .NET application is type safe and won't otherwise cause a GPF.

If you write your applications in VB.NET, you can be sure they're type safe unless you're fiddling around with some arcane, old-style API calls. However, administrators might well be interested in verifying the safety of .NET applications.

Traditional compiled code is *unmanaged*, meaning that it runs free of oversight—in the native language. Managed code, executables produced by .NET, run under the direction of the .NET CLR (Common Language Runtime). Security improvements result when managed code is thus observed and supervised during its execution, and its behaviors can be governed. In other words, executables can be managed by a system of permissions.

You may be surprised to learn that the CLR looks at *each method* being executed—a file-save method, for instance. Is this .NET application permitted to save a file to the hard drive? To actually execute the file-save method, the CLR examines *three* kinds of permissions: role-based, identity, and code access. Let's look at role-based security first. (XP and Windows Server 2003 both include special code-access security features called Software Restriction Policies, which work with unmanaged code. These features are described later in this chapter.)

Users and Groups

Most OS security under Windows is role-based. Essentially there's a list of danger spots (the Registry, file access, security settings, disk reformatting commands, and such) and a list of the computer's users with various levels of permission granted to each user. Interestingly, every single file on the hard drive is assigned a security descriptor. Certain roles (or groups) such as administrators are permitted wide access. Other roles, such as guests, have highly restricted permissions. Individual users can be assigned to one or more groups to define their permissions. As a result, certain kinds of code, as well as certain OS features, cannot be executed by certain users.

Administrators can also adjust system object security settings using various utilities—even Internet Explorer and Windows Explorer can be used to adjust file and folder access levels.

This approach combines the classic role-based approach (does the administrator give Susan permission to delete files?) with classic code-based security (does the administrator give code located in the "Intranet Zone" permission to delete files?).

On XP systems, go to Control Panel ➤ Administrative Tools ➤ Local Security Policy. In the tree choose Local Policies ➤ User Rights Assignment (to define group behaviors, such as whether Users and Power Users are permitted to back up files, change the clock, and so on). If you're not able to see these options, you're not an administrator and don't, yourself, have permission to manipulate others' permissions and behaviors.

The Principal

Role-based security creates a profile of permissions for each user, often by assigning the user to a particular "group," which is defined as allowed to do certain things, and prohibited from doing other things. Special groups (Administrators, for example) may have *everything*-level permission—they can view all logs, reformat hard drives, and so on.

.NET uses what it calls a "principal" object that behaves like a proxy for each user and interacts with "identity" objects that the runtime employs as a way of telling users apart. In the `System.Security.Principal` namespace you find both Windows principals (the `WindowsPrincipal` class), which map directly to existing Windows groups, as well as user's membership in those groups.

In addition, this same namespace includes the `GenericPrincipal` class, which manages the .NET-specific role-based security. You can also have your .NET applications specify custom principals as well.

Try this code (Listing 5.1), which illustrates two objects you can examine—the `WindowsIdentity` and the `WindowsPrincipal`—and then manage behaviors on a custom basis.

LISTING 5.1: EXAMINING WINDOWSIDENTITY AND WINDOWSPRINCIPAL

```
Imports System.Security.Principal  
Imports System.Threading
```

[Team Fly](#)

 Previous

Next 

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

    Dim myDomain As AppDomain = Thread.GetDomain()

    Dim p As WindowsPrincipal

    myDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal)

    p = CType(Thread.CurrentPrincipal, WindowsPrincipal)

    Console.WriteLine(p.Identity.Name.ToString())

    Dim WinRoles As Array = [Enum].GetValues(GetType(WindowsBuiltI
    Dim rName As Object

    For Each rName In WinRoles
        Try
            If p.IsInRole(CType(rName, WindowsBuiltInRole)) Then
                Console.WriteLine('The current user is in the " &
rName.ToString & " group.")
            Else
                Console.WriteLine("The current user is NOT in the
rName.ToString & " group.")
            End If
        Catch
            Console.WriteLine("No specification for the role " & r
        End Try
    Next rName

End Sub
```

For example, if executing Listing 5.1 reveals that this user is a member of the Administrator group, then you would permit them to view a secret log. Otherwise, your application would not take that action. Or you might disable some buttons on your user-interface form based on the security level permitted the current user. Are they forbidden to use the printer (*not* a member of Print-Operator group)? Disable the Print button in that case.

Alternatively, you can get some information from the `WindowsIdentity` object, or pass that object to the `WindowsPrincipal` object:

```
Dim w As WindowsIdentity
w = w.GetCurrent
MsgBox(w.Name & "Is this person a member of the guest group? " & w.IsGuest)
Dim p As WindowsPrincipal = New WindowsPrincipal(w)
```

In either case, you can get the user's name and the various groups (and therefore permissions) to which the user belongs.

Code-Access Security

Code-access security (CAS) grants or refuses permissions based not on the identity of the user but rather on the identity (or authenticity) of code or assemblies. Where does the code come from? What permissions does this code have? Typically, *code access* refers to a situation where one application is attempting to consume an object located in a different application, or exposed as a Web service method, for example. It means that code (as opposed to a particular user) is attempting to access your application or one of its public members.

The answer to "Does this code have permission to consume me?" can be provided by hashing (see Chapter 6 for demonstrations and details on this technique), or by the URL identifying the assembly's origin, by digital signatures, or by other means of verification.

However, CAS does not ignore or violate any role-based security settings. If some outside agency does not have permission from Windows security to delete a file, nothing in CAS can grant that permission and override Windows itself.

As with role-based security, administrators specify CAS permissions, and base these permissions on known origin of the assembly. If the origin is suspect, access can be denied to various system resources.

Code-access security allows you to specify which resources can be accessed by a particular body of code. It allows you to specify, for example, that an entire class refuses file access (no matter what the caller's permission levels), by adding an attribute, like this:

```
Imports System.Security.Permissions
<FileIOPermissionAttribute(SecurityAction.Deny)> Public Class MyNewClass
End Class
```

You can use code access in another way, as well. The previous example is called *declarative* because you add an attribute to the declaration of an assembly, class, or member. There are a slew of *attribute* classes in addition to the `FileIOPermissionAttribute` used above.

Alternatively, code-access security can be enforced via what's called the *imperative* mode, which is used for members only, not entire assemblies or classes. To employ this approach, instantiate an object from one of the permission classes and then call one of its action methods. Use code like this for the imperative approach:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Load
    Dim p As New FileIOPermission(PermissionState.Unrestricted)
    p.AllLocalFiles = FileIOPermissionAccess.AllAccess
End Sub
```

Everyone Must Agree

Security is relatively straightforward for traditional single-machine applications, such as a word processing program that you install from a CD on your personal computer. Security for distributed or

[Team Fly](#)

 Previous

Next 

dialog box query. They just guess, balancing their fear of virus infection against their need for whatever mobile service they're about to permit entry.

Let's face it—our local hard drive is no longer our primary source of data, nor in the future is it likely to remain even the primary source of executables. Your machine is susceptible to many sources of intrusion, including e-mail, subscription applications, Web services, automated updating, and so on.

Recall that most computer security tactics depend on identification: Does the user's logon ID match the password? Does the user belong to a group with permission to delete files? Does this incoming code originate from a trusted source? And so on.

How, then, can unidentified incoming code be safely executed, even when no logon/password or permission level has been established between the remote server and your computer?

.NET's answer to this question is CAS, providing several levels of trust (just as the Windows system offers various groups different levels of access to vulnerable resources). One useful aspect of CAS is that before you distribute your application or publish it as a service, you can specify which behaviors your code is permitted to carry out (can it delete a file?). If you say *no* to file deletion, then any attempt to delete a file from within your application violates the CAS protection and indicates that your application code has been infected with a virus. It's as if you tell a customer: I'm sending a messenger to see you, but she won't ask to come into your apartment. If she does, she's an impostor, so lock her out.

To summarize, CAS does the following:

- ◆ Lets administrators map permission levels to code groups
- ◆ Examines each assembly and gives or withholds permission (looking at both permissions allowed by the local security policies and permissions the code itself requests)
- ◆ Specifies individual permissions or sets of permissions that relate to system resources
- ◆ Allows your .NET source code to ask for permissions—those it must have, those it would like to have, and those it should never request (unless it's been invaded by a virus)
- ◆ Allows you to specify within your .NET code that any callers have a digital signature, or make it otherwise known that they possess specified permissions levels

As you see, CAS involves cooperation between you, the creator of a VB.NET application, and the end user or administrator. (Remember, too, that Internet Explorer contains its own group of security settings—and that by default, mobile code is enabled in IE.)

Administrators must be aware that several levels of permissions interact—it may be necessary, for example, to adjust CAS-level settings, Windows security policy, *and* browser settings for even a simple activity such as file reading via mobile code. If any of these layers of security refuse permission, the file reading cannot take place.

Administrators must usually modify these various security settings on an application-by-application (or mobile code source) basis. If the administrator wants to increase trust levels for a particular application or mobile source, there are usually several locations where that trust specification must be adjusted. Recall that Windows security settings cannot be overridden by any CAS settings. *Both* must offer permission for any behavior that they both govern (commonly, access to hard drives or other storage media, printers, and the Registry).

CAS Config Files

CAS looks for its security settings in three files containing XML documents, but, as you might suspect, you have to be an administrator to modify these files (you should probably take steps to ensure that power users cannot modify them). The files represent the settings for the user, the machine, and the enterprise. Let's go down into the dungeons where the secrets are kept. Find the machine level file here:

```
Windows\Microsoft.NET\Framework\v1.1.4322\CONFIG\security.config
```

The version number portion of the path can differ (v1.1.4322 in this example path), and your Windows directory might also have a different name. But you'll find it. Look at the file to see the various settings. The enterprise file is in the same path and is named `enterprisesec.config`. The user file can be found in this path for NT, Windows 2000 and XP:

```
Documents and Settings\Your User Name\Application data\Microsoft\CLR security config\ v1.1.4322\security.config
```

For Windows 95 and 98:

```
Windows\Your User Name\CLR security config\ v1.1.4322\security.config
```

The *Your User Name* location is the name you log onto the computer with.

***TIP** These files can be accessed via the .NET Framework Configuration Tool described later in this chapter.*

Descriptors

Technically, the OS builds a security descriptor for each new system object (folders, files, printers, and so on) that's created. This descriptor is just a little array of permission switches: the ACE (access control entry) permission lists the user's access permissions regarding the object, including whether this user can delete, modify, read, or change the owner of, the object. The most significant permission list in a security descriptor is the DACL, Dynamic Access Control List. It lists permissions governing access to the object.

When a VB.NET application is executed, both CAS and DACL settings must agree on various permissions. Put another way, DACL must permit your .NET application's assembly, and the user executing your application, to, say, access a particular folder or file.

Software Restriction Policy

You may have heard about Windows's Software Restriction Policies (SRP), available in Windows Server 2003 (formerly known as Windows .NET Server) and Windows XP. This is yet another set of locks on the door. With it, administrators have the ability to specify code-identity-based policies, which are separate from any role-based policies in effect.

You might think: What, another layer of code-based security for .NET assemblies or applications? Fortunately, SRP is mutually exclusive with .NET CAS. In other words, if CAS is in effect, SRP is turned off. And vice versa. CAS works only with managed code (running under the .NET Common Language Runtime). SRP turns on only when *unmanaged* native code is executing. CAS

TABLE 5.1: XP CODE EXECUTION SECURITY PERMISSIONS

| BEHAVIOR | HIGH | MEDIUM | MEDIUM-LOW | LOW |
|---|-------------|---------------|-------------------|------------|
| Download Signed ActiveX Controls | Disable | Prompt | Prompt | Enable |
| Download Unsigned ActiveX Controls | Disable | Disable | Disable | Prompt |
| Run ActiveX Controls and Plug-Ins | Disable | Enable | Enable | Enable |
| Initialize and Script ActiveX Controls Not Marked as Safe | Disable | Disable | Disable | Prompt |
| File Download | Disable | Enable | Enable | Enable |
| Font Download | Prompt | Enable | Enable | Enable |
| Access Data Sources Across Domains | Disable | Disable | Prompt | Enable |
| Allow Meta Refresh | Disable | Enable | Enable | Enable |
| Display Mixed Content | Prompt | Prompt | Prompt | Prompt |
| Don't Prompt for Client Certificate | Disable | Disable | Enable | Enable |
| Drag and Drop or Copy and Paste Files | Prompt | Enable | Enable | Enable |
| Installation of Desktop Items | Disable | Prompt | Prompt | Enable |
| Launching Programs or files in an IFRAME | Disable | Prompt | Prompt | Enable |
| Navigate Subframes Across Different Domains | Disable | Enable | Enable | Enable |
| Software Channel Permissions | High Safety | Medium Safety | Medium Safety | Low Safety |
| Submit Non-Encrypted Form Data | Prompt | Prompt | Enable | Enable |
| Userdata Persistence | Disable | Enable | Enable | Enable |
| Active Scripting | Disable | Enable | Enable | Enable |
| Allow Paste Operations | Disable | Enable | Enable | Enable |
| Allow Paste Operations via Script | Disable | Enable | Enable | Enable |
| Scripting of Java Applets | Disable | Enable | Enable | Enable |

Now it's time to turn our attention from this code-access security overview and see how to use some important security tools.

Using the Framework Configuration Tool

Administrators can use various tools provided with .NET to manipulate security policies. The .NET Framework Configuration Tool is an MMC snap-in; the Code Access Security Policy Tool is a utility named caspol.exe.

[Team Fly](#)

 Previous

Next 

are two distinct objects but they have identical members and are otherwise identical internally, including their data-clones, in other words). You use Equals, like this:

```
Dim obj1 As New Object
Dim obj2 As New Object
obj1 = obj2
If obj1.Equals(obj2) Then MsgBox("They are the same.")
```

When a class inherits an object type, the Equals method is supposed to be overridden. You provide a custom method to determine equality. Similarly, you should override the GetHashCode method to ensure it remains consistent when two objects are equal.

Testing for equality can be tricky in computer programming because the idea of "equality" (two objects holding an equal value, such as 2.114) can be confused with "identity" (two object variables pointing to the same object). The Equals method simply compares pointers, but many classes over-ride this method to force it to compare values (for example, the Integer class Equals method tells you whether two integer variables contain the same number).

ReferenceEquals

The System.Object type includes an additional equality testing method named ReferenceEquals. It returns True if the two object variables being compared *refer* (point to an identical address in memory) to the same object. ReferenceEquals cannot be overridden. (Think of ReferenceEquals as doing the job of the traditional Is comparison operator in classic VB.)

For example:

```
Dim l As Object
Dim m As Object
Dim n As New Object
    Console.WriteLine(Object.ReferenceEquals(l, m))
    m = n
    Console.WriteLine(Object.ReferenceEquals(m, n))
    Console.WriteLine(Object.ReferenceEquals(l, m))
```

results in:

- True (both refer to nothing—they are object variables but don't point to any object yet). This can be very confusing because these are *two different object variables*, yet they return equality.
- True (both *m* and *n* refer to the actual object *n*. *N* is an object because you instantiated it with the New command).
- False (*l* remains an object variable that has not been pointed to any object).

(Notice the unusual use of the term Object in Object.ReferenceEquals. Where is this Object ever instantiated so that you can use it in your source code? It's not. It's just there! Fortunately, this syntax is extremely rare in VB.NET programming.)

[Team Fly](#)

 Previous

Next 

In Control Panel, open Administrative Tools, then double-click *Microsoft .NET Framework 1.1 Configuration* (or it will be 1.0 if you don't have the latest version of .NET).

This tool contains a considerable collection of features for specifying how .NET security is enforced (and for manipulating most other configurable aspects of .NET). For example, choose the Applications node at the bottom of the tree and click the Add An Application To Configure link; you'll see a list of .NET applications that have been executed on this machine, as shown in Figure 5.2.

For an example of specifying particular permission sets (role-based) for users, click Everything under User, then click the Change Permissions link and you'll see the Create Permission Set dialog box shown in Figure 5.3.



FIGURE 5.2 Manage elements of individual .NET applications using this dialog box.





FIGURE 5.3 Build your own .NET permission configuration with this dialog box.

Or expand the Runtime Security Policy node and then expand the Enterprise, Machine, or User policy nodes, and you'll see how to manipulate permission sets, code groups, or policy assemblies.

If an administrator wants a simpler, though less specific, approach to specifying role-based policies, choose Microsoft .NET Framework 1.1 Wizards in the Administrative Tools dialog box in Control Panel. Then double-click the Adjust .NET Security icon and follow the instructions the wizard gives you, as shown in Figure 5.4:

Return now, though, to the options available to you in the .NET Configuration 1.1 dialog box. Right-click the Runtime Security Policy node, as shown in Figure 5.5:

NEW AND OPEN OPTIONS

The New and Open options are for testing policy settings, so you can see the effects without actually committing to the policy. Perhaps you want to build a policy on your computer, then when you've perfected it you can deploy it across your intranet. Or you want to view and test previously saved policy files, or try out some new policies. For whatever reason, the New and Open options permit you to experiment without necessarily committing the changes.

You can create a new policy and try out any of the adjustments available for .NET assembly security, then right-click the Runtime Security Policy node, as shown in Figure 5.5, and choose Evaluate Assembly to test your adjustments and see if there are any effects on one of your .NET applications (assemblies) that is affected by the adjustment.

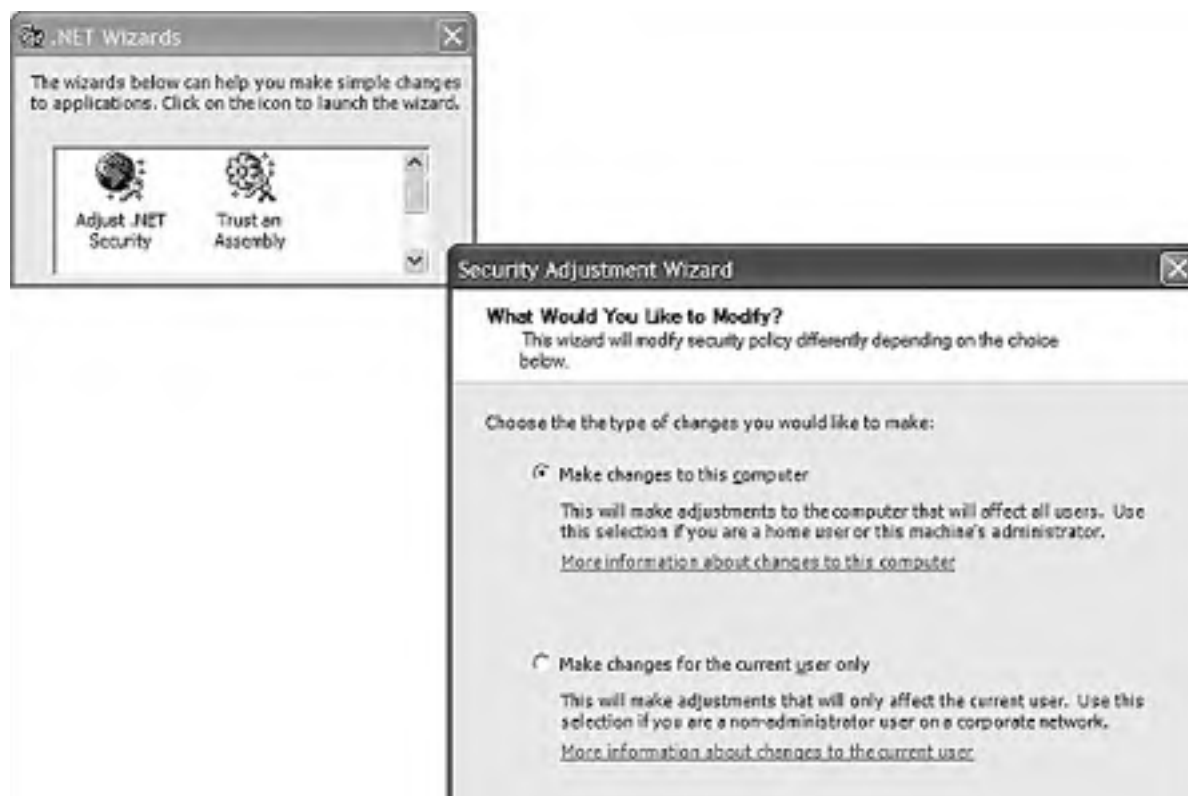




FIGURE 5.4 Administrators can adjust .NET security policies using this dialog box.

RESETTING

You can choose the Reset All option (shown in Figure 5.5) to restore the default policies, which are primarily defenses against code coming in from outside the local machine (Internet or intranet). In case you decide that it's easier to modify existing custom policies than to start over from scratch with the defaults, it's a good idea to first save the current policies before choosing to reset.

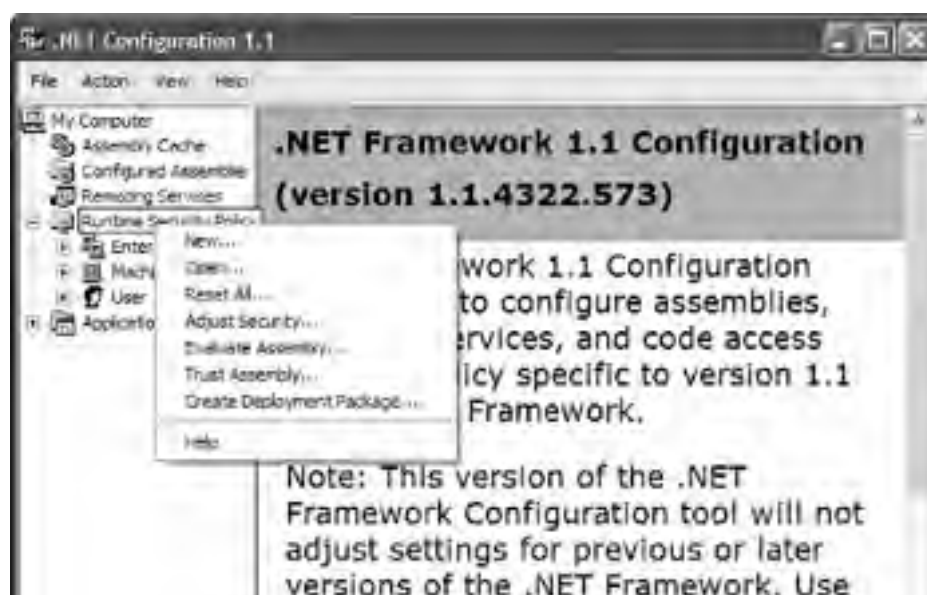
ADJUSTING SECURITY

The Adjust Security option (shown in Figure 5.5) allows you to make large-scale adjustments (as opposed to the more detailed changes you can make to specific sources or assemblies). Use the Security Adjustment Wizard to make global adjustments to all assemblies from an entire zone (such as the local computer zone or the intranet zone).

This Wizard is also good for quick responses to sudden security problems. You can use it to slam some doors shut globally until you can ferret out the source of the problems.

Say, for example, that there's a wacky hacker in your office. You don't know *who* is peeping into other people's files (is it Nicky? probably) but you're not yet *positive* that it's Nicky who's been telling everyone details about the list of salaries that he found by running an application located in accounting. You do need to take action before locking the suspect in a room with the personnel director and the bright lights.

Run the Security Adjustment Wizard and slam Nicky's intranet freedoms shut temporarily. You can do this by changing all code interactions from the intranet for Nicky's computer (and leave the rest of the office's computers alone). Start the Wizard by right-clicking the Runtime Security Policy node, as shown in Figure 5.5. Choose Adjust Security, then you see the Wizard as shown in Figure 5.6.



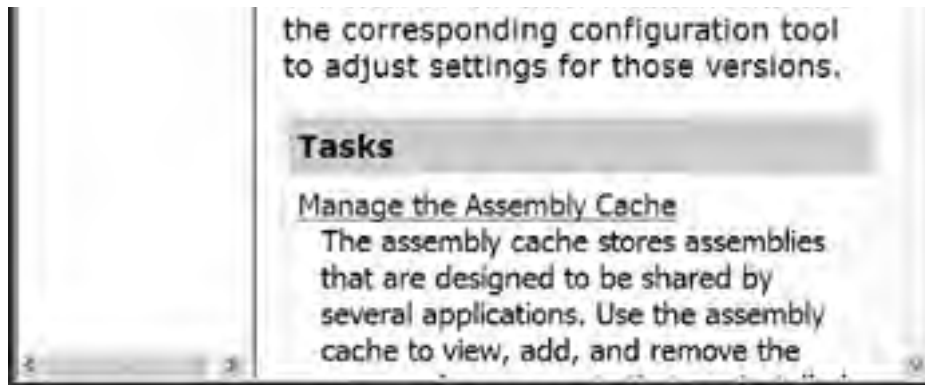


FIGURE 5.5 Choose several options from the Runtime Security Policy node.

However, you may want to beef up security in your Web services and other Internet/intranet-based applications. You may want to build in additional demands for validation, verification, permissions, trust-levels, or authentication.

The danger spots if you're exposing your code to strangers are the openings, the places where outsiders are allowed in. Just as you want to guard against entry via windows and doors in a house, you want to prevent outsiders from misusing exposed methods in your code. You have to also watch out for incoming data streams, network connections or other locations where outsiders are allowed in, and are therefore sources of possible attack. Here are some ideas to consider when writing a .NET program that you want to assure your customers is as safe as you can make it:

Assume that your source code either will be directly available to the intruder or can be reverse engineered. Don't, for example, embed secret keys—they likely won't be very secret. Also, don't think that you've successfully "hidden" security defenses within your code by making them obscure, convoluted, or disguised.

You *should* take these measures—obscuration and disguise are important security techniques. Make things as difficult as you can for hackers. Just don't rely on these tactics alone. There are too many ways that an outsider can get hold of source code (the company is sold, consultants request the code, someone carelessly leaves it lying around, and so on). Many security problems are caused by insiders—disgruntled workers, or people who are fired. Also, what you might think is a difficult, tricky maze can often be quickly solved by outsiders using special tools that can provide them with lists of threads and call stacks instantly. Consider this: Most people think that mazes are difficult to solve, but they actually aren't. All you have to do to get through *any* maze is to always turn left at every junction. (You could always turn right, as well, but the point is to consistently make the same turn.)

Also assume that hackers will test your public interface, rattling away at the bars on your windows to see if anything comes loose. For example, you shouldn't assume that they'll employ your methods by following the rules. Unexpected behavior is their stock in trade.

Some experts suggest that you always start off declaring *all* your members (properties, methods, and so on)—and, indeed, entire classes—*Private*. Only later, and only when compelled by the necessities of the communication needs of your application, should you extend the members' and classes' scope to *Friend* or, if absolutely necessary, *Public*. If, for example, a property must be exposed, *don't* make the property variable itself public. Instead, force outsiders to pass through property procedures to get to the variable:

```
Private m_title As String
Public Property title() As String
    Get
        Return m_title
    End Get
    Set(ByVal Value As String)
        m_title = Value
    End Set
End Property
```

Ensure—as much as possible—that incoming data is not going to foul things up. You can do this in several ways: by insisting that the identity of the sender be verified via digital signatures or other hashing, by verifying that the data is in the correct format (and otherwise trustable), and by refusing

[Team Ely](#)

 Previous

Next 

to accept input from any but expected sources. For example, in Windows, administrators can specify read-only for certain files or directories—and you can take advantage of this fact to prevent people from tampering with your incoming data. And, of course, watch any incoming *code* especially carefully.

Finally, you can employ the built-in .NET CAS features to insist that a caller pass a test before you allow them to employ a sensitive feature in your application. This test might be that they provide a password, that they have a particular permission, or that they otherwise authenticate themselves.

You could demand this test (which throws a security exception if the condition is not met—if the caller doesn't have the specified permission) at the very beginning of your application's execution path, during initialization. In that way, it functions as a kind of logon demand. Or you could demand the test at key points within your classes or methods. Listing 5.2 is an example.

LISTING 5.2: USING DEMAND TO TEST CALLER PERMISSION LEVEL

```
Imports System.Security.Permissions

Public NotInheritable Class SecurityCheck

    'instantiation of this class will require that they pass
    ' a parameter to the Sub New below (because THIS constructor is p
    Private Sub New()
    End Sub

    'This is the only way the caller can instantiate this class:

    Public Sub New(ByVal password As String)

        'You can demand a particular password here, if you wish,
        'though this is minor protection given that source code is
        'where you would store the validation of this password.

        'Here next you validate the permission level of the
        ' caller (and any callers of the caller)
        'If the caller does not have unrestricted File I/O
        ' permissions, an exception is thrown here
        'during instantiation and this SecurityCheck object is never

        Dim p As New FileIOPermission(PermissionState.Unrestricted)
        p.Demand()

        ' Now we 've checked the caller (and all their callers too)
        ' have the necessary permission, we can get on with the
        ' business of setting up the database connection and other
        ' housekeeping tasks.

    End Sub
```

Hackers try to accomplish two primary goals: mess things up using virii (or worms or other variations on virii), or peep at private information to learn secrets. This chapter dealt with the features that .NET has built in as defenses—and steps a programmer can take—to try to thwart hackers from achieving their first goal.

In the next chapter, you'll learn to employ the other great aspect of security: hiding data and protecting privacy via encryption. The .NET Framework offers you several quite powerful encryption engines. Some of these technologies were not even legal only a few years ago; they're so effective that the U.S. government banned them for fear that it couldn't eavesdrop on the bad guys' wicked messages and nasty plots. Chapter 6 provides you with code you can plug into your applications to add functionally unbreakable cryptologic protection for any secrets you, or your customers, might have.

This page intentionally left blank.

The Main Point about Equality

If you need to test for value equality, you can rely on the `=` test in most ordinary programming. To test for reference equality (two object variables point to the same object), you can use the `ReferenceEquals` method. Take a look at this code:

```
Dim a As Object = 1
Dim b As Object = 2
    If a.Equals(b) Then MsgBox("'a.Equals b")
        'a = b
    If Object.ReferenceEquals(a, b) Then MsgBox("a.ReferenceEquals b")
```

If you run this you see no message box. Object `a` holds a value of 1 and object `b` holds 2, so they are not "value equal." (`If a = b Then` would also fail.)

Remove the `'` comment from `a = b` and you've assigned `a` to point to (to *reference*) the same object as `b`, so the `ReferenceEquals` message box is displayed.

Change object `a` to `= 2` and the value equals (`.Equals`) message box is displayed because now the values held in these two objects are identical.

Note that in .NET objects are of two primary types: value and reference. Value types tend to be simple and small (all the numeric types are value types: integers, byte, char, single, double, boolean, decimal). Enum and structure are also value types. Value types execute faster.

When value types are compared for equality, the actual values (323 versus 62, for example) are compared. When reference types are compared for equality, only the *addresses* (pointers) where the objects sit in memory are compared. A reference equality means that two object instances (two different object variables that point to an instantiated object) *point to the same object*. It's like having two names for the same thing: *President* and *Bush*.

Reference types are larger and more complex: classes (and interfaces), arrays, and strings are all examples of reference types.

As usual, these classifications are important because you sometimes need to know the distinction between reference equality and value equality when you're writing a program. Also as usual, *there is an exception*. Strings are *technically* reference types but in practice they behave as value types: copying a string copies the actual string in memory (the value) not merely the reference (the pointer to an address). And when you test for the equality of strings, you are testing them as if they were value types—in other words, if two *different* strings hold the same value, they return `True`.

You'll be delighted to know that the experts have a term describing this hermaphroditic behavior on the part of .NET strings: If a reference type now and then acts like a value type (as strings do), then that type is said to possess "value semantics." Strings are technically reference types, but in practice they behave as value types. Fun, isn't it? Merrily we go along classifying this and that, happy little biologists that we are! Then we see a platypus laying an egg. Reality, like programming linguistics, always seems to have a way of messing up our neat little categories.

Technically, strings do not derive from System. Value Type, and they are also technically passed by reference when used as an argument in a function (you don't see this, though; it all goes on in some

automatically generated for you by the RSA encryption algorithm, as you will see later in this chapter. With that approach, your only problem is guarding the keys from prying eyes.

.NET cryptography includes sophisticated hashing functions. Hashing produces a unique value from a set of bytes. Hashing can be used to uniquely identify messages and various other objects. For example, a hash value for an assembly provides a unique ID number that can be used to prevent name-space collisions. Hash values can be used instead of digital signatures to authenticate information.

For our purposes, hashing can be used to translate a user's password into a unique key to be used for certain kinds of encryption and decryption. A hash value is extremely sensitive—changing merely a single bit in the source bytes results in a totally different hash value. Take, for example, the different hash values for *hand* and *hang*:

hand 150,33,162,68,164,71,236,116,149,152,248,50,117,96,255,134,9,94,44,38

hang

130,78,230,138,140,129,113,112,135,212,49,214,107,110,250,168,182,49,71,205

As you can see, changing a single character vibrates through the entire structure, affecting everything. Even changing a single *bit* would have an extensive effect on the result. This effect is quite useful because it makes deducing the original submitted bytes from the output hash value extremely difficult. And you don't want intruders deducing your password should they come upon your hash value.

Here are the Imports statements you need for the examples in this chapter:

```
Imports System.Security.Cryptography
Imports System.Text
Imports System.IO
```

If you want to use the strongest encryption offered by .NET, you must have the "high encryption pack." If you have XP, Internet Explorer 5.5 or later, or Windows 2000 with Service Pack 2, then high encryption is already in your operating system. Otherwise, download it from the Microsoft website. Without this update, key lengths are restricted. Key length plays a big part in how secure encryption systems can be.

***NOTE** For many years the government attempted to prevent strong encryption algorithms from public use, on the theory that criminals and foreign agents would be able to communicate free from government eavesdropping. However, restrictions were eventually lifted, hence the strong encryption pack from Microsoft.*

Hashing a Password

And here's a function (Listing 6.1) that returns a 20-byte array containing a hash value for whatever password you provide to it:

LISTING 6.1: GETTING A HASH VALUE


```
Private Sub Form1_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles MyBase.Load  
  
    Dim k() As Byte 'hold the returned hash value (20 bytes retur  
    k = makehash("morph")
```

[Team Fly](#)

 Previous

Next 

```
'display the hash value:
For i As Integer = 0 To 19
    Console.Write(k(i) & ',')
Next

End Sub

Public Function makehash(ByVal password As String) As Byte()

    ' Byte array to hold password
    'the size of this array affects the hash value
    Dim arrByte(password.Length - 1) As Byte

    'translate password into a byte array of ASCII values:
    Dim AscVals As New ASCIIEncoding
    Dim i As Integer = 0
    AscVals.GetBytes(password, i, password.Length, arrByte, i)

    Dim hashSha As New SHA1CryptoServiceProvider
    'Get the hash value of the password
    Dim arrhash() As Byte = hashSha.ComputeHash(arrByte)
    Return arrHash

End Function
```

Notice that it does matter what size byte array you feed into the ComputeHash method. Trailing spaces count. The strings "hope" and "hope " return different hash values.

Also, no matter what or how many source bytes you feed to a hash function, it always returns a predetermined number of bytes for its hash value. You can stream an entire file or a large assembly into a hash function, not merely a little password. But if you use the SHA1 algorithm, you always get back 20 bytes as your hash value, no matter how large the file you feed in. Here are the classes you can use in .NET to get hash values, along with number of bytes returned as the hash value:

| Class name | Hash Value Size in Bytes |
|---------------------------|--------------------------|
| SHA1CryptoServiceProvider | 20 |
| SHA256Managed | 32 |
| SHA384Managed | 48 |
| SHA512Managed | 64 |
| MD5CryptoServiceProvider | 16 |

Nearly everyone, including banks and the government, uses the SHA1 version, and we'll stick with it in this chapter as well. Although you get 20 bytes from SHA1, you need not use them all when providing a key to an encryption/decryption function, but in general the longer the password and key the more secure the encryption becomes. However, you must provide an identical hash value, the same number of bytes (the first eight, for example), to both the encryption and decryption functions. This requirement applies to what is called a symmetric encryption routine, which is the first type of routine we'll deal with in this chapter. It's called DES (Data Encryption Standard) and is a famous and frequently used contemporary encryption system.

The versions of SHA that offer larger values are designed to provide protection against brute force attacks (a computer speeding through all possible keys, until one of them unlocks the cipher-text). Roughly speaking, brute force attacks require exponentially greater time for each bit you grow the key. A key of eight bits, for example, requires 2^8 , but a key of nine bits requires 2^9 .

Hashing a File

One security danger is tampering. Someone might intercept a transmitted file or e-mail message and change it. For example, they might grab their bank account file and change a \$100 deposit into a \$100000 deposit. One way to ensure that no one has tampered with a file or other message is to create a hash value for that file.

Hashing a file provides you with a kind of super checksum—remember, if even a single bit is altered in the file, the file's entire hash value will then be quite different. Add some zeros to a deposit and the file's hash value will be very different.

You can stream a file into the ComputeHash method. Listing 6.2 shows how.

LISTING 6.2: COMPUTING A HASH VALUE FOR A FILE

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

    Dim objFilename As FileStream = New FileStream("c:\test.txt"
    FileMode.Open, _
    FileAccess.Read, FileShare.Read)
    Dim hashSha As New SHA1CryptoServiceProvider

    'Get the hash value of the file
    Dim arrhash() As Byte = hashSha.ComputeHash(objFilename)

    'display the hash value:
    For i As Integer = 0 To 19
        Console.Write(arrhash(i) & ",")
    Next

    objFilename.Close()
```

End Sub

[Team Fly](#)

 Previous

Next 

```
'define an 8-byte key
Dim k() As Byte = {12, 13, 44, 22, 21, 44, 22, 128}

'define an IV
Dim Vector() As Byte = {2, 4, 46, 141, 8, 5, 99, 2}

'Create a file stream
Dim fs As New FileStream( 'c:\testFile.txt", _
FileMode.Create, FileAccess.Write)

'create a byte array holding the message:
Dim barray As Byte() = (New UnicodeEncoding).GetBytes _
("Marva Johnson writ into the stream..")

Dim des As New DESCryptoServiceProvidier
'stuff the key & IV
des.Key = k
des.IV = Vector

Dim desencrypt As ICryptoTransform = des.CreateEncryptor()

'create a cryptostream, specifying DESencrypt
' as the transform (the encrypting scheme)
Dim cryptostream As New CryptoStream _
(fs, desencrypt, CryptoStreamMode.Write)

'save the encrypted file to the hard drive
cryptostream.Write(barray, 0, barray.Length)

cryptostream.Close()

End Sub

Public Sub decrypt()

'define an 8-byte key
Dim k() As Byte = {12, 13, 44, 22, 21, 44, 22, 128}

'define an IV
Dim Vector() As Byte = {2, 4, 46, 141, 8, 5, 99, 2}

Dim des As New DESCryptoServiceProvidier
'stuff the key & IV
des.Key = k
```

```
des.IV = Vector

'NOW BRING IT BACK IN AND DECRYPT IT

'create the input file stream
Dim fs As New FileStream("c:\testFile.txt", FileMode.Open, F

'create a decryptor from the crypto service provider
' (uses same key assigned earlier)
Dim desdecrypt As ICryptoTransform = des.CreateDecryptor()

'create a cryptostream for reading the file in, and decrypt i
Dim cryptostreamDecrypt As New CryptoStream _
(fs, desdecrypt, CryptoStreamMode.Read)

'display the result
MsgBox(New StreamReader _
(cryptostreamDecrypt, New UnicodeEncoding).ReadToEnd())

End Sub
```

Both the encrypt and decrypt procedures here are similar (DES is a *symmetrical* cryptographic algorithm, after all). First you must create a couple of byte arrays to hold the key and initialization vector (described below). For ordinary DES, the default size for both of these arrays is eight bytes.

You're also going to use two types of streams, a regular file stream (*fs* here) and a cryptostream that "wraps" around the filestream, thereby enciphering the stream. This use of streams prevents you from having to actually store the plaintext on a disk file where it becomes vulnerable to intruders. If you first store a plaintext on a hard drive in order to encrypt it, even if you delete it when you're finished, the data is usually still there. All deleting does is to remove the file's entry in the file allocation table. Even moderately sophisticated intruders have no problem reading hard drives without benefit of the allocation table.

Of course, many times you'll find it a practical necessity to store plaintext on hard drives. Nonetheless, cryptostreams and memorystreams offer you the option of operating on byte arrays in volatile memory, for example, rather than committing your secrets to a disk file. Instead you can just stream the output to the input of some other object.

In the above example, the plaintext is never committed to the hard drive, residing only in the source code in this byte array:

```
'create a byte array holding the message:
Dim barray As Byte() = (New UnicodeEncoding).GetBytes("Marva Johnson writ into
stream..")
```

After creating this byte array, you create a DES CryptoServiceProvider object and assign your key and IV (initialization vector) to it. If you omit either the key or IV assignments, they are concocted for you by the CryptoServiceProvider (the key and vector will contain random values). If you are

sending a message or storing a ciphertext (as here), you need to use the same key and initialization vector for both the encrypting and decrypting processes. For example, if you omit specifying the IV during decryption, a random IV will be supplied and as a result the first four bytes of the restored plaintext will be wrong.

Finally, you define an ICryptoTransform object that embodies both encryption and decryption methods and which contains the information about the key, the IV, and the encryption process (DES here), as shown in Listing 6.4. You use this object with a cryptostream class and your filestream (in this example) to actually encrypt the plaintext being streamed. Note that if you didn't specify a filestream (fs) when defining the cryptostream, the cryptostream's Write method wouldn't write to disk. The target could just as easily be a memorystream, for example, rather than a filestream.

LISTING 6.4: USING A CRYPTOSTREAM

```
Dim desencrypt As ICryptoTransform = des.CreateEncryptor()  
  
    'create a cryptostream, specifying DESencrypt  
    ' as the transform (the encrypting scheme)  
    Dim cryptostream As New CryptoStream _  
(fs, desencrypt, CryptoStreamMode.Write)  
  
    'save the encrypted file to the hard drive  
    cryptostream.Write(barray, 0, barray.Length)  
  
    cryptostream.Close()
```

The cryptostream object's constructor takes these arguments:

```
CryptoStream (Stream argument, ICryptoTransform transform, CryptoStreamMode mod
```

The mode can be either read or write. When you press F5 to run this example, you get back the original plaintext about Marva, but all that resides on the hard drive is the ciphertext

```
7"t~ 'f•ñt'—ò"oär—• oÍš• •%oŠ_6i-  
ÜYÓ4if°Oi• ooiP• Mfç>+a"° K!i5@*srÒ\çt!• u*ãÝIY7(±V÷
```

Understanding Initialization Vectors

The DES and other symmetrical encryption systems chain blocks of plaintext, and each block provides a kind of feedback to the subsequent block in the chain. In other words, if you change *Marva* to *Darva*, some or all subsequent bits that follow in the message will be affected by this change. In DES, for example, the plaintext is broken up into eight-byte groups, or blocks, which are each then manipulated as individual units. However, what about the *first* block? It has no preceding block to provide the feedback needed to distort it.

An initialization vector provides a fake block that gives simulated feedback to the first real block of plaintext. All you need to remember, though, is not to use something plain and simple as your initialization vector (such as all 1s). Instead, employ something that has a randomness. Also, it's actually not important that you try to hide the IV from intruders. You can send it just as it is to the recipient of your encrypted message (don't do this with your *key*, though—that must remain secret).

In the previous example (Listing 6.4), both the sender and recipient of the encrypted message knew the IV (it can be hard-wired into the encryption/decryption software, for example, as it is in the example). However, you could also permit it to be randomly generated by .NET (simply don't assign an IV to the `CryptoServiceProvider` object, as with `des.IV = Vector` in the example). If you don't assign an IV, one is randomly generated for you. However, if you take this approach, you must prepend or append the IV to the message prior to transmission. Then the recipient software strips off the IV, assigns it to the `CryptoServiceProvider` object, and decrypts the rest of the ciphertext message.

Discovering Key Sizes

You cannot feed in just any size key to the .NET encryption routines. Each has a default size (the largest size that particular algorithm permits), but you can set alternative, shorter key sizes if you must. DES defaults to an 8-byte, but the Rijndael algorithm wants 16-, 24-, or 32-byte keys.

If you need to find out which key sizes are required, just query the `LegalKeySizes` property. You can request `MinSize` (the smallest permitted), the `MaxSize` (largest), or the `SkipSize` (the increment). `SkipSize` tells you of any sizes available between the minimum and maximum sizes. For instance, the `SkipSize` for the Rijndael algorithm is 8 bytes, hence the sizes 16, 24, and 32. Note that the results are returned in bits, not bytes.

Here's how to query key sizes:

```
'create the TripleDES object
Dim des As New TripleDESCryptoServiceProvider()
Dim fd() As KeySizes
fd = des.LegalKeySizes() 'tells you the size(s), in bits
MsgBox(''minsize = "& fd(0).MinSize & Chr(13) & _
"maxsize = " & fd(0).MaxSize & Chr(13) & _
"skipsize = " & fd(0).SkipSize)
```

Run this code and you get: 128, 192, 64. Here are the key sizes for the .NET symmetric algorithms:

| Algorithm | Permitted Key Sizes | Default |
|-----------|-----------------------|---------|
| DES | 64 bits | 64 |
| TripleDES | 128, 192 bits | 192 |
| RC2 | 40–128 bits | 128 |
| Rijndael | 128, 192, or 256 bits | 256 |

Hashing while Encrypting

Encrypting your message keeps Eve the intruder from reading your secrets. But what prevents her from modifying your message? How do you ensure the integrity of your transmission? How do you know that you received what was sent?

Recall that this tampering problem can be solved via hashing, and if you wish you can combine hashing with encryption right in the same streaming operation. Here's an example. If a single character

[Team Fly](#)

 Previous

Next 

"r" in the ciphertext file is changed to "s," in "Marva Johnson writ into the stream.." the resulting deciphered plaintext can look like this:

```
Marv hnsûn writ into the stream..
```

Mangled, but not destroyed. However, other situations demand that you do more than merely eyeball a received message for oddities like this. After all, wire transfers, among other messaging, demand accuracy and authentication. What's more, humans are often simply not involved in the process of receiving encrypted messages, so there would be nobody there to notice the strange word `Marv hnsûn`. Usually an encrypted message isn't actually read by a human; instead, some software receives and processes the transmitted message. Hashing's the answer because it immediately and accurately sets off an alarm if data has been tampered with, no matter how slightly.

Listing 6.5 is a modification of the encrypt/decrypt source code used in a previous example in this chapter that combines the cryptographic process, along with a hashing process, to ensure both the privacy (encryption) as well as the integrity (hashing) of the message.

LISTING 6.5: COMBINING HASHING WITH CRYPTOGRAPHY

```
Private Sub Form1_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles MyBase.Load  
  
    encrypt()  
    decrypt()  
End Sub  
  
Public Sub encrypt()  
  
    'define an 8-byte key  
    Dim k() As Byte = {12, 13, 44, 22, 21, 44, 22, 128}  
    'define an IV  
    Dim Vector() As Byte = {2, 4, 46, 141, 8, 5, 99, 2}  
  
    'Create a file stream  
    Dim fsOUT As New FileStream("c:\testFile.txt", _  
        FileMode.Create, FileAccess.Write)  
  
    'create a byte array holding the plaintext message:  
    Dim barray As Byte() = (New UnicodeEncoding).GetBytes _  
        ("Marva Johnson writ into the stream.")  
  
    '***** GET THE HASH VALUE OF THE PLAINTEXT BYTE ARRAY:  
    Dim sha1 As SHA1 = sha1.Create()  
    Dim hashValue() As Byte = sha1.ComputeHash(barray)  
  
    Dim des As New DESCryptoServiceProvider  
    'stuff the key  
    des.Key = k  
    des.IV = Vector
```

```
Dim desencrypt As ICryptoTransform = des.CreateEncryptor()

'***** SAVE ciphertext and hashvalue
'create a cryptostream, specifying DESencrypt
' as the transform (the encrypting scheme)
Dim cryptostream As New CryptoStream(fsOUT, desencrypt,
CryptoStreamMode.Write)

'save the CIPHERTEXT to the hard drive
cryptostream.Write(barray, 0, barray.Length)

'append the hash value
cryptostream.Write(hashValue, 0, hashValue.Length)
cryptostream.Close()
fsOUT.Close()

End Sub

Public Sub decrypt()

'define an 8-byte key
Dim k() As Byte = {12, 13, 44, 22, 21, 44, 22, 128}

Dim des As New DESCryptoServiceProvider

'define the IV
Dim Vector() As Byte = {2, 4, 46, 141, 8, 5, 99, 2}

'stuff the key and IV
des.Key = k
des.IV = Vector

'NOW BRING THE MESSAGE BACK IN AND DECRYPT IT

'create the input file stream
Dim fs As New FileStream('c:\testFile.txt", FileMode.Open, F

'create a decryptor from the crypto service
' provider (uses same key assigned earlier)
Dim desdecrypt As ICryptoTransform = des.CreateDecryptor()

'create a cryptostream for reading the file in, and decrypt i
Dim cryptostreamDecrypt As New CryptoStream _
(fs, desdecrypt, CryptoStreamMode.Read)

'put the whole message into "m"
```

dark recess of the .NET netherworld). Up in the sunlight where we programmers reside, strings are pretty much always treated just as if they were value types. For example, like all other "objects" passed as parameters in VB.NET, when you write any function, VB.NET fills in *ByVal* automatically by default for every and any item in your function's argument list.

WHAT'S SHARED, WHAT'S STATIC?

Why don't you have to instantiate a Math object before using its methods, such as Abs or Max? The actual answer is: *just because*.

But here's the technical rationale: You've probably seen the term *static* (or *class member*) used with some methods in the documentation for C#, C++, and all the other C family of languages. In VB.NET this same trick is achieved by using the *Shared* command.

Ordinarily, if you want to use a method of an object you have to instantiate the object first. However, methods (or events, properties, or fields) declared as Static (in C languages) or Shared in VB.NET can be "invoked on a class" (translated: *used in your programs*) without your having to actually create an instance of that class. What? How can a property or method be usable without any instance of its object? Isn't that rather similar to the idea of global variables? Global variables so enrage professors of OOP that they've been known to shout "Verboten!" and slap their pointer against the blackboard so hard it breaks.

One example of a Shared method is the Max method. It tells you which number is larger than another. Sure, Max is a method of the Math class, but you don't have to instantiate a Math object in order to use the Max method. No. It's one of those privileged "shared" (AKA static) methods, so you can just use it in your code directly. C languages use the term *instance* to refer to methods (or other members) requiring that you must first instantiate their object before you're allowed to use the functionality of the members. We're figuring out what goes where, aren't we? Isn't clerical work fun?

[Visual Basic] Overloads Public Shared Function Max(Byte, Byte) As Byte
[C#] public static byte Max(byte, byte);

You *can't* instantiate a Math object even if you wanted to. It's Private. This won't work:

```
Dim ma As New System.Math
```

You must instead just directly use the Math.Max method:

```
Dim x As Integer = 2 : Dim y As Integer = 4  
MsgBox(Math.Max(x, y))
```

Shared members are an exception to the OOP rule that you're supposed to instantiate an object before you can actually, you know, *use* it.

Into the Void: While we're on the topic of C terminology, what does the term *void* mean? You see it quite often in the VB.NET documentation, though it's not part of the VB language. If you see `Public Static Void`, don't be alarmed. *Void* is just C#, which means: nothing is returned from this procedure. In other words, the procedure is what we VB programmers would call a Sub (as opposed to a Function). There's no returned value, for example:

```
Public Static Void Delete(string path).
```

```
Dim m As String = New StreamReader _
(cryptostreamDecrypt, New UnicodeEncoding).ReadToEnd()

'now test the hash

' compute the hash value of everything but the hash in ms
Dim sha1 As SHA1 = sha1.Create()
'figure out the bytes needed
Dim hashSize As Integer = sha1.HashSize / 8

'create byte array and fill with the entire
' restored plaintext plus the hash values
Dim sArray As Byte() = (New UnicodeEncoding).GetBytes(m)

'calculate the hash value of the entire message
' (to compare with the transmitted one "messageHashValue")
Dim hashValue() As Byte = sha1.ComputeHash(sArray, 0, sArray.
hashSize)

Dim messageHashValue(hashSize - 1) As Byte
Dim ms(sArray.Length - 1)

'extract the appended hash value from the message array
Array.Copy(sArray, sArray.Length - hashSize, messageHashValue
Array.Copy(sArray, 0, ms, 0, sArray.Length - hashSize)

'compute plaintext
Dim s As String = " Restored Plaintext Message:"
For i As Integer = 0 To ms.Length - 1 Step 2
    s &= Chr(ms(i))
Next i

'compare hashValue and msHashValue

For i As Integer = 0 To hashSize - 1

    If messageHashValue(i) <> hashValue(i) Then
        MsgBox("There has been an intrusion."& _
"The hash values are not identical."& s)
        Exit Sub
    End If

Next
MsgBox("The file has not been tampered with." & s)

End Sub
```

The RSA system (named after its creators, Professors Rivest, Shamir, and Adleman) uses a particularly clever encryption process. RSA uses *two keys*, one of which is made public. This is the first time in the entire history of cryptography that two different keys are used, one for enciphering (the public key) and a different one for deciphering (the private key). Until the RSA system was first suggested in the late seventies, no one had imagined that the key used to decipher could be anything other than the key used to encipher (or at least a version of that key).

For example, it seems essential that if the encipherment process involves moving, say, the third character to the end of the message—then the decipherment process must move the last character to the third position, to restore the plaintext. Another odd quality of the RSA system is that it does not employ either substitution (use x to mean the letter f , for instance) or permutation (switch the third character with the last character, for example). These are the two classic encryption processes. Instead, RSA enciphers using purely mathematical manipulations of the characters.

With the public-key RSA system, everyone knows the key that is used to encipher a message. Everyone on your network usually has access to a list of everyone else's public key. And if an outsider gets hold of this list, no harm done. Public keys do Eve no good. The second key, the one that *deciphers* the message, is kept private, known only to the person receiving the message (it's not even known to the person enciphering the message—all they need is your public key). Hundreds of different people could use your public key to encipher plaintext and send the resulting ciphertext messages to you. Then you use your secret key to decipher all those messages. One implication of this system is that you need not exchange secret keys with any of your communicants. Probably the single greatest weakness of traditional symmetric systems (including DES) is that both the encipherer and decipherer must know the secret key. So how do you transmit this secret key between these two people? You *could* encipher the secret key, but that doesn't solve the problem, just moves things back one step—you'd still have to exchange *another* key to unlock the enciphered first key.

The problem of key transfer is similar to the problem of transmitting the plaintext message: just how do I get the key from me to you?

Managing traditional paired (symmetric) keys introduces a nasty clerical problem, too. How do you provide key pairs for all people in a typical office? Each pair of people must have different keys (otherwise they could all read each other's messages). And if your office is networked, you have to generate many more keys than the simple number of people on the network. Because each communicating pair requires its own unique key, if you have 150 people you have to provide 11,175 unique keys. There are that many possible pairs of communicants in a network of 150 people. Obviously this isn't practical.

Several solutions have been developed to deal with this problem. One solution employs a key distribution center in which a DES key for each person is saved on the network in a central, secure repository. When Alice wants to send a ciphered message to Bob, a temporary DES session key is generated, then the temporary key is itself enciphered using Alice's stored key and the result is sent to Alice. Likewise, the temporary key is enciphered using Bob's stored key and the result sent to Bob. When each temporary key arrives at its destinations, it is

deciphered by both Alice and Bob. Now both communicants have the same key, which is then used to encipher and decipher the message.

If you encrypt multiple messages using the same key, you weaken the encryption system. Using a key distribution center makes it easy to generate a temporary key for each new message, and also avoids giving an Eve cryptanalyst the advantage of having several messages enciphered with a single key. Another obvious benefit of dynamic key generation is that even if an intruder were to somehow

get hold of a key, this still wouldn't lead to a major breach in security. Only a single message or session is compromised. These temporary keys are sometimes called *session keys* because they are used for a single communication session, then discarded.

But the best solution to the key problem is the *public key* method, because you neither have to transmit a secret key between two parties nor generate a unique key for each communication. If your network has 150 people, for example, you need to generate only 300 keys, one public key and one private key for each member of the network.

The public and private keys work together to unlock a message, like the way you open your safe deposit box by inserting two keys: you insert your key and the teller inserts the bank's key.

Public key encryption is deep and strong. The plaintext is quite gone. The RSA public key system works because some kinds of math operations are very easily accomplished in one direction, but functionally impossible in the other direction.

Let's quickly generate an actual public key using the RSA algorithm (which produces an XML string containing the key):

```
Private Sub Form1_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles MyBase.Load  
  
    Dim rsa As RSACryptoServiceProvider = New RSACryptoServiceProvider  
  
    Dim publicKeyOnly As String = rsa.ToXmlString(False)  
  
    Console.WriteLine(publicKeyOnly)  
End Sub
```

The result I get is this:

```
<RSAKeyValue>  
<Modulus>qJ9CevDiMmPsMHzkb1AmWc5s0j/+Zsv+mrMZskmJ/VX19b/Tgb96mR0tm5moSeChY8ISF2  
5fXvE4qax9OJaC8fbZfgf8WA9dPTm6J6CuYR1Hb03QD5uFV/ATZ2T8SLu0XkzwThuS2PngyojKIm+AQ  
23t7bfmMEkwtvs=  
</Modulus>  
<Exponent>AQAB</Exponent>  
</RSAKeyValue>
```

Your result will differ. The key is randomly generated. The RSA system permits keys to be anywhere between 384 to 16,384 bits in 8-bit increments. However, the default key size is 1,024 bits.

How RSA Works

RSA depends on a math operation that is in a category called *trap doors*. They are named after

the trick doors in the floor of a theatre stage through which Hamlet's ghost, for example, can slowly rise up through the fog. To the actor below stage waiting for his entrance, the elevator lift and the hinges and sliders make it quite obvious where the trapdoor is in the stage floor. However, to people onstage, the door is barely visible—it blends into the floor around it.

The idea of a trap door mathematical process is that some things can be easily accomplished in one direction, but are very difficult, if not practically impossible, to reverse. It's easy, for example, to

run a tree through a chipper and reduce it to sawdust. It's impossible to restore the tree from that pile of sawdust and chips. Trap door processes are also called *one-way functions*.

RSA employs the trap door involved when two very large prime numbers are multiplied. When you multiply two large prime numbers, it is very hard to figure out which primes were multiplied if all you have is the result of the multiplication.

When the numbers involved are small, of course, it's not hard at all to figure out which two primes were multiplied. Consider the number 15. It's pretty easy to figure out that 3 and 5 are the primes multiplied to get 15. But with large primes it's essentially impossible to figure out the factors.

The system works because when you multiply one prime by another prime, the resulting number cannot be produced by multiplying any *other* pair of primes. Therefore, there is only one possible pair of primes that, when multiplied, can produce this particular result. It is not practical—takes way too much time—to factor the result of multiplied primes to figure out which pair of prime numbers were multiplied to produce that result. The public key is the result of this multiplication of large primes. Only one person, the recipient of the message, knows the correct private key, and that private key is the two primes that were multiplied to produce the public key.

***NOTE** Theoretically, someone brilliant or lucky could figure out an algorithm that would factorize the public key into its prime factors with reasonable efficiency (in other words, would get the private key prime factors before the universe ends). So far, however, nobody has. Estimates are rough, of course, but current brute force attempts to crack 1,024-bit long RSA private keys require around 90 million Mips-years. Mips means a million processor instructions per second. Imagine what a Mips-year must represent. Perhaps if quantum computing is achieved, or some new Fermat works out a mathematical solution to the factorizing problem, we'll have to come up with a better system than RSA. For now, it works.*

The code in Listing 6.6 encrypts a message using RSA, then decrypts it.

LISTING 6.6: ENCRYPTION AND DECRYPTION USING THE PUBLIC KEY SYSTEM

```
Imports System.Security.Cryptography
Imports System.Text

Dim xmlKeys As String 'holds the public and private keys
Dim xmlPublicKey As String 'holds the public key only

'holds plaintext, then the encrypted version (in the encryptRSA f
Dim plainTextinBytes As Byte()

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

    'generate public and private keys
    Dim rsa As New RSACryptoServiceProvider

    'save both public and private keys
```



```
' in global variable for use in decryption procedure
xmlKeys = rsa.ToXmlString(True)

'save only public key for use in encryption procedure
xmlPublicKey = rsa.ToXmlString(False)

encryptRSA()
decryptRSA()

End Sub

Private Sub encryptRSA()

    Dim rsa As New RSACryptoServiceProvider

    'import public key:
    rsa.FromXmlString(xmlPublicKey)

    Dim message As String = "'Drastic weather changes experienced

    'turn message into a byte array:
    plainTextinBytes = (New UnicodeEncoding).GetBytes(message)

    'PlainTextinBytes.Length = 86 bytes going into the encryptior
    Try
        ' Encrypt
        plainTextinBytes = rsa.Encrypt(plainTextinBytes, False)
    Catch e As CryptographicException
        MsgBox(e.ToString)
    End Try

    'PlainTextinBytes.Length = 128 bytes after encryption

    'see it enciphered:
    For i As Integer = 0 To plainTextinBytes.Length - 1
        Console.Write(Chr(plainTextinBytes(i)))
    Next i

    Console.WriteLine()

End Sub

Private Sub decryptRSA()
```

```
Dim rsa As New RSACryptoServiceProvider

'import the keys from the XML string global variable
'this recreates an identical RSA object to
' the one used to create the keys (in the Form_Load event abc
rsa.FromXmlString(xmlKeys)

Dim decryption As Byte() = rsa.Decrypt(plainTextinBytes, False)

'see it deciphered:
For i As Integer = 0 To (decryption.Length - 1) Step 2
    Console.Write(Chr(decryption(i)))
Next i

End Sub
```

Here's how this example works. To simulate the relationship between the encryptor and decryptor, I set up two global variables to hold the keys. The decryptor (the person who receives the RSAencrypted ciphertext message) needs to know both the public and private keys. The encryptor knows only the public key.

An RSACryptoServiceProvider object can either generate or import just the public key, or the public/private key pair. It creates a public key when you execute the ToXMLString method with its argument set to False. It creates a public/private key pair when that argument is set to True. When you execute the FromXMLString method, you cause an RSACryptoServiceProvider object to make *a clone of the RSACryptoServiceProvider object that originated the keys*.

Recall that when using DES or other symmetrical algorithms, you can specify the key by simply making up a value, translating it into a byte array, then assigning that array to the Key property of the DES object. However, with the RSA system, you must do a little more key management. First, the decryptor (the person *receiving* the ciphertext) generates a public/private key pair using the ToXmlString method. This pair works together, and only these two values will work together. When generated, the keys are in a string in XML format. The public key of this pair is, you guessed it, made public. It's perhaps listed in a file or otherwise published. The encryptor (the person sending the message) gets a copy of the decryptor's public key. The encryptor then imports this public key into the RSA object, thereby permitting the RSA object to correctly encrypt the message.

Encrypting and Decrypting using RSA

Here are the steps taken in Listing 6.6 to "pass" the public key to the encryptor (message sender), retain the public/private key pair by the decryptor (message receiver), encrypt, and finally decrypt the message:

1. The decryptor (recipient) generates a public/private key pair (put into an XML string) with this code:

```
Dim rsa As New RSACryptoServiceProvider  
xmlKeys = rsa.ToXmlString(True)
```

[Team Fly](#)

 Previous

Next 

- The decryptor saves this XML string (`xmlKeys`) for later use when the ciphertext is received.
- The decryptor generates a public-key-only XML string using this code:

```
xmlPublicKey = rsa.ToXmlString(False)
```

- The decryptor sends the public key to the encryptor. The encryptor needs only the public key to accomplish the encryption. The encryptor assigns (imports) the public key (an XML string) to the encryptor's RSA object, using the `FromXmlString` method:

```
Dim rsa As New RSACryptoServiceProvider  
  
'import public key:  
rsa.FromXmlString(xmlPublicKey)
```

- The encryptor now uses the public key to translate a byte array containing the plaintext message into a byte array holding the encrypted message:

```
plaintextinBytes = rsa.Encrypt(plaintextinBytes, False)
```

The `False` argument when you use this `Encrypt` method specifies which padding algorithm you want to use (padding is necessary because the RSA algorithm wants specific-sized blocks to work with—so random numbers are generated by the `RSACryptoServiceProvider` object when you use the `Encrypt` method). A more recent, but some say compromised, padding technique, OAEP, is used if you set the Boolean flag to `True`. I set it to `False` to use the older but tried-and-true PKCS#1 v1.5 (Public Key Cryptography Standards) padding. Whichever padding you choose, be sure that you use it for both the encryption and decryption (set the flag argument the same way). Note that OAEP only works under XP.

- The encrypted byte array is sent to the decryptor (recipient).
- The decryptor creates a new `RSACryptoServiceProvider` object, but ensures that it's a clone of the one that generated the public/private key pair. This cloning takes place if the public/private keys held in the XML string are fed to the new `RSACryptoServiceProvider` object using the `FromXMLString` method:

```
Dim rsa As New RSACryptoServiceProvider  
  
rsa.FromXmlString(xmlKeys)
```

- Now the decryption can take place:

```
Dim decryption As Byte() = rsa.Decrypt(plaintextinBytes, False)
```

And you've got the original plaintext restored in a byte array.

In the example code, the ciphertext is displayed in the output window (it blows up to 128 bytes after the padding). Then after the decryption, the restored plaintext is sent to the output window. To simulate the transmission of the public key from the decryptor to the encryptor—and the transmission of the ciphertext from the encryptor to the decryptor—I just use a couple of global variables in this example code. Also, I persist the public/private key needed by the decryptor in another global variable.

[Team Fly](#)

 Previous

Next 

When you run across strings and objects behaving strangely, just relax. The string and object types—I'm not talking here about *every* kind of object, but rather the actual *object type*, as in: `If Object.ReferenceEquals(a, b) Then`—*are special*. They are exceptions to the "reference type" rules. They are treated differently by .NET. They are called "built-in reference types" and this means they are supposed to behave more like the good old built-in integers and booleans and other familiar data types. (But don't be fooled; the *built-in object* type is a pretty odd bird.) Speaking of odd birds, consider the Shared member, described in the sidebar titled "What's Shared, What's Static."

GetHashCode

`GetHashCode` returns a number unique to the object. Think of it as a kind of GUID, a computergenerated number intended to uniquely identify a particular object. The .NET hashcode for the string "Helen," for example, is always 222703087, but the hashcode for "Helex" is 222703097. Store "Helen," however, in two separate string variables and you get the same HashCode (as you should). Some books claim that different objects containing the same value result in different hashcodes. This is incorrect; *sometimes* you get the same hashcode, sometimes not.

Hashcodes are used two ways in .NET. A simple hashcode such as the one generated for Helen can be used as a unique index number to check for duplicate values when an object is added to a collection or hash table. This is useful to prevent duplicate entries, which are forbidden in a primary key field, for example. Unfortunately, though, the `GetHashCode` method as implemented in `System.Object` is relatively useless because unpredictable.

In general, inheriting classes override this `GetHashCode` method, and also override the `Equals` method at the same time. Not only can they thereby provide more useful hashcodes, but can simultaneously ensure that objects that are equals also return the same hashcode.

A more sophisticated hashing algorithm is used in the .NET Security assembly, and it can be used quite effectively to generate unique keys from text passwords. This use of hashing is explored in Chapter 6.

GetType

The `GetType` method returns a data or object type. .NET maintains *metadata* (information about information, like the signs in a bookstore categorizing the books: Philosophy, Cooking, and so on). In a .NET assembly, each object is stored with a description of its nature and relationships. `GetType` can be used in your programming if you need to figure out the type of an object:

```
Dim xn As Type = s.GetType
```

For additional information on how to use `GetType` in reflection—a technology that takes advantage of metadata—see Chapter 8.

ToString

The final fundamental method inherited from `System.Object` is `ToString`, and you know what that does. Often overridden, `ToString` is also automatically invoked behind the scenes when you use common commands such as `Console.WriteLine` or `MsgBox`.

`ToString` transforms some objects' names or qualifications into strings so they can be seen in message boxes or in the output window or otherwise viewed. I say *some* objects because in certain situations, `ToString` is optional. Read on.

printing process, you call the `PrintDocument` object's `Print` method, which doesn't produce any output but raises the `PrintPage` event. This is where you must insert the code that generates output for the printer. You can use any drawing methods of the `System.Drawing` class to generate the graphics elements on the page. The current page is printed when the `PrintPage` event handler terminates. The following statement initiates the printing; it's usually placed in a button's or a menu's `Click` event handler:

```
PrintDocument1.Print
```

To experiment with simple printouts, create a new project, place a button on the form, and add an instance of the `PrintDocument` object to the project. Then enter the previous statement in the button's `Click` event handler. After calling the `Print` method, the `PrintDocument1_PrintPage` event handler takes over. The signature of the `PrintPage` event handler is shown next:

```
PrintPage(ByVal sender As Object, _  
          ByVal e As System.Drawing.Printing.PrintPageEventArgs) _  
          Handles PrintDocument1.PrintPage
```

The second argument exposes all the properties you need to access the printer. The `Graphics` property represents the printer's page, which is where your output will be sent. The `MarginBounds` property contains information about the printable area of the page (basically, the user-specified margins, which you must take into consideration in your printing code) and the `PageSettings` property contains information about the page you're printing on (the size of the page, its orientation, margins, and so on). The properties of the `PageSettings` object include the `PrinterSettings`, which is another object that contains information about the printer—the settings specified by the user on the `Printer Setup` dialog box. You'll see how to use these properties, and how to display the corresponding dialog boxes, shortly. But first, let's generate a simple printout. Start a new project, place a button on the form, and then drop an instance of the `PrintDocument` control on the form. In the button's `Click` event handler, enter the following statements:

```
Private Sub Button1_Click(ByVal sender As System.Object, _  
                          ByVal e As System.EventArgs) _  
                          Handles Button1.Click  
    PrintDocument1.Print()  
End Sub
```

The code that will generate the printout must reside in the `PrintDocument` object's `PrintPage` event handler. The statements shown in Listing 7.1 will print a rectangle that encloses the printable area of the page and a short string within the rectangle. Insert them in the `PrintPage` event handler, then run the application and click the button to generate the printout.

LISTING 7.1: A VERY SIMPLE PRINTOUT

```
Private Sub PrintDocument1_PrintPage(ByVal sender As Object, _  
                                     ByVal e As System.Drawing.Printing.PrintPageEventArgs) _  
                                     Handles PrintDocument1.PrintPage  
    Dim G As Graphics = e.Graphics  
    Dim X, Y, W, H As Integer  
    X = e.MarginBounds.X
```

Y = e.MarginBounds.Y

[Team Fly](#)

 Previous

Next 

```
W = e.MarginBounds.Width
H = e.MarginBounds.Height
G.DrawRectangle(Pens.Blue, New Rectangle(X, Y, W, H))
Dim prnFont As New Font("Comic Sans MS", 36, FontStyle.Regular)
G.DrawString("Printing with VB.NET", prnFont, _
    Brushes.Green, 150, 300)
G.DrawString("Sample Printout", _
    New Font("Verdana", 16, FontStyle.Regular), _
    Brushes.Gray, 10, 10)
```

End Sub

The `PrintPage` event handler shown in the listing produces the page shown in Figure 7.1. The first statement creates a `Graphics` variable and stores there the `Graphics` object that represents the printing surface. The following few statements extract the origin and the dimensions of the printable area of the page. These settings are retrieved from the `MarginBounds` property of the `PrintPageEventArgs` argument of the event handler. The default margins are one inch on every side; you'll see shortly how you can allow users to specify different margins with the `Page Setup` dialog box.

The `DrawRectangle` method draws the rectangle that encloses the printable area of the page. The dimensions of the rectangle are specified in the default units of the printer's `Graphics` object, which are hundredths of an inch. The `DrawString` method draws a string with the specified brush and the specified font. The first call to the `DrawString` method prints a string within the page's margins, while the second call to the same method prints a string outside the margins. You can draw anywhere on the `Graphics` object, regardless of the margins. It's your responsibility to impose the margins and make sure that your graphics elements are limited within the area of the `MarginBounds` property. The syntax of the drawing methods is the same, whether you're drawing on a `PictureBox` control or printing on a page.





FIGURE 7.1 The output of the sample code, shown at 50% of its actual size

PaperSize Returns a PaperSize object that represents the size of the paper in the selected bin. The paper's dimensions are the same as the ones returned by the Bounds property. In addition to the dimensions of the paper, the PaperSize object exposes two more interesting properties, the Kind and PaperName properties. The Kind property returns a member of the PaperKind enumeration (Letter, B5 Envelope, etc.), while the PaperName property sets or returns the name of the paper for custom sizes.

PaperSource Returns a PaperSource object that represents the currently selected tray on the printer. For printers with a single tray, the PaperSourceName property of the PaperSource object is "Auto Sheet Feeder."

PrinterResolution Returns a PrinterResolution object that represents the printer's resolution. The X and Y properties of the PrinterResolution object return the current horizontal and vertical resolutions respectively in dots per inch. The Kind property of the PrinterResolution object exposes in turn several properties, including the Low, Medium, High, Custom, and Draft properties.

PrinterSettings Returns a PrinterSettings object that represents the properties of the printer. Use this property to read the properties set by the user on the Printer Setup dialog box, or set the same properties from within your code. The properties of the PrinterSettings object are described in the following section.

The PrinterSettings Object

The PrinterSettings object exposes the following properties, which you can use to retrieve the properties of the current printer.

InstalledPrinters A method that retrieves the names of all printers installed on the computer, as well as the names of any remote printers to which the computer has access. The same printer names also appear in the Print dialog box, where the user can select one of the available printers.

CanDuplex A read-only property that returns a True/False value indicating whether the printer supports double-sided printing. This feature won't affect your printing code; you'll print the pages as usual and the printer will print them on the appropriate side of each page.

Collate Another read-only property that returns a True/False value indicating whether the printout should be collated or not. This setting entails no changes in your code.

Copies A numeric property that returns, or sets, the requested number of copies of the printout.

DefaultPageSettings The PageSettings object that returns, or sets, the default page settings for the current printer. We usually assign the PageSettings property of the Page Setup dialog box to the DefaultPageSettings property of the PrintDocument object.

Duplex The property that returns or sets the current setting for double-sided printing.

FromPage,ToPage The printout's starting and ending pages, as specified in the Print dialog box by the user.

IsDefaultPrinter Returns a True/False value indicating whether the selected printer (the one identified by the PrinterName property) is the default printer. Note that selecting a printer other than the default one in the Print dialog box doesn't change the default printer.

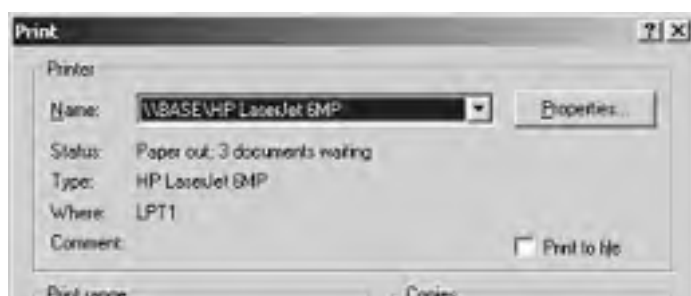


FIGURE 7.2 The Page Setup dialog box

To display this dialog box, drop the PageSetupDialog control on the form and then call its ShowDialog method. Before showing this dialog box, you must set the PageSettings property. You can create a new PageSettings object, set its properties, and then assign this object to the PageSettings property of the PageSetupDialog control. We usually assign to this property the DefaultPageSettings property of the PrintDocument object. After the user closes the Page Setup dialog box with the OK button, assign the control's PageSettings object to the DefaultPageSettings object of the PrintDocument object, to make the user-specified settings available to our code. Here's how we usually display the dialog box from within our application and retrieve its PageSettings property:

```
With PageSetupDialog1
    .PageSettings = PrintDocument1.DefaultPageSettings
    If .ShowDialog().DialogResult = OK Then
        PrintDocument1.DefaultPageSettings = .PageSettings
End With
```

The PrintDialog control displays the standard Print dialog box, shown in Figure 7.3, which allows users to select a printer and set its properties. If you skip this dialog box, the output will be sent automatically to the default printer and the default settings of the printer will be used.



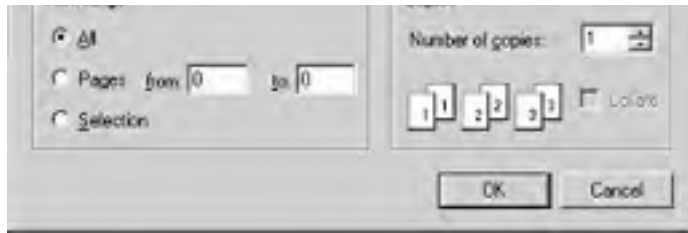


FIGURE 7.3 The Print dialog box

The printer selected on this dialog box automatically becomes the active printer; you don't have to insert any code to switch between printers. In addition to listing the available printers, this dialog box allows the user to specify the range of pages to be printed and the number of copies. Skipping a number of pages is straightforward—you simply perform all the calculations, but you skip the statements that actually print on the page. This means that printing just the last page of a document will take as long as printing the entire document. If your report is made up of a fixed number of rows per page, you can easily skip a number of pages by ignoring the number of rows that will fit in the first so many pages (e.g., if you're printing 25 rows per page, you can skip the first three pages by simply ignoring the first 75 rows).

Some of the options on the Print dialog box are not enabled by default. In the Print Range zone section, only the All option is enabled. To allow the user to print the currently selected section of the document, set the control's AllowSelection property to True. Likewise, to enable the Selection option, set the control's AllowSomePages property to True. Another interesting property of the PrintDialog control is the ShowNetwork property, which determines whether the dialog box allows the user to select a non-local printer (a printer connected to a different computer on the network). The following statements display both the Print and Page Setup dialog boxes. All print range options on the control are enabled, and the Page Setup dialog box is displayed only if the Print dialog box is closed with the OK button. If the Page Setup button is also closed with the OK button, the program starts printing by calling the Print method of the PrintDocument object.

```
PrintDialog1.PrinterSettings = _
    PrintDocument1.DefaultPageSettings.PrinterSettings
PrintDialog1.AllowSelection = True
PrintDialog1.AllowSomePages = True
If PrintDialog1.ShowDialog = DialogResult.OK Then
    PageSetupDialog1.PageSettings = PrintDocument1.DefaultPageSettings
    If PageSetupDialog1.ShowDialog = DialogResult.OK Then
        PrintDocument1.DefaultPageSettings = _
            PageSetupDialog1.PageSettings
        PrintDocument1.Print
    End If
End If
```

To summarize, before displaying the Print dialog box, you must set the PrinterSettings property. Unless you want to create a new PrinterSettings object, you will assign the DefaultPageSettings.PrinterSettings property of the PrintDocument object to the PrinterSettings property of the PrintDialog control. Likewise, before showing the Page Setup dialog box, you must set its PageSettings property to the DefaultPageSettings of the PrintDocument object. When the Page Setup dialog box is closed, we retrieve the settings specified by the user on this control and assign the control's PageSettings property to the PrintDocument object's DefaultPageSettings property.

The third of the printing controls is the PrintPreview control. The Print Preview dialog box displays a preview of the printed document. This dialog box exposes a lot of functionality and allows users to examine the output, and, when they're happy with it, they can send it to the printer. The PrintPreview dialog box, shown in Figure 7.4, is made up of a preview pane, in

which you can display one or more pages at the same time at various magnifications, and a toolbar. The buttons on the toolbar allow you to select the magnification, set the number of pages that will be displayed on the preview pane, move to any page of a multi-page printout, and send the preview document to the printer.

[Team Ely](#)

 Previous

Next 

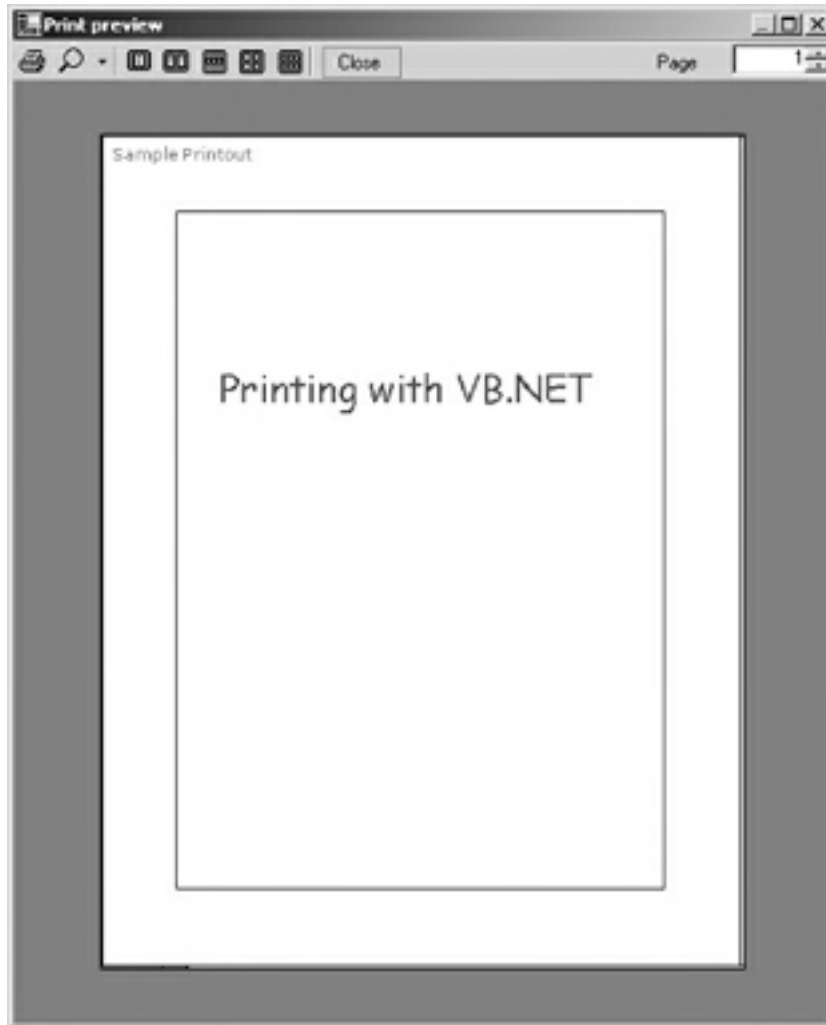


FIGURE 7.4 The Print Preview dialog box

Once you've written the code to generate the printout, you can easily direct it to the PrintPreview control. You don't have to write any additional code; just place an instance of the control on the form and set its Document property to the PrintDocument control on the form. Then call the control's ShowDialog method, instead of the PrintDocument object's Print method:

```
PrintPreviewDialog1.Document = PrintDocument1  
PrintPreviewDialog1.ShowDialog
```

After the execution of these two lines, the PrintDocument object takes over. It fires the PrintPage event as usual, but it sends its output to the preview dialog box, and not to the printer. The dialog box contains a Print button, which the user can click to send the document being previewed to the printer. The exact same code that generated the preview document will also print the same document on the printer.

The PrintPreview control will save you a lot of paper and toner while you're testing your printing code, because you don't have to actually print every page to see what it looks like.

The same PrintPage event handler of the PrintDocument object will generate both the actual printout and the document preview. In other words, you don't have to duplicate code. If the user is satisfied with the appearance of the printout, they can click the Print button at the top of the PrintPreview control to send the document to the printer. The Print Preview option adds a professional touch to your application; there's no reason why you shouldn't add this feature to your projects. In the examples in this section, we'll use this control to display the printouts on the screen.

shortly, and we'll review the basic methods for printing strings in the following section. The simplest form of the DrawString method accepts as arguments the string to be printed, a Font and a Brush object that determine how the string will be rendered on the Graphics object, and the coordinates of the string's upper-left corner:

```
Graphics.DrawString(str, tFont, tBrush, X, Y)
```

Some overloaded forms of the DrawString accept an additional argument, which is a System.Drawing.StringFormat object. The StringFormat object sets the alignment, rotation, and a few more properties that determine the appearance of the string. To print a string at a specific location on the page and specify a different alignment, use the following statements:

```
Dim strFormat As New System.Drawing.StringFormat()  
strFormat.Alignment = StringAlignment.Far  
e.Graphics.DrawString(str, pFont, pBrush, _  
New RectangleF(100, 200, 150, 50), strFormat)  
e.Graphics.DrawRectangle(pPen, New Rectangle(100, 200, 150, 50))
```

Change the Alignment property of the strFormat variable to print the same string in the same rectangle with different alignments. The last statement prints a rectangle that outlines the area in which the string will appear.

If the rectangle specified in the DrawString method isn't tall enough for the entire string, then the string will be printed partially, as shown in the last box of the first column in Figure 7.5. Notice that only part of the string is printed (and you can't tell how many lines of text are missing). To prevent a line from being partially printed, set the FormatFlags property of the StringFormat object to StringFormatFlags.LineLimit, as shown in the following statement:

```
strFormat.FormatFlags = StringFormatFlags.LineLimit
```

This flag will cause the DrawString method to print only as many lines of text as can fit in their entirety in the specified rectangle. We'll use this property to prevent the partial printing of the last line on a page.

The StringFormat property has a few more interesting settings. The Alignment property determines the horizontal alignment of the string and its settings are the members of the StringAlignment enumeration: Far (right aligned), Center, and Near (left aligned). The LineAlignment property determines the vertical alignment of the string, and its settings are also the members of the StringAlignment enumeration. The member Far causes the string to be aligned at the top of the corresponding rectangle, while the Near member causes the string to be aligned at the bottom of the corresponding rectangle.

To fully control the appearance of the text on the page, you need to know the arrangement of the text in a rectangle. If you're printing plain text, this rectangle is the entire page, minus the margins. If you're printing a tabular report, this rectangle is the cell in which the text must fit. In other words, you need to know how the DrawString will break the specified string into multiple text lines. We usually know the width of the rectangle in which the text will appear, but we need to calculate the height of the rectangle. The MeasureString method accepts the same arguments as the DrawString method, but it doesn't print anything. Instead, it reports the

number of characters that will fit in the specified rectangle and the number of text lines that will fit vertically in the rectangle. The simplest syntax of the MeasureString method is shown next:

```
Public Function MeasureString(str, fnt) As SizeF
```

[Team Fly](#)

 Previous

Next 

Strong Typing Weakens

For a decade, many VB programmers used a quick debugging technique of printing variables directly on a form to see their values. This convenience is missing in VB.NET (you can't print on a form with a simple Print command). Instead, to see variables you have to use either a `MessageBox` or `Console.WriteLine` to see results in the Output window. This has resulted in some modifications to the strict rules governing the message box.

People were getting a little annoyed at having to type `MessageBox.Show (x.ToString)` each time they were debugging and wanted to see this variable `x`. So `MessageBox` was shortened to `MsgBox`, the `Show` was omitted, and, after a while, even the `ToString` method was dropped. Now you can use the shorthand version: `MsgBox (x)`.

As you see, VB.NET is gradually abandoning some of the more onerous rules demanded by a strict adherence to what they call *strong typing*.

Is Color a Data Type?

Yes. Each time you work in an area of .NET that you've not previously dealt with (or that you've forgotten), you'll nearly always find yourself butting up against the problem of syntax and data typing. You see "cannot be converted" or "reference not set" or a handful of other error messages over and over.

THE VARIOUS MEANINGS OF *STRONG TYPING*

The phrase *strong typing* is used in several different ways in computer programming literature. It sometimes means using descriptive strings as the keys in a collection, rather than using index numbers.

In other places, you'll see *strong typing* described as the enforcement of a rule that each element in a collection be of a specific data type, not a generic "object" type..NET Framework collections in fact *do* include only object types, on the lowest level. And you can add anything to your collections, which is a useful freedom. But you are urged by OOP professors to "strongly type your collections." Don't just add objects and only objects. They point out that it prevents someone ignorant of your collection's purpose from adding, say, an automobile type to a collection of fresh fruit types (which could cause errors). However, it seems to me that this is actually a non-problem if you simply follow the usual OOP practice of always validating incoming data. Still other OOP experts say that a fundamental virtue of strong typing is that type mismatch errors are trapped during compilation rather than later at runtime. This claim ignores the fact that the only time some *outsider* is submitting objects to your class *is* during runtime.

Here's yet another, similar meaning of strong typing in the literature. It means that if you always be sure to declare your data types, you avoid errors in VB.NET. In earlier versions of VB you could leave the type ambiguous by using the `Variant`, or by implicit declaration (you never actually declare the variable's type, you merely use it in code by assigning a value to it—VB then interprets what kind of number it is, or if it's a string). Now, in .NET, you cannot use `Variants`, nor can you implicitly declare (you can turn `Option Explicit` off if you wish).

Nonetheless, you can be ambiguous by declaring a variable as an object type (and since everything is an object, this is similar to declaring a Variant). However, you're urged not to do this for three main reasons: to permit IntelliSense lists, to allow type checking during compilation (so certain kinds of incorrect data type usage are prevented), and to speed up compilation.

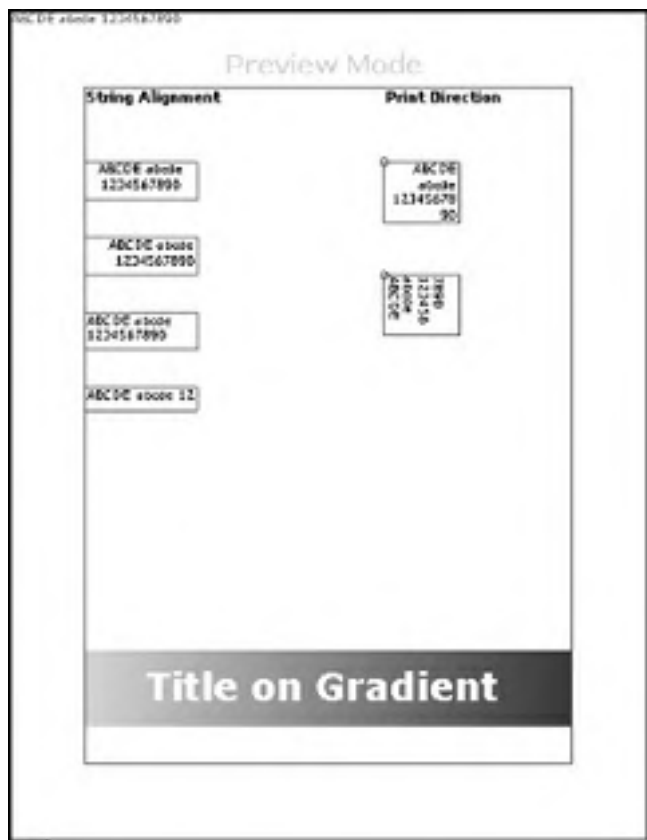


FIGURE 7.5 The output of the PrintTests project in preview mode

The two arguments are a string and the font in which it will be rendered. The `MeasureString` method returns a `SizeF` object with the horizontal and vertical dimensions of the rectangle that encloses the printed string. This form of the method doesn't break the string into multiple lines, unless the string contains newline characters.

There are many more overloaded forms of the method. The following is the most commonly used one:

```
Public Function MeasureString(str, fnt, lRect, _  
                             sFormat, chars, lines) As SizeF
```

The `lRect` argument is a `SizeF` structure that specifies the rectangle in which the string must fit and the `sFormat` argument is a `StringFormat` object that determines how the string will be printed. The method's return value is the rectangle that encloses the string when it will be rendered in the specified font. We usually set the `lRect` argument to the desired width and a

large height, and the method returns the exact number of lines of text needed to fit the text in the rectangle's width. The last two arguments are reference arguments, and they return the number of characters and the number of lines that fit into the specified rectangle. This overloaded form of the method is used to fill a page with text. We pass the entire string to be printed and the dimensions of the printable area to the method as arguments. The `MeasureString` method returns the number of characters that will fit on the page. Then we remove these characters from the beginning of the string and we continue with the next page. The process is actually a little more complex than this; we'll describe it in detail in the section "Printing Plain Text," later in this chapter.

THE PRINTTESTS PROJECT

In this section we'll exercise the basic methods for printing text on a page. The `PrintTests` project is a single form with a button that generates the page shown in Figure 7.5. The project's code prints a

short string with different alignments and orientations. It also prints a string that indicates whether the printout is displayed in a preview pane or has been sent to the printer (it's useful sometimes to know whether the user is previewing the printout or actually printing it). At the bottom of the page, it prints another string on a background filled with a gradient.

The following statements print a string centered in a rectangle with dimensions (150×50). The first rectangle on the page was generated with the following statements:

```
Dim strFormat As New System.Drawing.StringFormat()  
strFormat.Alignment = StringAlignment.Center  
e.Graphics.DrawRectangle(pPen, New Rectangle(100, 200, 150, 50))  
e.Graphics.DrawString(str, pFont, pBrush, _  
    New RectangleF(100, 200, 150, 50), strFormat)
```

The second and third rectangles in the first column are printed with similar statements; only the setting of the Alignment property of the `strFormat` object changes. The last rectangle shows what will happen if the rectangle isn't large enough for the string you want to print in it. The code makes use of the LineLimit property, as shown in the following statements:

```
strFormat.Alignment = StringAlignment.Center  
' Comment out this statement to see how it affects the printout  
strFormat.FormatFlags = StringFormatFlags.LineLimit  
e.Graphics.DrawRectangle(pPen, New Rectangle(100, 500, 150, 30))  
e.Graphics.DrawString(str, pFont, pBrush, _  
    New RectangleF(100, 500, 150, 30), strFormat)
```

The problem with the rectangles in the first column is that the rectangle doesn't exactly enclose the string. The rectangle's height is set to a rather arbitrary value. If we calculate the height of the rectangle, we'll be able to draw a rectangle that exactly encloses the text. Notice that this isn't necessary, because we usually don't draw the enclosing rectangle around strings, unless we print tabular reports made up of rows and columns.

To print the string's enclosing rectangle, we must first specify the width of the rectangle and then calculate the height of the rectangle by calling the `MeasureString` method. The `MeasureString` method will fit the text in the specified rectangle and will return the number of lines of text. Each line's height depends on the font in which the text will be rendered and is given by the `GetHeight` method of the `Font` object. The first of the following statements calls the `MeasureString` method, passing as arguments a `Font` object and a `Size` object representing the dimensions of the rectangle in which the text should fit. The height of the rectangle is set to a large value. The `MeasureString` method will set the argument's `chars` and `lines` to the number of characters and number of text lines that will fit in the rectangle. Then we use this value to find out the exact height of the rectangle in which the text will fit:

```
e.Graphics.MeasureString(str, pFont, _  
    New SizeF(500, 100), strFormat, chars, lines)  
Dim strHeight As Integer = pFont.GetHeight(e.Graphics) * lines
```

The `height` variable is the height of a rectangle with the specified width (500 units, which is 5 inches, in the case of the example) that exactly encloses the string. The string to be printed is broken

into multiple text lines automatically. The next step is to print the string in the specified area and then draw the rectangle that encloses the text:

```
e.Graphics.DrawRectangle(pPen, New Rectangle(500, 200, strHeight, 100))
e.Graphics.DrawString(str, pFont, pBrush, _
    New RectangleF(500, 200, strHeight, 100), strFormat)
```

The `strFormat` argument is a `StringFormat` object that gives you more control over the appearance of exactly how the string will be rendered. The `Alignment` property of the `StringFormat` object specifies the alignment of the text, and its value can be one of the members of the `StringAlignment` enumeration: `Center`, `Far`, and `Near`. The most interesting property of the `StringFormat` object is the `FormatFlags` property, which is also an object; it exposes (among others) the following properties: `DirectionRightToLeft`, `DirectionVertical`, `NoWrap`, and `LineLimit`. The first two properties determine the horizontal and vertical direction of the text. They're both Boolean values, and their default value is `False`. The `NoWrap` property disables the wrapping of text when set to `True`. The `LineLimit` property should be set to `True` if you don't want partial lines to be printed when the text doesn't fit entirely in the specified rectangle.

The first boxed string in the second column is printed in a rectangle with a width of 100 units (1 inch) and the necessary height. The rectangle's height is calculated with the help of the `MeasureString` method and it exactly encloses the string. Here are the statements that print the top boxed string in the second column.

```
strFormat = New System.Drawing.StringFormat()
strFormat.FormatFlags = StringFormatFlags.DirectionRightToLeft
e.Graphics.MeasureString(str, pFont, New SizeF(100, 800), _
    strFormat, chars, lines)
Dim strHeight As Integer = pFont.GetHeight(e.Graphics) * lines
e.Graphics.DrawRectangle(pPen, New Rectangle(500, 200, 100, strHeight))
e.Graphics.DrawString(str, pFont, pBrush, _
    New RectangleF(500, 200, 100, strHeight), _
    strFormat)
e.Graphics.DrawEllipse(New Pen(Color.Green, 3), _
    New Rectangle(495, 195, 10, 10))
```

You can achieve the same effect by setting the `Alignment` property of the `strFormat` object to `StringAlignment.Far`.

The second boxed string in the second column was printed with similar statements, only this time we've specified the `DirectionVertical` option with the following statements:

```
strFormat = New System.Drawing.StringFormat()
strFormat.FormatFlags = StringFormatFlags.DirectionVertical
```

Printing vertically is equivalent to rotating the text. The alignment of the text is left to right, but the `DrawString` method starts printing the text from the bottom of the corresponding rectangle. You can experiment with different settings of the `strFormat` variable to see how they affect the way text is rendered on the page.

To print the rectangle with the gradient at the bottom of the page, we must first create a gradient brush with the following statements:

```
Dim p1 As New Point(e.MarginBounds.X, 0)
Dim p2 As New Point(e.MarginBounds.Width + e.MarginBounds.X, 0)
Dim color1 As Color = Color.Aquamarine
Dim color2 As Color = Color.DarkMagenta
Dim grBrush As New _
System.Drawing.Drawing2D.LinearGradientBrush(p1, p2, color1, color2)
```

The points, p1 and p2, are the two ends of the gradient (in effect, the extent of the gradient) and the two colors are the gradient's starting and ending colors. These variables are used in the definition of the brush's gradient. The following statement draws a rectangle filled with the grBrush brush:

```
e.Graphics.FillRectangle(grBrush, _
    New RectangleF(e.MarginBounds.X, 850, _
    e.MarginBounds.Width, 100))
```

The rectangle's left edge is at the left margin of the page and the rectangle's width is equal to the width of the printable area of the page. To draw the string on top of the gradient, we create a solid white brush and then call the DrawString method passing the string to be printed, the font in which the string will be rendered, the wBrush solid brush, and the same rectangle. To center the string both vertically and horizontally in the gradient's rectangle, we had to set both the Alignment and Line Alignment properties of the StringFormat object:

```
strFormat.Alignment = StringAlignment.Center
strFormat.LineAlignment = StringAlignment.Center
```

The string was printed on top of the gradient with the following statement:

```
Dim wBrush As New SolidBrush(Color.White)
e.Graphics.DrawString("Title on Gradient", tFont, wBrush, _
    New RectangleF(e.MarginBounds.X, 850, _
    e.MarginBounds.Width, 100), strFormat)
```

The last few statements in the PrintPage event handler determine whether the printout is sent to the printer or is previewed. When the output is sent to the printer, the ClipBounds property of the Graphics object represents the rectangle of the page. When the output is sent to the Preview control, the same property represents a virtual drawing surface with a width and height that exceed 23,000 inches. The code examines the value of the e.Graphics.ClipBounds.Width property and, if it's larger than 100,000 units, prints the string "Preview Mode." Otherwise, it prints the string "Print Mode."

```
strFormat = New StringFormat()
tFont = New Font(tFont.Name, 24, FontStyle.Bold)
tBrush = Brushes.LightGray
If e.Graphics.ClipBounds.Width > 1000000 Then
    strFormat.Alignment = StringAlignment.Center
    e.Graphics.DrawString("Preview Mode", tFont, tBrush, _
        New RectangleF(e.PageBounds.X, 50, _
        e.PageBounds.Width, 100), strFormat)
```


control, most of you would suggest a method to print the control's contents. This is what we're going to do in this section: we'll create a new control that inherits from the TextBox control and implements two new methods, the Print and Preview methods.

Both methods will print the control's contents taking into consideration the current setting of the Font and WordWrap properties. If the WordWrap property is False, the method will print each paragraph on a single line. It will start printing at the left margin and it will chop the text beyond the page's right margin. If the control's WordWrap property is True, the code is a bit more complicated. It must break the text into multiple lines at word boundaries and stop at the bottom of the current page, also at a word boundary. We take advantage of the MeasureString property to find out how many characters can be printed on each page, then print so many characters and continue with the remaining text on the next page. We must also make sure that the last line of text isn't partially printed. Figure 7.6 shows the code of the application in preview mode.

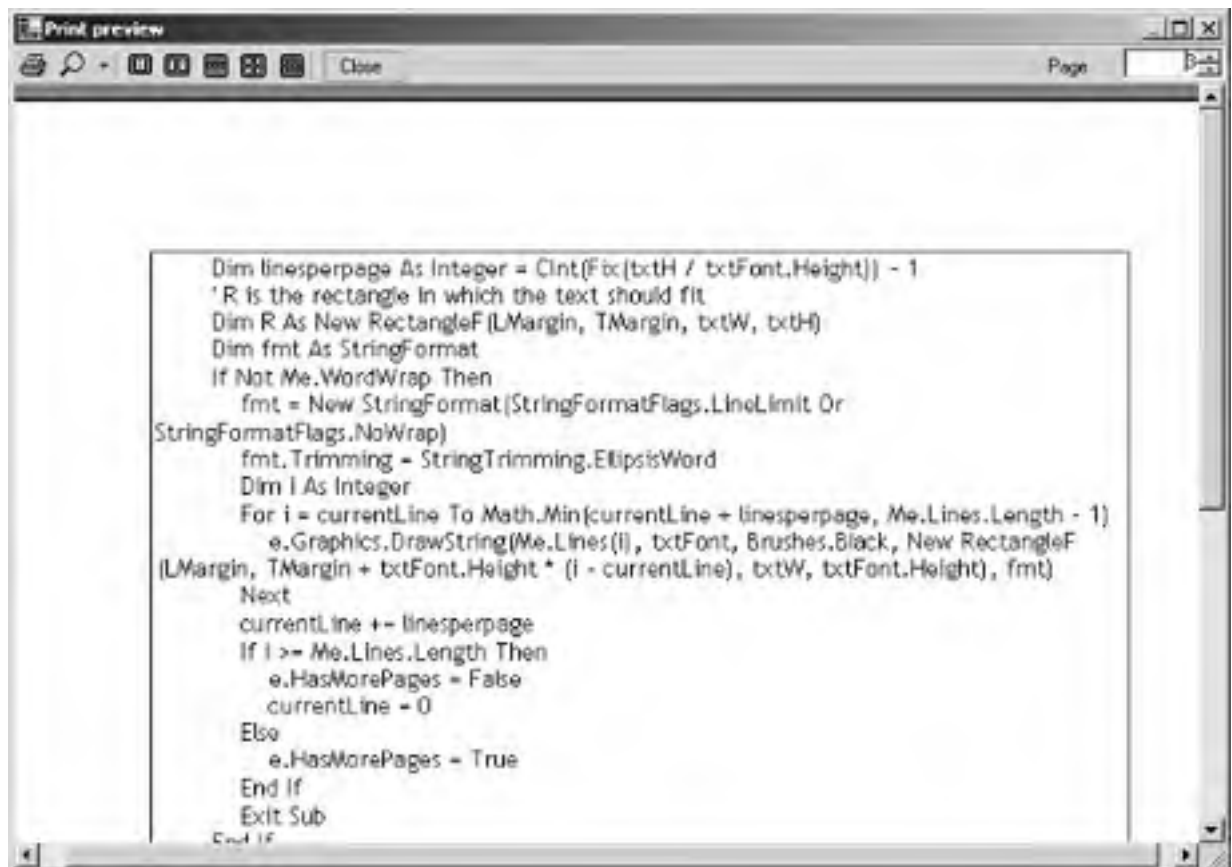


FIGURE 7.6 Printing the contents of the TextBox control

The sample project is called DemoControl. We'll create a custom control that provides the same functionality as the TextBox control, plus a custom method for printing its text. The Solution consists of two projects: a custom control and a test project. To use the custom control in your projects, add a reference to the DLL that will be created in the custom control's project Bin folder to any other project.

The custom control's code is fairly straightforward. Name the custom control PRNTextBox and a new class by that name will be created automatically. Then insert the following statement, right after the Class's definition:

```
Inherits System.Windows.Forms.TextBox
```

This statement tells the compiler that our custom control should include all the functionality of the built-in TextBox control. Then add the following declarations to create an instance of the

[Team Fly](#)

 Previous

Next 

PageSetupDialog control (we want to prompt users to set the page properties), an instance of the PrintPreview control (so that we can preview the text as it will be printed), and an instance of the PrintDocument object (we need this control to generate the printout). Notice that you can't create instances of these controls by dropping them onto the form, because you have no access to the UserControl object that represents an inherited control. In other words, you can't change the appearance of an inherited control with visual tools.

```
Friend PPView As New PrintPreviewDialog()  
Friend PSetup As New PageSetupDialog()  
Friend WithEvents PD As System.Drawing.Printing.PrintDocument
```

So far we've created the infrastructure for printing. The next step is to define the Print and Print-Preview methods. The reason we're adding two methods is that we don't want to bring up the preview window every time the users wants to print. Using the two methods, we allow the developer to determine whether the printout should be previewed or sent directly to the printer. The application that uses this control may have both a Print and a Preview command under the File menu, as do most applications that generate printouts.

The last step is to implement the PrintPage event handler of the PrintDocument object. Listing 7.2 shows the code of the event handler, which we'll explain in detail.

LISTING 7.2: PRINTING THE CONTENTS OF A TEXTBOX CONTROL

```
Private Sub PD_PrintPage( _  
    ByVal sender As Object, _  
    ByVal e As System.Drawing.Printing.PrintPageEventArgs)  
    Handles PD.PrintPage  
    Static currentChar As Integer  
    Static currentLine As Integer  
    Dim txtFont As Font = Me.Font  
    Dim txtH, txtW As Integer  
    Dim LMargin, TMargin As Integer  
    ' The dimensions of the printable area of the page  
    With PD.DefaultPageSettings  
        txtH = .PaperSize.Height - .Margins.Top - .Margins.Bottom  
        txtW = .PaperSize.Width - .Margins.Left - .Margins.Right  
        LMargin = PD.DefaultPageSettings.Margins.Left  
        TMargin = PD.DefaultPageSettings.Margins.Top  
    End With  
    e.Graphics.DrawRectangle(Pens.Blue, _  
        New Rectangle(LMargin, TMargin, txtW, txtH))  
    ' If landscape orientation, swap width and height  
    If PD.DefaultPageSettings.Landscape Then  
        Dim tmp As Integer  
        tmp = txtH  
        txtH = txtW  
        txtW = tmp  
    End If
```

```
' linesperpage is the number of lines per page
Dim linesperpage As Integer = CInt(Fix(txtH / txtFont.Height)) - 1
' R is the rectangle in which the text should fit
Dim R As New RectangleF(LMargin, TMargin, txtW, txtH)
Dim fmt As New StringFormat(StringFormatFlags.LineLimit)
If Not Me.WordWrap Then
    Dim i As Integer
    For i = currentLine To Math.Min(currentLine + linesperpage, _
        e.Lines.Length - 1)
        e.Graphics.DrawString(Me.Lines(i), txtFont, Brushes.Black, _
            New RectangleF(LMargin, _
                TMargin + txtFont.Height * (i - currentLine), _
                txtW, txtFont.Height), fmt)
    Next
    currentLine += linesperpage
    If i >= Me.Lines.Length Then
        e.HasMorePages = False
        currentLine = 0
    Else
        e.HasMorePages = True
    End If
    Exit Sub
End If
fmt = New StringFormat(StringFormatFlags.LineLimit)
Dim lines, chars As Integer
e.Graphics.MeasureString(Mid(Me.Text, currentChar + 1), txtFont, _
    New SizeF(txtW, txtH), fmt, chars, lines)
If lines = linesperpage Then
    If Me.Text.Substring(currentChar + chars, 1) <> vbLf And _
        Me.Text.Substring(currentChar + chars, 1) <> vbLf Then
        While chars > 0 AndAlso _
            Me.Text.Substring(currentChar + chars, 1) <> vbLf _
            Me.Text.Substring(currentChar + chars, 1) <> vbLf
            chars -= 1
        End While
        chars += 1
    End If
End If
e.Graphics.DrawString(Me.Text.Substring(currentChar, chars), _
    txtFont, Brushes.Black, R, fmt)
currentChar = currentChar + chars
If currentChar < Me.Text.Length Then
    e.HasMorePages = True
Else
    e.HasMorePages = False
    currentChar = 0
End If
End Sub
```


The event handler starts by calculating the printable area of the page and the left and top margins. If the user has requested a landscape orientation, the code swaps the page's width and height. Then it starts generating pages. If the WordWrap property of the TextBox control is set to False, the program prints one line at a time in a rectangle that fits a single line of text vertically, and doesn't care about long lines that won't be printed entirely on the page.

If the WordWrap property is set to True, the code calls the MeasureString method to find out how many characters will fit in a rectangle equal to the printable area of the page (the size of the page minus the margins specified by the user on the Page Setup dialog box) and prints so many characters in this rectangle. Then it subtracts the number of characters printed on the current page from the total number of characters left to be printed and examines whether there are more characters to be printed. If we're done, the code sets the HasMorePages property to False and exits. If not, it sets the same property to True and exits. In the next invocation of the PrintPage event handler, it prints another page and continues until the entire text is printed.

The code that implements the Print method is straightforward. For each page, it creates a rectangle equal to the printable area of the page and fills it with text. If the control's Wrap property is set to False, the Print method prints one line at a time regardless of the line's length. Notice that the DrawString method uses a StringFormat property with its FormatFlags property set to LineLimit. This setting prevents the DrawString method from partially printing the last line on the page. One disadvantage of the MeasureString method is that it doesn't take into consideration word boundaries. We don't want to break the last word on the page, so the program backtracks from the last character reported by the MeasureString method until it finds a space or a newline character. The last line is broken at this character, so the new page will start at a word boundary. Tabs are not treated as white space; only newline and space characters are. Punctuation symbols follow the previous word. This is a very important detail, which you should handle in your text printing code.

Figure 7.7 shows a detail of the program's printout, when printed with the WordWrap property set to False. Notice that the lines that can't fit across the page are truncated at a word boundary and an ellipsis is added after the last visible word to indicate that some text is missing. The breaking of the word and the insertion of the ellipsis are handled automatically with the Trimming property of the StringFormat class, which is set to the value StringTrimming.EllipsisWord. See the members of the StringTrimming enumeration for other possible settings of the Trimming property.

```
    If l >= Me.Lines.Length Then
        a.HasMorePages = False
        currentLine = 0
    Else
        a.HasMorePages = True
    End If
    Exit Sub
End If
fmt = New StringFormat(StringFormatFlags.LineLimit)
Dim lines, chars As Integer
' Call the MeasureString method to retrieve the number of characters
' and number of lines of text that will fit in the specified rectangle
' when printed in the specified font. Then print on the page as many
' characters of the text as reported by the MeasureString method.
a.Graphics.MeasureString(WidWm.Text, currentChar + 1, txtFont, ...
```

```
So far we've calculated the number of lines and number of...
that can be printed on the printable area of the page. However...
method reports the exact number of characters that will fit on the...
and doesn't take into consideration word boundaries. If the last...
is not at a word boundary (a space or carriage return character)...
backtrack in the string until we find the last word boundary. This...
we'll stop printing at a word boundary and we won't break the last...
the page.
```

FIGURE 7.7 Printing with the WordWrap property set to False

PrintReportTitle The report's title.

PrintReportDate A Boolean property that determines whether the date will be printed at the top of each page.

PrintPageNumbers A Boolean property that determines whether a page number will be printed at the top of each page.

PrintTitleSmallFont, PrintTitleLargeFont The font settings that will be used to print the report's title (large font) and the date/numbers (small font).

PrintTitleColor The color of the report's title.

The code of the Print method displays the PageSetup dialog box, then passes the PrintDocument object to the Document property of the PrintPreview control and calls its ShowDialog method to initiate the printout. Listing 7.3 shows the Print method's code.

LISTING 7.3: INITIATING THE PRINTOUT OF THE PRNLISTVIEW ITEMS

```
Public Sub Print()  
    PD = New Printing.PrintDocument  
        PSetup.PageSettings = PD.DefaultPageSettings  
        PSetup.ShowDialog()  
        PWidth = PD.DefaultPageSettings.PaperSize.Width  
        PHeight = PD.DefaultPageSettings.PaperSize.Height  
        ' cellWidth and cellHeight are the width and height  
        ' of the current subitem s cell  
        X = PD.DefaultPageSettings.Margins.Left  
        Y = PD.DefaultPageSettings.Margins.Top  
        If PD.DefaultPageSettings.Landscape Then  
            Dim tmp As Integer  
            tmp = PWidth  
            PWidth = PHeight  
            PHeight = tmp  
        End If  
        PageWidth = PWidth - _  
            (PD.DefaultPageSettings.Margins.Left + _  
             PD.DefaultPageSettings.Margins.Right)  
        PageHeight = PHeight - _  
            (PD.DefaultPageSettings.Margins.Top + _  
             PD.DefaultPageSettings.Margins.Bottom)  
        PPView.Document = PD  
        X = PD.DefaultPageSettings.Margins.Left  
        Y = PD.DefaultPageSettings.Margins.Top  
        PageNo = 0  
        PPView.ShowDialog()  
End Sub
```

The code that implements the various properties is trivial and we need not show it here. All properties are stored in private variables, which are used by the `PrintPage` event handler's code, which does all the work. Before looking at this code, let's discuss a few global variables. These variables must maintain their values between successive invocations of the `PrintPage` event handler. They are as follows:

```
Private X, Y As Integer
Private Ytop As Integer
Private PWidth As Integer
Private PHeight As Integer
Private PageNo As Integer
Private cellWidth, cellHeight As Integer
```

`X` and `Y` are the coordinates of the current cell; they're updated after printing a cell, or after switching to another row of cells. `CellWidth` and `CellHeight` are the dimensions of the current cell. The width of the current cell is determined by the width of the corresponding column of the `ListView` control, and the height of the current cell is determined by its contents and the setting of the `PrintMaxCellLines` property. The width of a printed column is proportional to the width of the corresponding column of the control, but not equal. The code maintains the relative widths of the columns, but it also fills the page (minus the margins, of course). `PWidth` and `PHeight` are the dimensions of the page; the code subtracts the appropriate margins from these two variables to calculate the printable area of the page. Finally, the `PageNo` variable stores the number of the current page.

The variables `PD` (a `PrintDocument` object), `PSetup` (an instance of the `PageSetup` dialog box) and `PPView` (an instance of the `PrintPreview` dialog box) are declared on the form level with the following statements:

```
Friend PPView As New PrintPreviewDialog
Friend Psetup As New PageSetupDialog
Friend WithEvents PD As System.Drawing.Printing.PrintDocument
```

Finally, the `PageWidth` and `PageHeight` variables are also declared at the form level as integers, because they're used throughout the code.

Listing 7.4 shows the code of the `PrintPage` event, which generates the printout one page at a time.

LISTING 7.4: GENERATING THE LISTVIEW CONTROL'S PRINTOUT

```
Private Sub PD_PrintPage(ByVal sender As Object, _
    ByVal e As System.Drawing.Printing.PrintPageEventArgs) _
    Handles PD.PrintPage
    PageNo += 1
    PageWidth = PWidth - _
        (PD.DefaultPageSettings.Margins.Left + _
        PD.DefaultPageSettings.Margins.Right)
    PageHeight = PHeight - _
        (PD.DefaultPageSettings.Margins.Top + _
        PD.DefaultPageSettings.Margins.Bottom)
    X = PD.DefaultPageSettings.Margins.Left
```



```
Y = PD.DefaultPageSettings.Margins.Top
Static startItem As Integer
Dim ColWidths(Me.Columns.Count - 1) As Integer
' calculate the width of each column and the total width of the grid
Dim i As Integer, totWidth As Integer
For i = 0 To Me.Columns.Count - 1
    ColWidths(i) = Me.Columns(i).Width
    totWidth = totWidth + ColWidths(i)
Next

PrintPageHeader(e.Graphics)
Dim R As Rectangle, RF As RectangleF
Dim caption As String
Dim titleFont As Font = Me.Font
Dim itemFont As Font
Dim titleBrush As New SolidBrush(Color.Black)
Dim itemBrush As System.Drawing.Brush
itemBrush = Brushes.Black
Dim borderPen As New Pen(_borderColor, 1)
Dim txtWidth As Integer
Dim fmt As New StringFormat
fmt.Trimming = StringTrimming.EllipsisCharacter
Dim txtSize As SizeF
For i = 0 To Me.Columns.Count - 1
    caption = Me.Columns(i).Text
    cellWidth = Convert.ToInt32(ColWidths(i) * _
        (PageWidth - (Me.Columns.Count - 0) * _
        _ColumnPadding) / totWidth)
    txtSize = e.Graphics.MeasureString(caption, titleFont)
    txtSize.Height = Math.Min(txtSize.Height, _
        titleFont.GetHeight(e.Graphics) * _maxCellLines)
    cellHeight = Convert.ToInt32(txtSize.Height)
    R = New Rectangle(X, Y, cellWidth + _ColumnPadding, cellHeight)
    e.Graphics.DrawRectangle(borderPen, R)
    RF = New RectangleF(X + _ColumnPadding, Y, cellWidth - _
        _ColumnPadding, cellHeight)
    Select Case Me.Columns(i).TextAlign
        Case HorizontalAlignment.Center _
            : fmt.Alignment = StringAlignment.Center
        Case HorizontalAlignment.Left _
            : fmt.Alignment = StringAlignment.Near
        Case HorizontalAlignment.Right _
            : fmt.Alignment = StringAlignment.Far
    End Select
    e.Graphics.DrawString(caption, titleFont, titleBrush, RF, fmt)
    X = X + cellWidth + _ColumnPadding
Next
Dim itm, sitm As Integer
```

```
Y = Y + cellHeight + VSpacing
Dim SF As SizeF

' Now iterate through a number of items and print them.
' The exact number of items that will be printed varies,
' depending on the height of each cell. Some of the items/subitems
' may not fit on a single line
' The index of the last item printed is stored in the startItem stati
' variable, so that printing will resume with the following item the
' next time the PrintPage event is fired
    fmt.Trimming = StringTrimming.EllipsisCharacter
    For itm = startItem To Me.Items.Count - 1
        X = PD.DefaultPageSettings.Margins.Left
        Dim tallestCell As Integer = 0
        For sitm = 0 To Me.Items(itm).SubItems.Count - 1
            caption = Me.Items(itm).SubItems(sitm).Text
            cellWidth = Convert.ToInt32(ColWidths(sitm) * _
                (PageWidth - Me.Columns.Count * _
                _ColumnPadding) / totWidth)
            ' itemFont is set to the font used to render
            ' the corresponding subitem on the control
            itemFont = Me.Items(itm).Font
            SF = New SizeF(cellWidth, 100)
            txtSize = e.Graphics.MeasureString(caption, _
                itemFont, SF, fmt)
            ' keep track of the tallest cell in the current item
            txtSize.Height = Math.Min(txtSize.Height, _
                itemFont.GetHeight(e.Graphics) * _maxCellLines)
                + VSpacing
            If txtSize.Height > tallestCell Then tallestCell = _
                Convert.ToInt32(txtSize.Height)
            ' print the subitem with its original alignment
            ''' NOTE: The following statement won't work !!!
            '''   fmt.Alignment = Me.Columns(sitm).TextAlign
            '''   The Alignment property of the StringFormat object
            '''   doesn't have the same settings as the ListViewItem
            '''   object's TextAlign property.
            Select Case Me.Columns(sitm).TextAlign
                Case HorizontalAlignment.Center _
                    : fmt.Alignment = StringAlignment.Center
                Case HorizontalAlignment.Left _
                    : fmt.Alignment = StringAlignment.Near
                Case HorizontalAlignment.Right _
                    : fmt.Alignment = StringAlignment.Far
            End Select
            RF = New RectangleF(X + _ColumnPadding, Y, cellWidth - _
                _ColumnPadding, txtSize.Height - VSpacing)
            e.Graphics.DrawString(caption, itemFont, itemBrush, RF, f
```

```
        X = X + cellWidth + _ColumnPadding
    Next
    Y = Y + tallestCell + VSpacing
    If PD.DefaultPageSettings.Landscape Then
        e.Graphics.DrawLine(New Pen(_borderColor), _
            PD.DefaultPageSettings.Margins.Left, _
            Y, PD.DefaultPageSettings.Margins.Left + _
            PageWidth, Y)
    Else
        e.Graphics.DrawLine(New Pen(_borderColor), _
            PD.DefaultPageSettings.Margins.Left, _
            Y, PD.DefaultPageSettings.Margins.Left + _
            PageWidth + 1, Y)
    End If
    ' start a new page is the current row's cells exceed
    ' 95% of the page's printable area
    If Y > 0.95 * (PHeight - _
        PD.DefaultPageSettings.Margins.Bottom) Then
        ' now print the vertical lines
        X = PD.DefaultPageSettings.Margins.Left
        For i = 0 To Me.Columns.Count - 1
            e.Graphics.DrawLine(New Pen(_borderColor), X, _
                Ytop, X, Y)
            cellWidth = Convert.ToInt32(ColWidths(i) * _
                (PageWidth - Me.Columns.Count * _
                _ColumnPadding) / totWidth)
            X = X + cellWidth + _ColumnPadding
        Next
        '''' draw the last vertical line
        e.Graphics.DrawLine(New Pen(_borderColor), X, _
            Ytop, X, Y)
        ' and the bottom horizontal line
        e.Graphics.DrawLine(New Pen(_borderColor), _
            PD.DefaultPageSettings.Margins.Left, Y, _
            PD.DefaultPageSettings.Margins.Left + _
            PageWidth, Y)
        e.HasMorePages = True
        startItem = itm + 1
        Exit Sub
    End If
Next
' draw the grid of the last page,, which is usually smaller than
' the other pages
X = PD.DefaultPageSettings.Margins.Left
For i = 0 To Me.Columns.Count - 1
    e.Graphics.DrawLine(New Pen(_borderColor), X, Ytop, X, Y)
    cellWidth = Convert.ToInt32(ColWidths(i) * (PageWidth - _
        (Me.Columns.Count - 0) * _ColumnPadding) / totWidth)
```



```
        X = X + cellWidth + _ColumnPadding
    Next
    e.Graphics.DrawLine(New Pen(_borderColor), X, Ytop, X, Y)
    e.Graphics.DrawLine(New Pen(_borderColor), _
        PD.DefaultPageSettings.Margins.Left, Y, _
        PD.DefaultPageSettings.Margins.Left + _
        PageWidth - _ColumnPadding, Y)
    e.HasMorePages = False
    ' This is a static variable and won't be reset automatically
    startItem = 0
End Sub
```

The listing is fairly lengthy, but straightforward. The first `For ... Next` loop retrieves the widths of the control's columns and stores them in the `ColWidths` array. It also stores the total width of the columns in the `totWidth` variable. Each printed column's width is calculated with the following statement (`sitm` is the order of the columns):

```
cellWidth = Convert.ToInt32(ColWidths(sitm) * _
    (PageWidth - Me.Columns.Count * _
    _ColumnPadding) / totWidth)
```

The width of the printed column is proportional to the width of the corresponding control's column, but we make sure that the printed columns fill the page.

For each new page, the code prints the report's title, the date, and the page number. Then it prints the header of each column (the headers are picked from the control's `Columns` collection). After printing the headers, the code iterates through the control's items. It uses the `SubItems` collection to read each cell's contents and the `Columns` collection to read each cell's alignment. The font for each row is read from the `Items` collection. The code assumes that the subitems are rendered in the same font as the first column (you can change this behavior by reading the `Font` property of each element in the `SubItems` collection).

The most interesting part of the code is the statements that calculate the height of each cell, shown next:

```
txtSize = e.Graphics.MeasureString(caption, itemFont, SF, fmt)
txtSize.Height = Math.Min(txtSize.Height, _
    itemFont.GetHeight(e.Graphics) * _maxCellLines) _
    +VSpacing
If txtSize.Height > tallestCell Then tallestCell = _
    Convert.ToInt32(txtSize.Height)
```

The program calls the `MeasureString` method to retrieve the necessary height of the cell in which the text must fit (the cell's width is known). Then it updates the `tallestCell` variable, which stores the height of the tallest cell in the row. The code will advance by so many units before printing the next row—if there's room for another row on the page. Once the current cell's width and height are known, the code prints the cell's text in a rectangle specified by the current cell's origin and its dimensions.

LISTING 7.5: IMPORTING API FUNCTIONS IN .NET APPLICATIONS

```
Private Declare Function BitBlt Lib "gdi32" Alias "BitBlt" _
    (ByVal hDestDC As Integer, ByVal x As Integer, _
    ByVal y As Integer, ByVal nWidth As Integer, _
    ByVal nHeight As Integer, ByVal hSrcDC As Integer, _
    ByVal xSrc As Integer, ByVal ySrc As Integer, _
    ByVal dwRop As Integer) As Integer

Private Declare Function GetDC Lib "user32" Alias "GetDC" _
    (ByVal hwnd As Integer) As Integer

Private Declare Function ReleaseDC Lib "user32" Alias "ReleaseDC" _
    (ByVal hwnd As Integer, ByVal hdc As Integer) As Integer
```

The BitBlt function accepts as arguments a handler to the destination device context (hDestDC argument) and an argument to the source device context (hSrcDC argument) and copies the bitmap (or part of it) of the source device context to the destination device context. The remaining arguments determine what part of the destination device context will be filled and what part of the source device context will be copied. The arguments x, y, nWidth, and nHeight determine the area to be copied, while the arguments xSrc and ySrc determine the origin of the source bitmap to be copied. The two bitmaps can't have different dimensions. The GetDC function retrieves a handler to the device context of the control specified by its argument and the ReleaseDC function releases this handler.

To use the two API functions explained here, you must write some code to capture two designated keystrokes. In our sample code we'll use the keystroke Ctrl+P to print the entire form, and the keystroke Alt+P to print the current form. Set the form's KeyPreview property to True and enter the following statements in the form's KeyUp event handler:

```
Private Sub Form1_KeyUp(ByVal sender As Object, _
    ByVal e As System.Windows.Forms.KeyEventArgs) _
    Handles MyBase.KeyUp
    If e.KeyCode = Keys.P And e.Alt Then
        bmp = CreateScreenshot(captureArea.Form)
        PrintBMP()
    End If
    If e.KeyCode = Keys.P And e.Control Then
        bmp = CreateScreenshot(captureArea.Screen)
        PrintBMP()
    End If
End Sub
```

The CreateScreenShot() function, whose code is shown in Listing 7.6, accepts as argument a constant that determines what part of the screen we want to capture (the current form or the entire

screen) and returns the corresponding bitmap. Once we have the bitmap, we call the PrintBMP() function to print the same bitmap. The bmp variable is declared at the form's level, so that it can be accessed by all procedures.

LISTING 7.6: CAPTURING A WINDOW OR THE DESKTOP TO A BITMAP

```
Function CreateScreenshot(ByVal Capture As captureArea) As Bitmap
    Dim Rect As Rectangle
    If Capture = captureArea.Form Then
        Rect = New Rectangle(Me.Left, Me.Top, Me.Width, Me.Height)
    Else
        Rect = Screen.PrimaryScreen.Bounds()
    End If
    Dim gDest As Graphics
    Dim hdcDest As IntPtr
    Dim hdcSrc As Integer

    Dim screenBMP As New Bitmap(Rect.Right, Rect.Bottom)
    gDest = gDest.FromImage(screenBMP)

    hdcSrc = GetDC(0)
    hdcDest = gDest.GetHdc
    If Capture = captureArea.Form Then
        BitBlt(hdcDest.ToInt32, 0, 0, _
            Rect.Width, Rect.Height, hdcSrc, Me.Left, Me.Top, SRCCOPY)
    Else
        BitBlt(hdcDest.ToInt32, 0, 0, _
            Rect.Right, Rect.Bottom, hdcSrc, _
            Screen.PrimaryScreen.Bounds.Left, _
            Screen.PrimaryScreen.Bounds.Top, SRCCOPY)
    End If
    gDest.ReleaseHdc(hdcDest)
    ReleaseDC(0, hdcSrc)
    Return screenBMP
End Function
```

The CreateScreenShot() function accepts a single argument, which is a member of the CaptureArea custom enumeration (its members being Screen and Form). It defines a rectangle that encloses the specified area to be captured and then uses the BitBlt() subroutine to capture this rectangle into a bitmap with the proper dimensions. This bitmap is the function's return value.

The PrintBMP() subroutine, shown in Listing 7.7, displays the Page Setup dialog box and then initiates the printing.

Try `System.String`. Nothing. Try leaving out *System*: Typing `String`. (don't forget the dot.) displays a nice list of all the methods you can use with strings. The `IndexOf` method looks promising at first glance. It's the replacement for `InStr`, and it does locate substrings, but it still forces you to write a messy loop. Isn't there any way to just pass a delimited string to a function and get back an array of the pieces? Doesn't this function exist?

How about `String.Parse`? No such entry. Keep looking. What about `String.Split`? Eureka. Help describes it like this: "Identifies the substrings in this instance that are delimited by one or more characters specified in an array, then places the substrings into a `String` array." This near-English sentence *might* be describing what we're after. Unfortunately there's no mention here of the actual, necessary word *parse*, but .NET Help is being improved all the time. Eventually, it may even become nearly understandable. One always hopes.

.NET has only been out three years. Perhaps someday they'll have the wisdom to get an actual writer to rewrite the Help descriptions. If so, the Search feature will be more useful than it is today, when searching for *parse* results in dozens of hits for the completely unrelated task of type conversion.

Between WWII and 1980 you would buy a transistor radio and get an instruction manual with sentences like this: "Press top button on the bottom of our most handy top. Not blue. Then pause fine. Happy you luck! Now hear song fast everywhere!" It only took the typical Asian electronics manufacturer three decades before grasping the value of hiring a \$14,000-a-year copy editor who knows English to review equipment manuals for correct English.

The syntax for `Split` is displayed like this:

```
Overloads Public Function Split( _  
    ByVal ParamArray separator() As Char _  
) As String()
```

So you have to pass an array of `Chars` specifying your delimiter(s), and it returns a string array. You may or may not know how to build an array of characters. Again you have to search around. You might think you can create the array, create a char variable holding your comma, then assign the variable to the array, like this:

```
Dim c As Char = ","  
Dim delim As Char()  
delim(0) = c
```

You get the famous "no object" error message from the compiler: "Object reference not set to an instance of an object." Oh well. No point trying to figure out why you cannot just create and declare an array, then fill it with values. When you come upon this kind of problem, try other syntaxes, or hope there's some example code in Help.

Another way to solve this problem is fairly strange, too. You create a string, then change it into a char type using the `ToCharArray` method:

```
Dim delimStr As String = ","  
Dim delim As Char() = delimStr.ToCharArray()
```

Why there is a ToCharArray method, but no ToStringArray method, is anybody's guess.

The most straightforward way to solve the problem of adding values to a char array is to just Dim the char array, and initialize it with the comma directly:

```
Dim delim As Char() = {",", "}"
```

[Team Ely](#)

 Previous

Next 

To print in .NET, use the PrintDocument control, which generates output for the printer. To present a preview of the printout, just assign the PrintDocument object to the Document object of the PrintPreview control. The code you'll write is identical, regardless of whether the output is printed or previewed. The PrintDocument object fires the PrintPage event every time a new page is started. In this event's handler, you must insert the code that will generate your printout. Every time you fill a page, you exit this event handler and the printout will be sent to the printer. If you want to print additional pages, just set the HasMorePages to True. Otherwise set it to False.

The two examples in this chapter are two custom controls, which expose the same functionality as the TextBox and ListView controls, respectively, and add a Print method. The Print method of the custom TextBox control prints the control's text, while the Print method of the custom ListView control prints the control's items. Both methods take into consideration the control's properties that may affect the printout.

Alas, the word *type* itself is used in a new way in VB.NET as well. It's used the way C programmers prefer to use it—and now we must also use it that way. As you know, in .NET pretty much everything that is in existence at runtime is an *object*. Similarly, the term *type* includes pretty much every programming building block available during design time. *Type* is a larger category than *class* because a class is a type, but so are arrays, modules, enumerations, structures, interfaces, and value types.

The System.Type class includes quite a few features you can employ to enumerate a specific type's members: events, properties, methods, and fields. System.Type also includes capabilities that modify a specific type's properties and fields, as well as dynamically executing methods.

Getting a Grip on Assemblies

Assemblies also include information about objects' members: fields, properties, events, methods, and parameters. Physically, on the hard drive, an assembly is quite similar to the traditional "compiled" VB application (such as a DLL or EXE file, or a set of them working together). But an assembly also contains information about the application that would have previously been put into the Windows Registry or a COM type library. Assemblies contain "self-describing" metadata, security information, the name and version of the assembly, the "culture," a list of dependency files (bitmaps, other libraries such as needed DLLs, schemas, whatever).

THE INFLATION AND BIFURCATION OF VISUAL BASIC

Where would we be without constantly shifting diction in computer languages? VB continues the inflation that started in VB version 4 several years ago—when the language first began the explosion from its original modest 300+ commands to what are now tens of thousands of classes, and many more tens of thousands of often overloaded members within those classes. One side effect of this inflation is that part of our programming time is spent merely trying to wrestle with the language's classification system. Like biologists, we must try to deal with many nested categories—containers within containers. For example, a line of code describing a single object might require four or five "qualifiers" separated by dots, like this:

```
Reflection.Assembly.GetAssembly(Me.GetType())
```

And, instead of the earlier simplicity of programming like this:

```
ListBox1.AddItem(n)
```

.NET usually requires additional qualification:

```
ListBox1.Items.Add(n)
```

Biology, with its focus on categorization, isn't considered the most sophisticated branch of science. Math is. You can visualize the varying degrees of scientific rigor illustrated by this story: A biologist, a physicist, and a mathematician were traveling through Scotland by train and saw a black sheep in a meadow. "All sheep in Scotland are black," concluded the biologist. "No," said the physicist, "all we can say is that one sheep in Scotland is black." The mathematician said, "All we can say is that one sheep in Scotland is black on one side."

In spite of the bifurcation and inflation that Visual Basic has undergone, few programmers can cling to earlier, simpler versions of VB. If nothing else, the bandwidth-restricted, security-conscious programming techniques demanded by the Internet (not least of which is statelessness) require that the language, and its practitioners, adapt.

[Team Fly](#)

 Previous

Next 

An assembly contains within it both the usual information about a project (the properties of a form, for example) and metainformation (information about information), plus esoterica, such as methods used primarily by the VB.NET runtime and normally kept hidden from the programmer (*GetActiveControl* or *ResetForeColor*, for example).

The relationship between the terms *assembly*, *solution*, *project*, *module*, and *namespace* is rather slippery. But you may as well learn about these containers. At the lower level in this system of categorization, you have modules (traditionally used in VB as places to put global variables or functions) and forms. Forms are technically referred to as modules (how's that for confusing?), specifically *form modules*. Next up in the system is the *project*, which is a container for one or several related modules and their dependencies (a project is often the entire application—there are no additional projects). Higher still is the entire *solution* (an assembly) that can have one or more projects within it.

Containers within Containers

So, a given assembly (application) can contain multiple projects, and each project can contain multiple modules. Now for the really slippery part: *namespaces*. They are not so much an actual programming *container* as they are a convenience, a shorthand to permit programmers to avoid having to fully qualify methods, enumerations, and other features made available to a programmer. Namespaces are also a way to avoid *collisions*. A "collision" occurs when two types share the same name. For example, two functions might both be named *FactorIt*, but might do completely different jobs and require different parameters. An error is thrown if names collide within a given namespace, or if you have two namespaces with colliding names. This alerts you that you must qualify the names to avoid the collision. By qualifying (specifying which namespace you intend), you solve the problem of duplicated type names.

When you import a namespace, you are specifying a group of related classes—such as the System.Reflection namespace that contains 133 classes (along with those ghostly class "sketches" called *interfaces*)—that collectively make up the reflection technology.

In any case, think of a namespace as a fairly flexible rubber band that you can use to tie together related classes. Flexible because namespaces can be huge (spanning multiple modules) or, by contrast, quite small (a single module can contain many namespaces, if you wish it so). You, of course, can create namespaces in your source code as a way of subdividing it. You can also reference external class libraries by using the `Imports` command. What you cannot do is use the same name for a method or other type *twice* in the same namespace.

Security Issues

You may have noticed that security problems are at the root of many contemporary computing problems—not to mention that security issues often cause programming difficulties. In fact, some argue that OOP itself is primarily an attempt to increase security.

Those hackers are having much more of an impact than is commonly acknowledged.

Security also underlies many aspects of .NET programming. The metadata in an assembly explains each type, and all the members, within that assembly. An assembly also includes a public key—much like authentication technology used with e-mail and file transmission. This key is examined by .NET to discover whether or not the assembly has been tampered with, either by a disk accident or by an evildoer. In addition, assemblies' components can contain various permissions levels—just as you can grant sharing and grouping permissions within the Windows world, you can also thus secure and manage access to your projects.

[Team Fly](#)

 Previous

Next 

```
Class puria

    Private mXX As String

    'a couple of overloaded constructors
    Sub New(ByVal initValue As String) 'let them pass a string
        MyBase.New()
        mXX = initValue
    End Sub

    Sub New(ByVal sa As Single) 'or let them pass a single
        MyBase.New()
        mXX = sa
    End Sub

    Public ReadOnly Property TheID()
        Get
            TheID = mXX
        End Get
    End Property

    Public Function SendBak() As String
        Return mXX.ToString
    End Function

End Class
```

BINDINGFLAGS

The `BindingFlags` are a way you can filter the results. You can divide members into instance or static categories. Here, by mixing two flags together with `Or` you get both instance and public constructors.

`BindingFlags` can be used when retrieving other reflected data, such as `PropertyInfo`, for example. There are a variety of `BindingFlags` in the enumeration, but here are the most useful for reflected properties:

- ◆ `DeclaredOnly`
- ◆ `FlattenHierarchy`
- ◆ `IgnoreCase`
- ◆ `IgnoreReturn`
- ◆ `Instance`
- ◆ `NonPublic`

- ◆ Public
- ◆ Static

If you are interested in seeing all the properties, for example, you could use this list of flags:

```
Dim cinfo As ConstructorInfo() =  
t.GetConstructors((BindingFlags.Instance Or _ BindingFlags.NonPublic  
Or BindingFlags.Public))
```

These flags will give you all properties (public, protected, private, and instance). Here is a list of the methods of the `Type` class that permit you to filter data by using the `BindingFlags`:

- ◆ GetMembers
- ◆ GetEvents
- ◆ GetConstructor
- ◆ GetConstructors
- ◆ GetMethod
- ◆ GetMethods
- ◆ GetField
- ◆ GetFields
- ◆ GetEvent
- ◆ GetProperty
- ◆ GetProperties
- ◆ GetMember
- ◆ FindMembers

In the previous example, you have created a class with overloaded constructors. You use reflection to access your class *puria* and report back what constructors it uses. When you run this example, your `TextBox` reports:

```
Here are the public constructors of the type Reflect1.puria  
Void .ctor(System.String)  
Void .ctor(Single)
```

Ignore that `Void`, it's C#. What you're interested in is the details that tell you this class has two constructors, one accepting a string and the other accepting a single. First you get the type, then you get the *constructorinfo*.

What makes this technique more useful is using it against an assembly or class that *isn't visible in your code window*. In other words, you don't really need to use reflection on *puria* here; after all, you wrote it, and you can sit there and read it—so you already know what constructors it has and their argument lists.

But let's assume you don't know the details about an object, or that you can't easily read them. Ignore for now that you could use IntelliSense or the object browser to find out details about objects

in various assemblies. Assume you want to find out the constructors for the System.TimeSpan class. Edit this line:

```
Dim t As Type = GetType(System.TimeSpan)
```

Then rerun the program to get this result, showing that this class has four possible constructors: Here are the public constructors of the type System.TimeSpan

```
Void .ctor(Int64)
Void .ctor(Int32, Int32, Int32)
Void .ctor(Int32, Int32, Int32, Int32)
Void .ctor(Int32, Int32, Int32, Int32, Int32)
```

Of course, constructor information isn't the only data you can request from reflection on a type. You can also extract properties, methods, events, and fields, using the following objects associated with the Type object respectively: PropertyInfo, MethodInfo, MemberInfo, and FieldInfo.

To see public properties, enter:

```
Dim t As Type = GetType(pur1a)
    TextBox1.Text = 'Here are the properties of the type "& t.ToString & c
Dim cinfo As PropertyInfo() = _ t.GetProperties((BindingFlags.Public Or _
BindingFlags.Instance))
```

which results in this answer:

```
Here are the properties of the type Reflect1.pur1a
System.Object TheID
```

Or if you run this example using the TimeSpan class as your target of reflection, you get this result:

```
Here are the properties of the type System.TimeSpan
Int64 Ticks
Int32 Days
Int32 Hours
Int32 Milliseconds
Int32 Minutes
Int32 Seconds
Double TotalDays
Double TotalHours
Double TotalMilliseconds
Double TotalMinutes
Double TotalSeconds
```

As you can see, this reflection technique begins to show promise. You could select *during runtime* from among these various properties. And the same dynamic runtime decisionmaking is possible with the methods of a class. Use this:

```
Dim Minfo As MethodInfo() = t.GetMethods((BindingFlags.Public Or
BindingFlags.Instance))
```


to get this result:

```
Here are the methods of the type System.TimeSpan
Int32 CompareTo(System.Object)
Int32 GetHashCode()
Boolean Equals(System.Object)
System.String ToString()
Int64 get_Ticks()
Int32 get_Days()
Int32 get_Hours()
Int32 get_Milliseconds()
Int32 get_Minutes()
Int32 get_Seconds()
Double get_TotalDays()
Double get_TotalHours()
Double get_TotalMilliseconds()
Double get_TotalMinutes()
Double get_TotalSeconds()
System.TimeSpan Add(System.TimeSpan)
System.TimeSpan Duration()
System.TimeSpan Negate()
System.TimeSpan Subtract(System.TimeSpan)
System.Type GetType()
```

Accessing the Current Project's Assembly

In the following example, you load the assembly of the currently running VB.NET project:

```
Dim a As Reflection.Assembly
a = Reflection.Assembly.GetAssembly(Me.GetType())
```

NOTE You can either continue with the above sample code or start a new VB.NET application.

Notice that you use the `GetAssembly` method, in concert with the `GetType` method of the form object, to instantiate this assembly. You can also use `Load` or `LoadFrom` methods of the assembly class, as later examples illustrate further on in this chapter.

Replace the code in the previous example with this:

```
Imports System.Reflection

Dim cr As String = ControlChars.CrLf

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

    Dim a As Reflection.Assembly
    a = Reflection.Assembly.GetAssembly(Me.GetType())
```



```
AddText('Module: ' & a.GetModules() (0).Name)
AddText("////////")
AddText("Type: " & a.GetTypes() (0).Name)
AddText("////////")
Dim T As Type = a.GetTypes() (0)
Dim Enumerator As IEnumerator = _
    T.GetMethods.GetEnumerator
AddText("Methods:.....")
While (Enumerator.MoveNext)
    AddText(CType(Enumerator.Current, MethodInfo).Name)
End While
AddText("Fields:..... ")
Dim FEnumerator As IEnumerator = T.GetFields.GetEnumerator
While (FEnumerator.MoveNext)
    AddText(CType(Enumerator.Current, FieldInfo).Name)
End While
End Sub
Private Sub AddText(ByVal s As String)
    TextBox1.Text &= s & cr
End Sub
```

You see the name of your project (module), the type of `Me` (the current object, `Form1`), followed by a lengthy list of methods—most of which are kept invisible to programmers in the normal course of programming in VB.NET. Reflection, however, goes deep into an assembly (an application) and returns with all the metadata. In this example, there are no fields, so that loop returns no results.

If you want to see the static (shared) members, use this set of bit codes:

```
((BindingFlags.Static Or BindingFlags.NonPublic Or BindingFlags.Public))
```

To see instance members, use this argument:

```
((BindingFlags.Instance Or BindingFlags.NonPublic Or BindingFlags.Public))
```

Accessing a Loaded Assembly

You can also use the `GetType` method an alternative way. What if you want to access an assembly that is referenced in your current solution, such as `System.XML` or `System.Data`?

In this example, you provide a known type as an argument of `GetType`, which then automatically gives you access to that type's entire assembly. In this next example, you use `GetType` to load the `System.Data` assembly by providing the `DataRow` class as your argument.

Add a `ListBox` to the previous example's form and replace the code in the previous example with Listing 8.2.

LISTING 8.2 ACCESSING A KNOWN TYPE

```
Dim cr As String = ControlChars.CrLf

    Dim a As System.Reflection.Assembly
    Dim t As Type() 'array to hold all the types (classes mostly)

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

        a = GetType(System.Data.DataRow).Assembly

        t = a.GetTypes

        Dim re As Integer = t.Length
        TextBox1.Text = 'There are " & re & types in " & a.FullName

        Dim i As Integer

        For i = 0 To re - 1
            ListBox1.Items.Add(i + 1 & ". " & t(i).ToString)
        Next i

    End Sub
```

Loading a File from an Assembly

To get a file within an assembly, you can use the technique illustrated in Listing 8.3. Here the contents of the file are displayed in a hex block.

LISTING 8.3: ACCESSING A FILE FROM INSIDE AN ASSEMBLY

```
Imports System.Reflection

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

```
Dim f As String

Dim a As Reflection.Assembly

a = Reflection.Assembly.GetAssembly(Me.GetType())

Dim St As IO.FileStream = _
    a.GetFiles()(0)

Dim L As Integer = St.Length

Dim Arr(L) As Byte

St.Read(Arr, 0, L)

Dim I As Integer

For I = 0 To L - 1
    f &= Hex(Arr(I))
Next

TextBox1.Text = f

End Sub
```

Loading an Assembly from a File

You can, of course, query the types within an assembly. The code example in Listing 8.4 illustrates two techniques:

- How to fill an array that holds all the classes (types) within an assembly, and display them
- How to load an assembly from a hard disk file (the previous examples in this chapter have accessed already-loaded assemblies that were part of the currently executing project)

Here you access the Microsoft.VisualBasic "compatibility namespace" (used to permit certain elements of traditional VB6 source code to work within a .NET project).

See the notes below concerning how to figure out the path to the assembly. You will likely need to replace the argument for the LoadFrom method with the path that works on your system—shown in the following code in boldface. The validation key can differ, and even the path might be different as well.

LISTING 8.4: VIEWING THE COMPATIBILITY NAMESPACE

```
Imports System.IO
Imports System.Reflection
Imports Microsoft.VisualBasic

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

    Dim cr As String = ControlChars.CrLf

    Dim a As System.Reflection.Assembly
    Dim t As Type() 'array to hold all the types (classes mostly)

    Try
        a = System.Reflection.Assembly.LoadFrom _
            ('c:\windows\assembly\gac\microsoft.visualbasic\ _
            7.0.5000.0__b03f5f7f11d50a3a\microsoft.visualbasic.dll")
        t = a.GetTypes
        Catch ex As FileNotFoundException
            TextBox1.Text = "Cannot find file for assembly"
            Return
        Catch ex As TypeLoadException
            TextBox1.Text = "Cannot load types"
            Return
        End Try

    Dim re As Integer = t.Length
    TextBox1.Text = "There are " & re & " types in " & a.FullName

    Dim i As Integer

    For i = 0 To re - 1
        TextBox1.Text &= i + 1 & "." & t(i).ToString & cr
    Next i

    TextBox1.SelectionLength = 0 'turn off the default selection
End Sub
```

The .NET assemblies are held in the `Windows\Assembly` folder (and elsewhere). However, you cannot directly access the assemblies by using a simple path string such as this:

```
System.Reflection.Assembly.LoadFrom("c:\windows\assembly\ microsoft.visualbasic
```

Instead, you must use a fully qualified path (which doesn't appear in Windows Explorer):

```
c:\windows\assembly\gac\microsoft.visualbasic\ _
7.0.5000.0__b03f5f7f11d50a3a\microsoft.visualbasic.dll
```

To get the exact format for any .NET assembly that you want to work with using the Assembly.LoadFrom method, you can usually add an Imports statement to a VB.NET project, press F5 to execute the project, then look for (and copy) the line that Invokes the assembly in the Output window.

When you run this project, you'll see the results displayed in Figure 8.1:

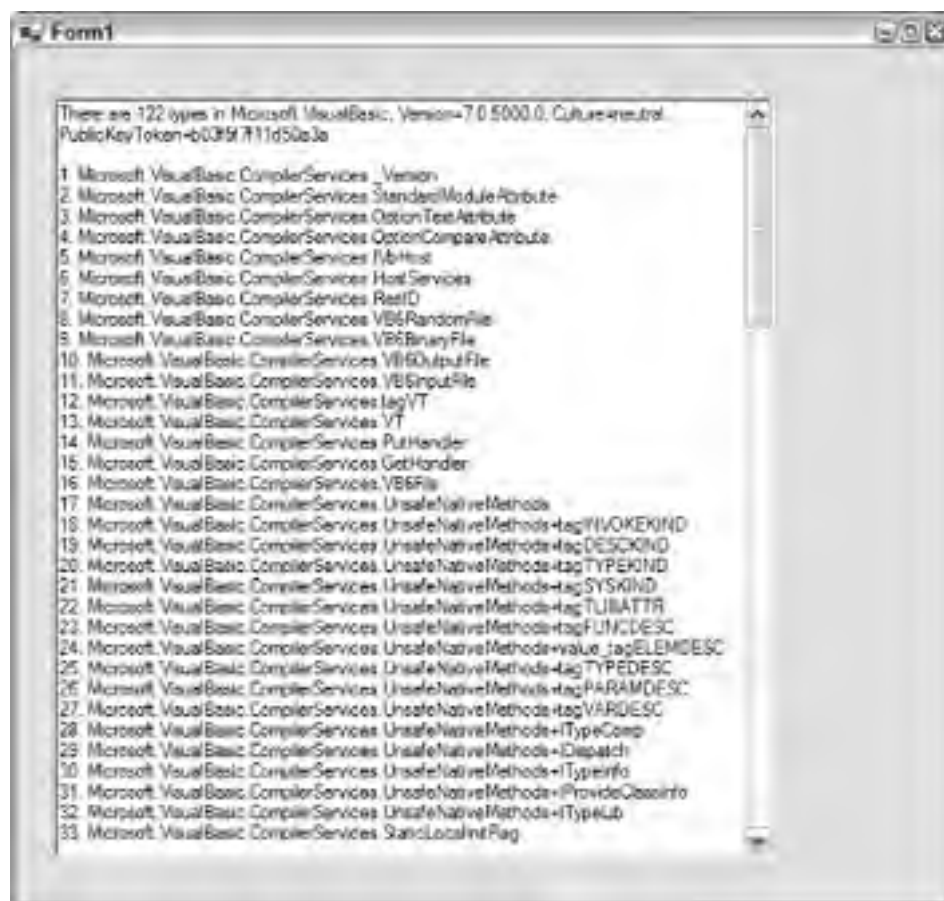


FIGURE 8.1 Getting a list of all the types in an assembly isn't difficult.

The LoadFrom method of an assembly object takes a string path as its argument, and loads the file. This is how you can access assemblies that are not currently loaded within the now-active VB.NET solution.

Getting the Methods in a Class

Modifying Listing 8.4, you can display all the methods within any given type using an enumerator. Add a ListBox to your form, then use the code in Listing 8.5. (Remember to replace the argument for the LoadFrom with the path that works on your system. The validation key can differ, and even the path might be different as well.)

LISTING 8.5: VIEWING A TYPE'S METHODS

```
Dim cr As String = ControlChars.CrLf

Dim a As System.Reflection.Assembly
Dim t As Type() 'array to hold all the types (classes mostly)

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

[Team Fly](#)

 Previous

Next 

```
Try
    a = System.Reflection.Assembly.LoadFrom('c:\windows\asse
microsoft.visualbasic\7.0.5000.0__b03f5f7f11d50a3a\microsoft.visualba
    t = a.GetTypes
Catch ex As FileNotFoundException
    TextBox1.Text = "Cannot find file for assembly"
    Return
Catch ex As TypeLoadException
    TextBox1.Text = "Cannot load types"
    Return
End Try

Dim re As Integer = t.Length
TextBox1.Text = "There are" & re & " types in " & a.FullName

Dim i As Integer

For i = 0 To re - 1
    ListBox1.Items.Add(i + 1 & ". " & t(i).ToString)
Next i

TextBox1.SelectionLength = 0 'turn off the default selection

End Sub

Private Sub ListBox1_SelectedIndexChanged(ByVal sender As System.
ByVal e As System.EventArgs) Handles ListBox1.SelectedIndexChanged
    TextBox1.Clear()

    TextBox1.Text = "Methods of " & t(ListBox1.SelectedIndex).ToS

    Dim Enumerator As IEnumerator = _
        t(ListBox1.SelectedIndex).GetMethods.GetEnumerator

    While (Enumerator.MoveNext)
        TextBox1.Text &= CType(Enumerator.Current, MethodInfo).Na
    End While

End Sub
```

When you run this example, click in the ListBox on the VisualBasic.Financial class to see all its methods. You probably recognize them from earlier versions of VB.

More about Types

The classification system you're used to as a .NET programmer (the nested classification: assembly/module/class/enum, for example) breaks down somewhat when you use reflection. An enum is considered a type, even when that enum resides within another type.

For example, if you create a module in your project that looks like this:

```
Module convertsomething

    Function ReturnMoney(ByVal x As String) As Decimal

        Return CDec(x)

    End Function

    Public Enum Weeks
        FirstWeek = 1
        SecondWeek = 2
        ThirdWeek = 3
        FourthWeek = 4
    End Enum

End Module
```

Then you use `GetType` to extract the types from within your entire project (the assembly containing your whole VB.NET solution):

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    Dim a As System.Reflection.Assembly
    Dim cr As String = ControlChars.CrLf

    a = GetType(Reflect1.convertsomething).Assembly

    Dim t As Type

    For Each t In a.GetTypes

        TextBox1.Text &= t.Name & cr

    Next

End Sub
```

You may be surprised to see that the enum, which appears to be on the same level (within the same container, the module) as the function `ReturnMoney`, isn't in fact on that level at all. It's considered on the same level as its container (the module `Convertsomething`) and the `Form1` class.

Here's the result you get when you iterate through the types in the solution assembly; the types discovered are the module, the enumeration within that module, and the form class:

```
convertsomething  
Weeks  
Form1
```

Accessing Specific Members

As you doubtless noticed, when you get an assembly's types (or members) using one of the Get methods (such as GetTypes or GetConstructors) you use an array to hold the information:

```
Dim cinfo As ConstructorInfo()
```

But is there a way to directly access a specific type or member from a class or assembly? You bet. During runtime, you may need a way to quickly access an array of reflection data (you could iterate through the array, but it's faster to use a direct access method—particularly if the assembly is huge and contains many hundreds of types, for example).

Assume you have a class named TestClass with two constructors, and you are interested in retrieving only the constructor that accepts two strings. You can pass an array with two strings to the GetConstructor method, as illustrated in this example. You can, of course, also use the other Get methods. This example (Listing 8.6) illustrates how to get a specific method from a class using GetMethod.

LISTING 8.6: DIRECTLY CONTACTING A TYPE

```
Private Sub Form1_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles MyBase.Load  
  
    Dim t() As Type = {GetType(String), GetType(String)}  
  
    Dim cInfo As ConstructorInfo = GetType(TestClass).GetConstructor(t)  
  
    MsgBox(cInfo.ToString)  
  
    Dim mInfo As MethodInfo =  
  
    GetType(TestClass).GetMethod("SendBak")  
  
    MsgBox(mInfo.ToString)  
  
End Sub  
  
End Class  
  
Public Class TestClass  
    Public Sub New(ByVal s As String)
```

```
End Sub

Public Sub New(ByVal s As String, ByVal s1 As String)

End Sub

Public Function SendBak() As String
End Function

End Class
```

Searching for Members or Data

You can search through a class's members, looking for specific information, such as the name of methods, fields, and such, or the data in a field.

In Listing 8.7, you access the data within the fields in a class. This technique can be used to locate a specific member or datum.

LISTING 8.7: LOCATING MEMBER DETAILS

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

    Dim cr As String = ControlChars.CrLf

    Dim mInfo As MemberInfo()

    Dim FInfo As FieldInfo

    Dim tc As New TestClass

    mInfo = tc.GetType.FindMembers(
        MemberTypes.Field, BindingFlags.NonPublic Or _
        BindingFlags.Instance, Nothing, Nothing)

    TextBox1.Text = "Data in fields within" & tc.ToString & cr

    For i As Int16 = 0 To mInfo.Length - 1

        FInfo = CType(mInfo(i), FieldInfo)

        TextBox1.Text &= FInfo.ToString & "contains this data: "
        TextBox1.Text &= FInfo.GetValue(tc) & cr

    Next
```

```
'illustrates how to permit the user to select from various me
'at runtime, then provide any necessary parameters
'and execute the chosen method

Label1.Text = ''Click one of the methods in the ListBox to se

'display the methods in the TestClass

Dim Enumerator As IEnumerator = t.GetMethods.GetEnumerator
While (Enumerator.MoveNext)
    ListBox1.Items.Add(CType(Enumerator.Current, MethodInfo).
End While

End Sub

Private Sub ListBox1_SelectedIndexChanged(ByVal sender As System.
ByVal e As System.EventArgs) Handles ListBox1.SelectedIndexChanged

mInfo = t.GetMethod(ListBox1.SelectedItem.ToString)

paramInfo = mInfo.GetParameters() 'get the parameters of the
l = paramInfo.Length - 1

Dim m As String = "The paramters you must pass to this meth

For j As Int16 = 0 To l
    m &= paramInfo(j).ParameterType.ToString & ", "
Next j

Label1.Text = "This method requires" & l + 1 & _
" parameter(s) which you must supply. " & m & _
" so, please type the parameter(s) into the TextBox," & _
separated by commas. When finished, click the button.

End Sub

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

```
'Provide all parameters
Dim p(paramInfo.GetUpperBound(0)) 'set p to contain paraminfo

Dim delimStr As String = "','" : Dim delimiter As _
Char() = delimStr.ToCharArray()

Dim tt As String = TextBox1.Text
Dim split As String()
split = tt.Split(delimiter)

Dim C As Integer
Dim s As String

For Each s In split

    'figure out parameter's variable type
    Select Case paramInfo(C).ParameterType.Name

        Case "String" 'case sensitive

            p(C) = split(C)

        Case "Int32"

            Dim NewInt As Integer = Integer.Parse(split(C)) _
'turn string into integer

            p(C) = NewInt

    End Select

    C += 1
Next s

mInfo.Invoke(obj, p)

End Sub
End Class

Public Class TestClass

    Public Sub narz(ByVal YourFirstName As String, ByVal YourLastName
        MsgBox("Hi!" & YourFirstName & " " & YourLastName)
```

```
End Sub

Public Sub AddFive(ByVal YourAge As Integer)

    MsgBox('In five years you will be ' & YourAge + 5)

End Sub

End Class
```

Right off the bat you reflect a type variable (a class named TestClass) and instantiate that class in the variable obj. You use the CreateInstance method of the Activator class:

```
Dim t As Type = GetType(TestClass)
Dim obj As Object = Activator.CreateInstance(t)
```

This happens to be a local assembly that is actually running while the CreateInstance method instantiates a TestClass object. However, if you want to instantiate a COM object from an assembly in a file, you use the CreateComInstanceFrom method. Or, to create a .NET object from an assembly in a file, use the CreateInstanceFrom method.

In addition to defining obj as an instance of the TestClass class, you create several other form-level variables so they will have the scope to be accessed from the Form_Load event as well as button and ListBox events.

Code in the Form_Load event sets things up for the user. You fill the ListBox with the name of each method in the type t. You access the enumerator of the GetMethods method, then use it to step through the methods and list their names in the ListBox.

In this chapter, however, I've used several looping techniques to manipulate or extract reflected data: For...Next based on a Length property, For...Each based on MemberInfo objects:

```
Dim m As MemberInfo
For Each m In cinfo
```

For...Each based on the collection of types within an assembly:

```
Dim t As Type
For Each t In a.GetTypes
```

In any case, Microsoft currently advises using enumerators for looping, whenever possible.

After the ListBox is filled, the user clicks one of the methods listed there and its parameters (their number and data types) are described in the label.

As soon as a method is clicked, you use the GetMethod method to assign the selected method from the t class and store it in a methodinfo class.

```
mInfo = t.GetMethod(ListBox1.SelectedItem.ToString)
```


Then you extract metadata (all the parameters in this case) from the selected method. These parameters are held in the paramInfo array:

```
paramInfo = mInfo.GetParameters() 'get the parameters of the method
```

The length of this array (less one, as is so often necessary) is assigned to the variable `l`. It will be used in several loops in this project. You then create a string to inform the user how many and of what type the necessary parameters are.

After the user types the parameter or parameters into the TextBox, he or she clicks the button. Now you must parse the parameters that the user typed. Before VB6 this required the usual parsing tedium—having to adjust the start and length information that you provide to the SubString method. You had to fiddle with parameters and multiple strings, like this:

```
Dim c, c1 As Integer, tArray(l) As String
Dim tt As String = TextBox1.Text
Dim tz As String = tt
For i As Integer = 0 To l
    c1 = tt.IndexOf("'",")
    If c1 = -1 Then tArray(i) = tt : Exit For
    tz = tt.Substring(c1 + 1, tt.Length - c1 - 1)
    tArray(i) = tt.Substring(c, c1)
    tt = tz
Next i
```

Over the years I've wished that VB had a parse function to which you passed a string, and a delimiter (such as `"'"`), and that the function returned an array of substrings. Fortunately, in VB6, the Split method was added to the string object. Now, instead of the messy code above, you can use this much simplified version of parsing:

```
Dim delimStr As String = "'" : Dim delimiter As Char() _
= delimStr.ToCharArray()
Dim tt As String = TextBox1.Text
Dim split As String()
split = tt.Split(delimiter)
```

Then you use `Select...Case` to figure out the data types of each parameter in the parameter array. Reflection presents you with an unusual job: you may have to translate a string data type into other data types. Normally, you think of coercing (or casting, as they prefer to call it) such as using `CInt`.

Another related issue is that of debugging applications. No amount of error-handling code will do you any good if your application is producing wrong results, or contains syntax errors. Errors, or bugs, in our code are not always obvious, and any non-trivial application contains quite a few of them—just check out the number of "fixes" and "patches" for major commercial applications. Debugging is the most important part of coding an application, and all modern development environments, such as Visual Studio, provide numerous tools to assist you in locating problems in your code and fixing them. The bulk of this chapter is devoted to the debugging tools of Visual Studio.

Structured Exception Handling

A good deal of the code we write handles errors. It shouldn't come as a surprise, but more than half of a professional application's code validates data and handles possible errors. Most of the errorhandling code we write will never be executed. Users aren't expected to enter a discount percentage that exceeds 100%, or a future birth date. This isn't supposed to happen and it may never happen. However, you must validate the discount's value from within your code and not proceed with your calculations until the user supplies a valid value. Or, even worse, attempt to calculate the age of a person with a future birth date. We should make a very important distinction here. A valid value is not necessarily the `correct` value, but there's nothing you can do about that.

***NOTE** Exception is the latest term for errors. For years, we used to fix errors in our code and handle runtime errors. Obviously, error is not the best term for describe something that occurs in our code, so a new, less embarrassing term was invented.*

Consider an application that performs static calculations. The individual parameters supplied by the user may be correct, but when they're combined, the calculations may fail (i.e., the calculations may produce results that don't make sense). No parameter is in error; they're incompatible with one another. For example, they may result in a division by zero, the calculation of the square root of a negative value, and so on. Our task is to detect from within our code any incompatibilities, or anomalies, in the data and allow users to revise it, rather than allow the application to crash. The situation we just described can't be handled with data validation. We'll discover a problem with our data only after we attempt to use it in some calculations.

Another type of error you can't prevent with data validation are those caused by the hardware itself. The disk may be full when you attempt to save a large file, or the drive you're accessing may be disconnected. When your application runs into a situation like this, it should be able to detect the error and handle it. To handle errors that surface at runtime, we use *structured exception handlers*. Exceptions that are handled from within the application's code are called *handled exceptions* and they result in robust applications. Exceptions that are not handled from within your code are called *unhandled exceptions* and they lead to program crashes (in effect,

it's the CLR that handled these errors in a rather crude manner). Figure 9.1 shows the result of an unhandled exception. After you close this message, the application will terminate. Figure 9.2 shows a message box displayed from within an error handler. Any user can understand this error message and the program will most likely give the user a chance to specify a different path, instead of crashing. This type of error can be easily avoided by using a File Open dialog box, which forces the user to select an existing file, but you get the idea.

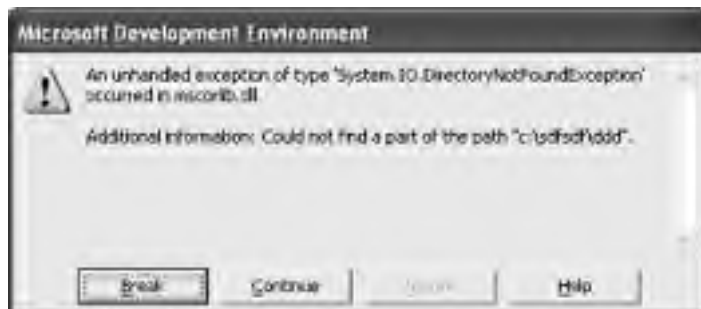


FIGURE 9.1 Unhandled exceptions lead to program crashes.

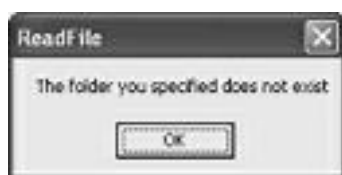


FIGURE 9.2 A robust application handles exceptions from within its code.

A *structured exception handler* contains two sections of code: the actual code that will always work under perfect conditions (the strictly "application code," so to speak) and another section of code that will be activated if something goes wrong in the first section. First, you must identify the sections of code that may throw an exception. The code that opens a file, for example, is a prime candidate for error handling. A file on a CD can't be opened for writing, and a file on a network drive may not be available at all times. These are error situations that can't be prevented. All you can do is capture the error the moment it occurs and keep it from crashing the application. To capture an error, we embed sections of our code in a structured exception handler, which has the following form:

```
Try
    'Code that may cause a run-time error
Catch ex As Exception
    'Code to handle the error
Finally
    'Code to do any final clean up.
End Try
```

Only the Try and End Try statements are mandatory, but you'll rarely write code that doesn't use the Catch statement. The Catch statement accepts as argument an object that represents an exception. This argument can be a specific exception (like an overflow exception) or a generic exception. The simplest form of the statement is the following, which catches all possible exceptions and handles them with the same handler:

```
Try
    'one or more statements
Catch ex As Exception
    MsgBox("An error occurred, program will terminate!" & _
        vbCrLf & ex.Message)
End
End Try
```

The error handler shown here terminates the application. At the very least, you must prompt the user to save any open document. We usually terminate the execution of the procedure that contains the exception, but we don't terminate the application.

[Team Fly](#)

 Previous

Next 

All objects that represent exceptions derive from the `Exception` class, and the .NET Framework provides many types of exception objects. We'll look at the `Exception` object and the most common types of exceptions in the following section. In this section we're going to use the `Exception` class's `Message` property, which returns a description of the error.

You can also include multiple `Catch` statements to catch, and handle, different types of errors. A section of code that performs math calculations may generate an overflow exception. While an overflow is the most likely type of exception for a code segment that performs math calculations, we can't ignore all other types of exceptions. The following structured exception handler shows how to catch specific exceptions and handle them individually, as well as how to handle all other types of exceptions:

```
Try
    ' statements
    ValidResult = True
Catch OfException As OverflowException
    MsgBox("An overflow occurred")
    ValidResult = False
Catch Ex As Exception
    MsgBox(ex.Message)
    ValidResult = False
End Try
```

The code sets the *ValidResult* Boolean variable to a True/False value to indicate whether the operations completed successfully or not. When you use multiple `Catch` blocks in your code, you must order them from the most specific to the least specific. If you don't, the most general (less specific) exception will be caught first and the code that handles the more specific exceptions will never be executed. A code segment that writes to a file may run into many different exceptions. Some of the errors you should handle individually are the following:

- ◆ File can't be written to
- ◆ User has no permission to write to the file
- ◆ May run out of disk space
- ◆ Other generic errors (disk failure, removal of the media, and so on).

You will see shortly the code that opens a file for reading and writing and how it handles all errors. Another form of the `Catch` clause allows you to specify conditions, which limits its scope and allows us to handle exceptions conditionally: if the specified condition isn't met, the `Catch` clause isn't activated. If you want to handle an exception differently when the value of a variable is negative than when the value of the same variable is zero or positive, use an exception handler with the following structure:

```
Try
    ' statements
Catch Ex As Exception When runningTotal < 0
    ' handle exception for negative values
Catch Ex As Exception
    ' handle same exception for all other values
End Try
```


The Finally Clause

The statements in the Finally clause are executed regardless of which of the two blocks of the Try statement, the Try or Catch block, was executed. This is where we insert any clean-up code. The following statements execute a command against a database. The *CMD* object represents an SQL command and the *CN* object represents a connection to the database that has already been established successfully. The connection must be closed regardless of whether the command was executed successfully or not. If you close the connection in the Catch clause, the connection will not be closed if the command is executed successfully. If you insert another call to the Close method at the end of the structured exception handler, you may get another error, outside the handler. If the connection is closed in the Catch clause and you attempt to close it again after the exception handler, you'll get an error to the effect that you're attempting to close a connection that's already closed. By inserting the call to the Close method in the Finally block of the statement, we make sure that the connection is always closed. Notice that the code examines the Connection object's State property to make sure it doesn't attempt to close an already closed connection, because this would throw another exception.

```
Try
    CMD.ExecuteNonQuery
Catch Ex As Exception
    MsgBox Ex.Message
Finally
    If Not cn.State = ConnectionState.Closed then
        CN.Close
    End If
End Try
```

The ReadWriteFile Project

In this section we'll implement two structured error handlers to deal with runtime errors that may occur while opening, reading from, and writing to files. First, we must determine the operations that may throw exceptions at runtime. We'll use the Open method of the File class to open a file for reading. The file's name will be supplied by the user through the FileOpen common dialog box. If you look up the Open method in the documentation, you'll find a list of all the exceptions this method may raise; they're listed in Table 9.1.

TABLE 9.1: THE EXCEPTIONS OF THE OPEN METHOD

| EXCEPTION | CONDITION |
|-----------------------------|--|
| ArgumentException | The path argument is a zero-length string, contains only white space, or contains one or more invalid characters as defined by InvalidPathChars. |
| ArgumentNullException | The path argument is a null reference (Nothing in Visual Basic). |
| ArgumentOutOfRangeException | The value of the <i>mode</i> argument is invalid. |
| PathTooLongException | The specified path, filename, or both exceed the system- |

defined maximum length. For example, on Windows-based platforms, paths must be fewer than 248 characters, and filenames must be fewer than 260 characters.

| | |
|-----------------------------|--|
| IOException | An I/O error occurred while opening the file. |
| DirectoryNotFoundException | The specified path is invalid, such as being on an unmapped drive. |
| UnauthorizedAccessException | The path specified is a directory, or the calling application does not have the required permission to write to, or read from, the file. |
| FileNotFoundException | The file specified in path was not found. |
| NotSupportedException | The path is in an invalid format. |

Some of these errors can be easily prevented; we can easily make sure that the path is not an empty string and the file is opened in a valid mode. Since we're using the `OpenFile` built-in dialog box, these errors will never occur. However, another application might read the file's name from another source (such as a text file) and it may run into a bad file/path name.

The statements in the `Try` clause of the sample code in Listing 9.1 attempt to open a file for reading. If any of the `Open` method's possible exceptions occurs, the program displays a warning and exits the procedure, without opening the file.

LISTING 9.1: HANDLING EXCEPTIONS WITH THE BASIC FILE I/O OPERATIONS (1)

```
Private Sub btnFileIOExceptions_Click( _  
    ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) _  
    Handles btnFileIOExceptions.Click  
  
    Dim FName As String  
    Dim txt As String  
    OpenFileDialog1.CheckFileExists = True  
    OpenFileDialog1.CheckPathExists = True  
    OpenFileDialog1.Filter = "Text Files|*.txt"  
    Dim stream As FileStream  
    If OpenFileDialog1.ShowDialog = DialogResult.OK Then  
        Try  
            'Dim ex As New System.IO.DirectoryNotFoundException  
            'Throw ex  
            FName = OpenFileDialog1.FileName  
            stream = File.Open(FName, FileMode.Open)  
            Catch authorizationException As UnauthorizedAccessException  
                MsgBox("The file is read-only")  
                Exit Sub  
            Catch excptnDirectory As System.IO.DirectoryNotFoundException  
                MsgBox("Specified directory not found")  
                Exit Sub  
        End Try  
    End If  
End Sub
```



```
    Catch excptnFile As System.IO.FileNotFoundException
        MsgBox("'Specified file not found")
        Exit Sub
    Catch IOExcptn As System.IO.IOException
        MsgBox("Couldn't open the file")
        Exit Sub
    Catch excptn As Exception
        MsgBox("Failed to open file." & vbCrLf & excptn.Message)
        Exit Sub
End Try
Dim b As Byte
Dim buffer(stream.Length - 1) As Byte
Try
    stream.Read(buffer, 0, stream.Length)
    Catch excptnIO As System.IO.IOException
        MsgBox("Error in reading file")
        Exit Sub
    Catch excptnNotSupported As System.NotSupportedException
        MsgBox("Can't read from file")
        Exit Sub
    Catch excptn As Exception
        MsgBox("The following error occurred while reading" & _
            vbCrLf & Excptn.Message)
Finally
    stream.Close()
End Try
txt = UTF7.GetString(buffer)
Console.WriteLine(txt)
End If
End Sub
```

All errors are fatal and we exit the subroutine. In the following section, we'll discuss a technique for repeating operations that failed.

If the file is opened successfully, the program attempts to read its contents into an array of bytes using the `FileStream` object's `Read` method. This method may throw several exceptions, which are listed in the `Read` method's entry in the documentation. The two most likely ones are the `IOException` and the `NotSupportedException`. The `IOException` exception occurs when the operating system can't read from the file (either because it's locked by another user or because the media is bad) and the `NotSupportedException` exception occurs when we attempt to perform an illegal operation on the file (such as moving to the beginning of a forward-only stream). The structured error handler handles these two exceptions separately, and then it handles all other exceptions. In the `Finally` clause we close the `Stream`.

The first two statements in the `Try` clause are commented out. You can insert statements to throw exceptions in your code to test your error handler. Declare variables to represent an exception type and use them to throw any exception with the `Throw` method.

The code for writing to a file is a little more challenging. The user may select a read-only file, and this is an exception we can handle from within our handler. If you attempt to open a read-only file for writing with the Open method, the UnauthorizedAccessException exception is thrown. In this exception's handler, you can verify that the file is read-only and prompt the user to reset the file's read-only attribute. If the user agrees to change the read-only attribute, the code executes the Open method again. Notice the nested exception handler embedded in the UnauthorizedAccessException Catch clause. This handler takes care of any exceptions thrown by the statements that call the SetAttributes and Open methods. Listing 9.2 shows the complete code for a very simple operation. As you can see, the core of the code consists of a few statements (the statements that open the file and write a string to it). The remaining statements handle possible errors and make the program easier to use.

LISTING 9.2: HANDLING EXCEPTIONS WITH THE BASIC FILE I/O OPERATIONS (2)

```
Private Sub Button2_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
    Handles Button2.Click

    Dim path As String
    Dim FS As FileStream
    Dim cException As ArgumentNullException
    Dim RepeatOperation As Boolean = True
    While RepeatOperation
        RepeatOperation = False
        If SaveFileDialog1.ShowDialog <> DialogResult.OK Then
            Exit Sub
        End If
        Me.Refresh()
        path = SaveFileDialog1.FileName
        Try
            FS = File.Open(path, FileMode.OpenOrCreate)
        Catch exPath As PathTooLongException
            MsgBox("'Invalid path name")
            RepeatOperation = True
        Catch exPath As DirectoryNotFoundException
            MsgBox("The folder you specified does not exist")
            RepeatOperation = True
        Catch exFile As FileNotFoundException
            MsgBox("The file you specified does not exist")
            RepeatOperation = True
        Catch exArgumentNull As ArgumentNullException
            MsgBox("You have not specified the file to open")
            RepeatOperation = True
        Catch AccessException As UnauthorizedAccessException
            Dim reply As DialogResult
            If File.GetAttributes(path) And _
                FileAttributes.ReadOnly = FileAttributes.ReadOnly Then
                reply = MsgBox("File is read-only. Reset it?", _
                    MsgBoxStyle.YesNo)
            End If
        End Try
    End While
End Sub
```

```
        If reply = DialogResult.Yes Then
            Try
                File.SetAttributes(path, _
                    File.GetAttributes(path) And _
                    (Not FileAttributes.ReadOnly))
                FS = File.Open(path, FileMode.OpenOrCreate)
            Catch ex As Exception
                MsgBox('Could not reset read-
only attribute!")
                Exit Sub
            End Try
        End If
    Else
        MsgBox("Can t access file!")
        Exit Sub
    End If
Catch GeneralException As Exception
    MsgBox(GeneralException.Message)
Exit Sub
End Try
End While

Dim b(1024) As Byte
Dim temp As UTF8Encoding = New UTF8Encoding(True)
Dim buffer() As Byte
Try
    buffer = System.Text.Encoding.UTF8.GetBytes(
        ("Write this string to the file")
    FS.Write(buffer, 0, buffer.GetLength(0))
Catch exUnsupported As NotSupportedException
    MsgBox("The stream doesn't supported the requested operation")
Exit Sub
Catch IOExc As IOException
    MsgBox("Error writing to file." & vbCrLf & _
        "Please make sure the file isn't " & _
        "read-only and the disk isn't full")
Catch GeneralExc As Exception
    MsgBox("Error in application")
Exit Sub
Finally
    FS.Close()
End Try
MsgBox("Data saved successfully")
End Sub
```

Our error-handling code doesn't do a lot, except for displaying specific error messages that will help the user understand the condition that prevented the successful completion of the operation. However, the Catch clauses can be as complicated as you can make them.

To read the attributes of a file, use the `GetAttributes` method; to set an attribute, use the `SetAttributes` method of the `File` class. Both methods may throw exceptions, which you should handle with a nested error handler. In the `UnauthorizedAccessException` handler's code we attempt to reset a read-only file. If the operation succeeds, we repeat the statements that write a string to the file. If the file's read-only attribute can't be reset (this will be the case for a file on a CD), the subroutine is terminated.

Resuming Statements that Failed

The unstructured error-handling techniques of VB6 are supported by VB.NET, but you shouldn't use them. The error-handling mechanism of VB6 relied on the `GoTo` statement, which is considered bad technique. Use too many error handlers in a procedure and you won't be able to read your own code. There's one feature of the `On Error` statement you'll miss, however, and this is the `Resume` statement. In an error handler, you can attempt to fix the error that prevented a statement from completing successfully and then repeat the offending statement by calling the `Resume` statement. If a statement that attempts to write to a file fails because the media is read-only (a CD drive, for example), you can display a message to the user, give the user a chance to select another path, and repeat the statement.

Structured exception handlers don't provide an equivalent mechanism, so you'll have to resort to the `GoTo` statement. Let's consider a code segment that attempts to open a connection to a database. The operation may fail because the client computer has no access to the database server, or because the connection string is invalid. If you can't establish a connection to the database, you should probably terminate the application. You can't expect an end user to supply a valid connection string, but let's look at a technique that allows us to repeat a statement that failed to execute.

The following code segment attempts to open a connection. If the operation fails, the code prompts the user as to whether it should repeat the operation. If the user chooses to repeat the operation, the program prompts for a new connection string. End users shouldn't be allowed to edit connection strings, but you can give your users a chance to check their computer's connection to the local network, or otherwise troubleshoot the problem. There's not much you can do from within your application's code, but you may wish to repeat the operation, rather than terminating the application.

```
repeatOperation:
Try
    CN.ConnectionString = txtConnString.Text
    CN.Open()
Catch ex As ArgumentException
    Dim reply As MsgBoxResult
    reply = MsgBox(''Could not open Connection" & vbCrLf & _
        ex.Message & vbCrLf & "Retry?", MsgBoxStyle.OKCancel)
    If reply = MsgBoxResult.OK Then
        Dim ConnString As String = _
            InputBox("Please enter a different" & _
                "connection string", , txtConnString.Text)
        txtConnString.Text = ConnString
        GoTo repeatOperation
    Else
```



```
        Throw ex
    End If
Catch ex As SqlClient.SqlException
    Throw New Exception(ex.Message)
End Try
```

The setting of the `ConnectionString` property is read from a `TextBox` control. The code catches the `ArgumentException` exception, which is thrown when the setting is incorrect, and then prompts the user about the action to be taken. If the user decides to repeat the operation, the code will execute the `GoTo` statement to transfer control to the `Try` statement. Should the operation fail again, the process is repeated.

If the user can't troubleshoot his or her connection to the database server, the program throws an exception, which must be intercepted and handled by the calling procedure. The most reasonable course of action for this type of problem is to terminate the execution of the application.

The Exception Class

The `Exception` object represents an exception; there are different exception classes for different exceptions, all deriving from the `Exception` class. Each exception object provides information about a specific exception. The `Message` property is a string with the exception's description. The `InnerException` property is an `Exception` object that represents an exception thrown while an error handler was in effect. If an exception occurs in an exception handler's code, the `Message` property describes the current error, while the `InnerException` property represents the error that activated the error handler. The `Source` property is another string with the name of the object that caused the exception or the name of the assembly where the exception occurred. Finally, the `StackTrace` property holds a stack trace, which is a list of all the called methods preceding the exception and the line numbers in the source file(s). The `TargetSite` property returns the name of the method that threw the current exception.

AN OUNCE OF PREVENTION

Preventing errors is even better than catching them. Sometimes we can eliminate all sources of error by validating the data we'll use in our calculations. Let's say you're calculating loan payments, which depend on the loan's amount, the interest rate, and the loan's duration. Instead of embedding all the calculations in a structured exception handler, you can examine the values of the loan's parameter `loan` before you perform the calculations. If you make sure that the interest rate is a value between 1 and 20 (or 0.01 and 0.2, depending on how the interest rate should be expressed) and that the loan's amount and duration are positive values of a reasonable size, then you can perform the calculations. Data validation is extremely important and you should always validate the data you're going to process. Sometimes invalid data may not cause runtime exceptions, but they will certainly produce incorrect results. Your loan payment calculation routine may very well accept a negative interest rate, but what does this really mean? No bank will ever pay you to get a loan. If you validate your data, you can display descriptive error messages to the user and help them correct their mistakes as early as possible. A structured exception handler may display a very generic message, but your code that validates individual values will provide very specific descriptions for all possible errors.

If the exception isn't handled in the subroutine where it occurred, you can still recover the line that caused the exception through the `StackTrace` property. The `StackTrace` property is a long string that contains the chain of procedure calls all the way from the start of the application to the procedure that threw the exception. A typical value of the `StackTrace` property is the following:

```
at Exceptions.Form1.Proc2() in
    C:\Toolkit\CH09\Exceptions\Form1.vb:line 168
at Exceptions.Form1.Proc1() in
    C:\Toolkit\CH09\Exceptions\Form1.vb:line 147
at Exceptions.Form1.Button1_Click(Object sender, EventArgs e) in
    C:\Toolkit\CH09\Exceptions\Form1.vb:line 139
```

This information isn't of much use to your code; you can view the chain of procedure calls to the offending procedure in the Call Stack window, which is discussed shortly. However, it can be of great help in troubleshooting applications that have already been deployed. You can dump this information to a file before you quit the application and use this file's contents as your starting point when you're called to service your application at the customer's site.

To better handle exceptions from within your code, you should catch specific exceptions whenever possible, rather than a generic exception. But how do we know in advance all possible exceptions that a certain statement may cause? When an unhandled exception is thrown, a message box with the exception's description pops up. The description includes the type of the exception (`System.OverflowException`, or `ArgumentException`, and so on). You can force the most common exceptions and see their types. You can also look up the names of the methods you call in the Try block in the online documentation and find out the types of exceptions each method can cause. Your effort should be aimed toward the elimination of conditions that will lead to an exception. When this isn't possible, you should catch different types of exceptions and handle them separately. The following are some of the most common exceptions, which are represented by individual objects that derive from the `Exception` class. We list first a general class and its descriptions, followed by more specific classes that represent specific exceptions of the same type. An `ArgumentException` exception is thrown every time you call a method with an argument that doesn't match the argument list of the method. An exception of this type may be caused because one of the arguments is null (Nothing in VB), because you've specified more arguments than the method accepts, or because you've specified an enumeration member that doesn't exist. These three exceptions are represented by the classes listed in the Derives classes under the `ArgumentException` entry.

ArgumentException Represents the exceptions that occur when one or more of the arguments passed to a method is not valid.

Derived classes `ArgumentNullException`, `ArgumentOutOfRangeException`,
`ComponentModel.InvalidEnumArgumentException`

ArithmeticException Represents errors resulting from invalid arithmetic, casting, or conversion operations.

Derived classes `DivideByZeroException`, `NotFiniteNumberException`,
`OverflowException`

ArrayTypeMismatchException Represents the exception thrown when you

attempt to store a value of the wrong type to an array element.

[Team Fly](#)

 Previous

Next 

Data.DataException Represents exceptions generated by the ADO.NET components—the `ReadOnlyException` exception, for example, which derives from the `DataException` class and represents the exception that's thrown when a statement attempts to set the value of a read-only field.

Derived classes `Data.ConstraintException`,
`Data.DeletedRowInaccessibleException`, `Data.DuplicateNameException`,
`Data.InRowChangingEventException`, `Data.InvalidConstraintException`,
`Data.InvalidExpressionException`, `Data.MissingPrimaryKeyException`,
`Data.NotNullAllowedException`, `Data.ReadOnlyException`,
`Data.RowNotInTableException`, `Data.StringTypingException`,
`Data.TypedDataSetGeneratorException`, `Data.VersionNotFoundException`

Data.DBCConcurrencyException Represents exceptions that occur during update operations, due to concurrency violations.

Data.SqlClient.SqlException Represents the exceptions returned by SQL Server during the execution of a query or stored procedure. You should catch the `SqlException` exception when you call one of the `Command` class's `Execute` methods.

Data.SqlTypes.SqlTypeException Represents exceptions that occur when you attempt to assign a value of the wrong type to a field or parameter.

Drawing.Printing.InvalidPrinterException Represents exceptions that occur when you attempt to access a printer using invalid settings.

InvalidCastException Represents exceptions that occur during invalid casting or conversion operations.

IO.InternalBufferOverflowException Represents the exception that occurs when a file buffer overflows.

IO.IOException Represents an I/O exception. There are several specific I/O errors, which are represented by classes deriving from the `IOException` class.

Derived classes `IO.DirectoryNotFoundException`, `IO.EndOfStreamException`
`IO.FileLoadException`, `IO.FileNotFoundException` `IO.PathTooLongException`

MemberAccessException Represents an exception that occurs when you attempt to access a class member that doesn't exist.

Derived classes `FieldAccessException`, `MethodAccessException`,
`MissingFieldException`, `MissingMemberException`, `MissingMethodException`

RankException Represents the exception that occurs when you pass an array with the wrong number of dimensions to a method.

Runtime.Serialization.SerializationException Represents exceptions that occur during the serialization or deserialization process.

Security.Cryptography.CryptographicException Represents exceptions that occur during cryptographic operations.

Security.XmlSyntaxException Represents an exception that's thrown when parsing an XML document that contains syntax errors.

StackOverflowException Represents the exception that's thrown when the execution stack overflows because of too many nested method calls (usually in recursive procedures).

An application should also include a handler for all exceptions that aren't handled in their respective procedures. Since unhandled exceptions propagate upward in the call stack, you can catch them all at the beginning statements of the application. To include an exception handler for all unhandled exceptions, start the application with the Run method, as shown in the following code segment:

```
Sub Main()  
    Try  
        System.Windows.Forms.Application.Run(New Form1())  
    Catch ex As SqlClient.SqlException  
        MsgBox('SQL server responded with the following message:" & _  
            ControlChars.CrLf & ex.Message & ControlChars.CrLf & _  
            "Application will terminate!", _  
            MsgBoxStyle.Exclamation, "Error!")  
    Catch ex As Exception  
        MsgBox("APPLICATION ERROR ! " & vbCrLf & _  
            ex.Message & vbCrLf & ex.StackTrace())  
    End Try  
End Sub
```

The code shown here handles two types of exceptions: the ones thrown by SQL Server (the `SqlClient` class) and general exceptions. When exceptions are caught at this level, it's too late to do anything about them, so you should include error handlers in your procedures, as close to the source of the error as possible.

The generic exception handler shown here can be used as a logging tool. For example, you can dump the call stack to a file and then examine this file to find out where the exception occurred and the exception's type.

Throwing Custom Exceptions

While it's relatively easy to handle errors in an application's code (after all, your code can interact with the user), things are very different when you write your own components. When you write a class, for example, you may run into a situation that can't be handled from within the class's code. In this case, you must throw an exception from within your class's code and let the calling application handle it. If the arguments passed to a function that performs a series of calculations, for example, are invalid, there's not much we can do in our code. We can't interact with the user, because the component may be running on a remote machine or a web server. What good are error messages displayed on the application server's monitor? They will remain there indefinitely, since no user will see them.

The solution is to throw an exception from within our code. The arguments themselves need not be invalid; just their combination results in an impossible situation. Consider a class that exposes a number of properties that must be set before calling a method that acts on these properties. If the calling application attempts to call the method without setting the properties first, the method must pass some information back to the calling application. The most robust technique of passing error

information back to the calling application is to throw an exception. As you will see, you can throw a generic exception or a custom exception. A custom exception is an object that inherits from the `ApplicationException` object and may convey additional information to the calling application, besides an error message.

To raise an exception in a class, use the `Throw` method followed by an `Exception` object. The simplest method of passing an `Exception` object to the application that uses your custom class is to create an `Exception` object by calling its constructor. The `Exception` object's constructor accepts as argument a string, which is a description of the error. The following statement will cause an exception when executed:

```
Throw New Exception("Age can't be negative")
```

The description of the error should be as specific as possible, and different conditions should produce different errors.

Consider the `BDate` property, which stores a person's birth date. Obviously, the birth date can't be a future date, so we insert some validation code in the `Property` procedure's code:

```
Public Property BDate() As Date
    Get
        Return _BDate
    End Get
    Set(ByVal Value As Date)
        If DateDiff(DateInterval.Year, Now, value) > 0 And _
            DateDiff(DateInterval.Year, Now, _PersonBDate) < 100 Then
            _BDate = Value
        Else
            Throw New Exception("Invalid date of birth")
        End If
    End Set
End Property
```

You can also define your own exceptions with a class that inherits from the `ApplicationException` class. Your class should contain three constructors—one without arguments, one with a description, and one with a message and another exception object. Let's implement a custom exception for the `BDate` property, the `AgeException` class. Listing 9.3 shows the implementation of the `AgeException` class.

LISTING 9.3: DEFINING A CUSTOM EXCEPTION

```
Public Class AgeException
    Inherits ApplicationException

    Public Sub New()
    End Sub

    Public Sub New(ByVal message As String)
        MyBase.New(message)
    End Sub
```

```
Public Sub New(ByVal message As String, ByVal inner As Exception)
    MyBase.New(message, inner)
End Sub
End Class
```

Now we can revise our BDate property's code and raise an exception of the AgeException type, when an attempt is made to set this property to a future date, as shown in Listing 9.4.

LISTING 9.4: THROWING A CUSTOM EXCEPTION

```
Public Property BDate() As Date
Get
    Return _BDate
End Get

Set(ByVal Value As Date)
    If DateDiff(DateInterval.Year, Now, value) > 0 And _
        DateDiff(DateInterval.Year, Now, _PersonBDate) < 100 Then
        _BDate = Value
    Else
        Throw New AgeException("'Invalid date of birth")
    End If
End Set
```

The last constructor of the custom Exception class allows you to pass the exception raised in the class to the calling application. The following structured exception handler may appear in a class's code and catches any error that occurs in the Try block. Instead of handling the error in your class's code, you can pass the exception to the calling application by passing it as argument to the CustomException class's constructor:

```
Try
    'statements
Catch Exc As Exception
    Throw New customException("Error in XXX class", Exc)
End Try
```

Bypassing Error Handlers

Structured error handlers are powerful tools in designing robust applications, but they introduce problems in debugging code. When an exception occurs, the handler takes over and you don't know the exact statement that caused the exception (unless the exception handler includes a single executable statement, which isn't very common). You can configure the Debugger to break on all errors, even if they're handled by a Catch statement. Certain exceptions may be handled silently, and you won't even know that they occurred.

To configure the Debugger, open the Debug menu and select the Exceptions command to see the Exceptions dialog box, which is shown in Figure 9.3. As a VB.NET developer, you're interested in

VB projects can be executed in one of two modes, as you already know—Debug mode and Release mode. When an application is compiled in Debug mode, the executable contains debugging information and is not optimized for speed. When you're ready to release the project, you can switch from Debug to Release mode and recompile. An executable compiled for release contains no debugging code and is optimized for speed. To change the execution mode, open the project's Property Pages and select the Configuration tab under Configuration Properties, as shown in Figure 9.4.

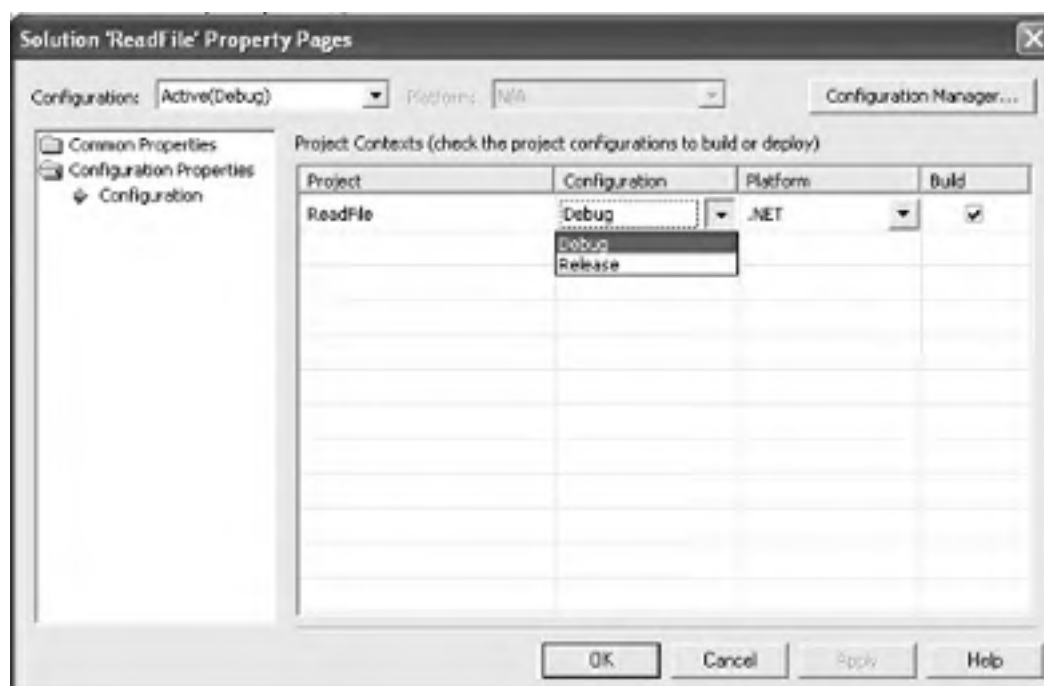


FIGURE 9.4 Setting the project's execution mode

Developing means testing and debugging. To aid debugging, Visual Studio provides a number of tools that we're going to explore in this section. The universal debugging tool is single-step execution: we execute one statement at a time and break to see the values of the variables, make sure that the statement produced the correct result, and continue by executing the next statement. You can also change the values of the variables to force an error condition and see how your code handles it. Before exploring the tools for debugging an application, let's discuss for a moment the type of errors that can occur in the process of coding an application.

Types of Programming Errors

Programming errors fall into three categories: syntax errors, runtime errors, and logic errors. *Runtime errors* are the errors that can't be prevented at design time; they must be handled with structured exception handlers, as we discussed in the first part of this chapter.

A *syntax error* is caught by the editor as you type and there's no excuse for syntax errors in

your code. As soon as you type a statement that contains a syntax error, the editor will underline the statement in error with a red wiggly line. If you rest the cursor on top of the statement in question, a description of the error will appear in a ToolTip box, as shown in Figure 9.5. The most common syntax error is calling a function with the wrong number of arguments, or arguments of the wrong type. To make the most of the editor's ability to catch syntax errors, you must turn on the Explicit and Strict options in every module.

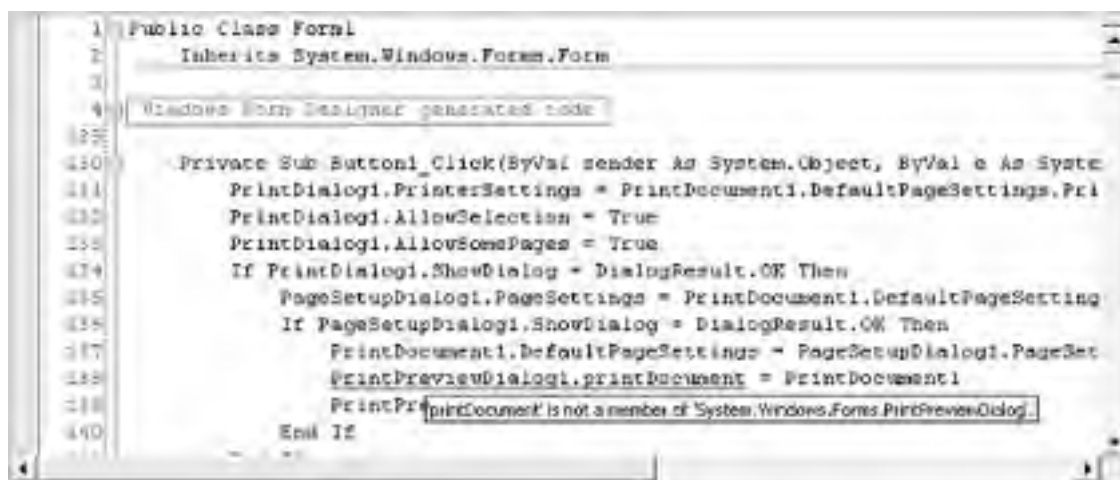


FIGURE 9.5 Syntax errors are caught by the editor as you type.

When the Explicit option is on, every variable must be declared. If you attempt to use a nondeclared variable, the editor will catch it as soon as you type it. When the Strict option is on, variables must be of specific types, and you can't rely on the compiler's ability to cast a variable's type according to the context in which it's used. If you got accustomed to using variables as you need them with earlier versions of Visual Basic, it's never too late to kick the habit. Let's see what the Strict and Explicit options can do for us. Turn off the Explicit and Strict options by inserting the following two statements at the top of the form's code window:

```
Option Explicit Off
Option Strict Off
```

Then insert the following statements in a button's Click event handler:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
    Handles Button1.Click
    Dim a As Integer
    Dim b As Integer
    b = 9999.999
    a = Convert.ToInt64(b ^ 3)
    MsgBox(a)
End Sub
```

OK, it's rather unlikely that you'll attempt to assign a double value to an integer variable in such a short procedure, but if this were a long procedure, the declaration of the variables might be at the beginning of the procedure's code, and when you enter the actual statements the declaration might not be in view. It's not the most uncommon type of error.

When the procedure is executed, an overflow will occur and the program will crash. You can use a structured exception handler in your code, which will prevent the program from crashing, but the result is probably incorrect. Do we really want to convert the result, which is a double value, to an integer?

Turn on the Explicit and Strict options. As soon as you do so, the last two statements will be underlined by the editor. The editor is warning you that you can't convert a double to an integer (for the first statement) and a long value to an integer. As soon as you realize that the two statements are in error, you will revise your code accordingly. If you start coding with the Strict option turned off and you turn it on after you've written a bit of code, the editor will catch many conversion errors.

INFINITY AND OTHER ODDITIES

VB.NET can represent two very special values, which may not be numeric values themselves but are produced by numeric calculations. They're the NaN (not a number) and Infinity. If your calculations produce a NaN or Infinity, you should confirm the data and repeat the calculations, or give up. For all practical purposes, neither NaN nor Infinity can be used in everyday business calculations. However, statements that produce these two values do not throw exceptions. If a NaN value is produced in your calculations, all following statements will produce NaN values as well. You may discover that something went wrong in your calculations, but not discover it for many statements—or even another procedure—after the line that produced the first NaN value. These two values must be handled carefully from within your code, so let's take a closer look at the mechanisms for handling NaN and Infinity values.

FROM VB6 TO VB.NET

VB.NET introduces the concepts of an undefined number (NaN) and infinity to Visual Basic. In the past, any calculations that produced an abnormal result (i.e., a number that couldn't be represented with the existing data types) generated runtime errors. VB.NET can handle abnormal situations much more gracefully. NaN and Infinity aren't the type of result you'd expect from meaningful numeric calculations, but at least they don't produce runtime errors.

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

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

Then divide this value by zero:

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

and display the variable's value:

```
MsgBox(infVar)
```

The string "Infinity" will appear on a message box. This string is just a description; it tells you that the result is not a valid number (it's a very large number that exceeds the range of numeric values that can be represented in the computer's memory). Another calculation that will yield a non-number is when you divide a very large number by a very small number. If the result exceeds the largest value that can be represented with the double data type, the result is Infinity. Declare three variables as follows:

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

[Team Fly](#)

 Previous

Next 

The notation 1E299 means 10 raised to the power of 299, which is an extremely large number. Likewise, 1E-299 means 10 raised to the power of -299, which is equivalent to dividing 10 by a number as large as 1E299. Then divide the large variable by the small variable and display the result:

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

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

NaN is not new. Packages such as Mathematica and Excel have been using it for years. The value NaN indicates that the result of an operation can't be defined: it's not a regular number, not zero, and it is not Infinity. NaN is more of a mathematical concept, rather than a value you can use in your calculations. The Log() function, for example, calculates the logarithm of positive values; the logarithm of a negative value does not exist. If the argument you pass to the Log() function is a negative value, the function will return the value NaN to indicate that the calculations produced an invalid result.

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

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

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

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

will all yield NaN. If you make `var2` a very small number, like 1E-299, the result will be zero. If you make `var1` a very small number, then the result will be Infinity.

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

Testing for Infinity and NaN

To find out whether the result of an operation is a NaN or Infinity, use the IsNaN and IsInfinity methods of the single and double data type. The integer data type doesn't support these methods,

[Team Fly](#)

 Previous

Next 

even though it's possible to generate Infinity and NaN results with integers. If the `IsInfinity` method returns `True`, you can further examine the sign of the Infinity value with the `IsNegativeInfinity` and `IsPositiveInfinity` methods.

In most situations, you'll display a warning and terminate the calculations. The statements of Listing 9.5 do just that. Place these statements in a Button's Click event handler and run the application.

LISTING 9.5: HANDLING NAN AND INFINITY VALUES

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

Listing 9.5 will generate a NaN value. Change the value of the `var1` variable to 1 to generate a positive infinity value, or to `-1` to generate a negative infinity value. As you can see, the `IsInfinity`, `IsPositiveInfinity`, `IsNegativeInfinity`, and `IsNaN` methods require that the variable be passed as argument, even though these methods apply to the same variable. An alternate, and easier to read, notation is the following:

```
System.Double.IsInfinity(result)
```

This statement is easier to understand, because it makes it clear that the `IsInfinity` method is a member of the `System.Double` class. If you change the values of the `var1` and `var2` variables to the following and execute the application, you'll get the message "Encountered a very large number":

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

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

Dealing with Logic Errors

Logic errors are the result of poor design or bad programming practices. A program that contains logic errors doesn't always crash; it usually produces incorrect results. Actually, your program will produce the correct results most of the time and may produce incorrect results every once in a while. To discover logic errors in your code, you must test it thoroughly. If a logic error goes undetected, it will eventually cause problems to the users (it will also harm your image as a developer).

We know our code contains logic errors if it doesn't produce the expected results. The universal debugging tool is to execute your code one statement at a time and examine the values of the variables, verify that program's control flow, and so on. In this mode of execution, the CLR executes one statement and stops. While no code is executed, you have at your disposal several windows, where you can interact with the CLR.

BREAKPOINTS

To discover an error in your application's logic, you can execute one statement at a time and then examine the values of the various variables. If everything looks correct, you continue by executing the next statement, and so on until you discover a statement that doesn't work as expected: it produces the wrong result, uses an un-initialized variable, etc. To interrupt the execution of your code, set a breakpoint at the statement where you want the program to halt its execution by clicking in the left border of the editor, in front of the desired statement. Breakpoints are indicated by brown dots; you can remove a breakpoint by clicking the dot again. Breakpoints can be set at executable statements. For example, you can't set a breakpoint at a Dim statement (unless it's followed by an initialization statement, which makes it an executable statement). If you attempt to set a breakpoint at an invalid location, the IDE will warn you and will reject the action.

Then you can start the application as usual, by pressing F5 or selecting Start from the Debug menu. The Start Without Debugging command, whose shortcut is Ctrl+F5, ignores the breakpoints. As soon as the breakpoint is reached, the program will halt and you'll be taken to the editor's window, where the statement with the breakpoint is highlighted in yellow. The statement hasn't been executed yet. You can hover the pointer over the names of the variables you're interested in and examine their values. To execute this statement press F10. The statement will be executed and the program will break again on the following statement. You can also press F5 to continue the execution of the application normally.

Breakpoint Properties

In a large application you may hit a breakpoint many times before you discover an abnormal condition. A problem may surface during the last iteration of a loop, for example. You may also notice that an error occurs when a certain variable becomes negative. To handle similar situations efficiently, you can set two properties, the Hit Count and Condition properties. The

Hit Count property determines how many times a breakpoint must be hit before execution breaks. If the breakpoint is in a loop that executes 1,000 times and you suspect that the code fails during the last iteration, you can set the Hit Count property of the breakpoint to 998. The program's execution will halt only the last time through the loop. The Condition property is an expression that determines whether the breakpoint is hit or skipped.

[Team Fly](#)

 Previous

Next 

All breakpoints are listed in the BreakPoints window, along with their properties. To view this window, open the Debug menu and select Windows ➤ BreakPoints. As you add breakpoints in your code, their definitions appear in the BreakPoints window. To set a condition for a breakpoint, select it first and click the Properties button on the BreakPoints window's toolbar. The Break Condition dialog box will appear; here you can set a condition, like the one shown in Figure 9.6. Then you can specify when the program will break: either when the condition becomes true, or when the condition changes values. The condition may contain multiple variables, such as:

```
BaseValue < 0 And (iCounter > 9 Or partialSum > 1000000)
```



FIGURE 9.6 Setting the properties of a breakpoint

USING THE DBEBUGGER'S WINDOWS

Once you've set the proper breakpoints in your code, you can start executing the program in Debug mode. The program will halt as soon as it hits the first breakpoint. The line at which the program stopped is highlighted. This is the next statement that will be executed when you continue the program's execution. At this point you can examine the values of the variables in the Autos window, which is shown in Figure 9.7. This figure shows the code window at the location of the breakpoint and the Autos window. To view this window, open the Debug menu and select Windows ➤ Autos.

The Autos window contains a list of all the variables in the current statement and allows you to change the value of any of them. You can use this feature to cause an exception, for example, and see how the exception handler works. You can press F10 to execute the current statement and stop at the next one, or press F5 to continue the program's execution normally. However, you can't edit the code and continue, as you're probably used to doing in VB6.

Another helpful window is the Locals window, which is similar to the Autos window but displays the local variables (the variables that are valid in the current scope). As statements are executed, the contents of the Locals and Autos windows change to reflect the current settings of the corresponding variables.

The Watch window is similar to the Autos window, only here you add variables at will. While at breakpoint, right-click on any of the procedure's variables and select Add Watch. The variable will be added to the current Watch window. There are several Watch windows. Add variables to each one of them as needed and then open the appropriate Watch window with the Debug ➤ Window ➤ Watch command.

Another related window is the Quick Watch window, which is the quickest method to view a variable's contents. It's no different than the Watch window, but it allows you to view a variable or two occasionally without changing the contents the Watch windows.



FIGURE 9.7 The Autos window displays the values of the variables used in the current statement.

To step through your code, use the buttons of the Debug toolbar. The Start button starts the execution of the application and the Stop Debugging button ends the debugging session. The Step Into button executes the current statement and selects the next statement to be executed. Keep clicking this button to execute one statement at a time and at the same time view the values of the variables in the Locals window. A common debugging technique is to calculate the values of certain variables manually and then verify that our code has generated the same values. If not, we know the statement that produces the error.

The Step Over button treats a section of code as a single statement. This section of code is usually a function or subroutine that has been debugged and is not part of the problem. Click the Step Into button to execute isolated statements, and when you reach the statement that calls the procedure you can click the Step Over button. The procedure's code will be executed at once and the debugger will stop at the following executable statement.

Sometimes we test a section of code, or a function, thoroughly and we assume that it will always work. The well-tested code may fail when it's incorporated into a larger program. The source of this behavior may be a form-level variable. To simplify debugging, try to minimize the scope of your variables. Variables should be invisible outside the section of code that uses them, so that they won't inadvertently affect unrelated parts of code. Function arguments should be passed by value, and rarely by reference. If a function changes the value of a reference variable, it may affect the code following the statement that called the function. Even a thoroughly tested and debugged function may cause problems to a larger module if it acts on reference variables.

The last button on the debugger's toolbar is the Step Out button, which executes the remaining statements in a function at once and breaks at the statement following the point where the function was called. All these buttons are enabled while you work in Debug mode.

THE OUTPUT AND COMMAND WINDOWS

The two most basic debugging tools are the Output and Command windows. Most of us insert statements in our code to print diagnostic messages, which appear on the Output window. In the

Command window, which is activated when a breakpoint is hit, you can execute statements and request, or set, the values of the variables in scope. For example, you can examine the values of the variables involved in the following statement and verify manually that the statement, as coded, produced the correct result. You can even copy a statement from the code window into the Command window and execute it. If the statement throws an exception, you'll see the description of the exception in the Command window, but the execution of the application won't be terminated. You can change the value of one or more variables and proceed. If you need to change the statement, however, you must interrupt the program's execution.

To send messages to the Output window, use the `Debug.WriteLine` method (or the `Console.WriteLine` method). Inserting `WriteLine` statements in our code is a crude form of debugging, but it works. The messages allow us to quickly isolate the section of the code where the problem lies and take it from there. Another quick and dirty debugging technique is to insert statements that display messages on message boxes. The advantage of using the `MessageBox` statement is that it suspends the execution of the application, without hiding the user interface. If you want to see how a statement affects the user interface, you can display a message box before and after the statement in question and examine the changes in the form. This technique works well with graphics applications. Note that every time a breakpoint is hit, the form becomes invisible and you can only work with the code window.

THE CALL STACK WINDOW

Another important debugging tool is the Call Stack window. This window shows the path of the application from procedure to procedure and you can see not only where the exception occurred, but also how the application got there. Many exceptions occur in procedures, and they don't occur every time. A procedure may work fine most of the time and fail only when it's called from a specific statement in another procedure. To debug this type of problem, you should be able to backtrace the path of execution and find out how the procedure was called. You may discover in the procedure's code that one of the arguments has the wrong value, for example. This bug can't be fixed from within the procedure's code; you need to find out the statement that called the procedure and examine the code leading to this statement. This, in turn, may entail backtracing the call to this procedure until you find out the statement that's responsible for setting a variable to a value that causes a problem down the road.

The information in the Call Stack window shown in Figure 9.8 tells you the following: The program stopped at line 148, which is in the `Proc2()` procedure. `Proc2()` was called from within the `Proc1()` procedure's code, by the statement in line 141. Finally, `Proc1()` was called from within the `Button1_Click` procedure, which is a button's Click event handler. Double-click any of the lines in the Call Stack window and the editor will select the corresponding statement in the code window and highlight the statement in green. As you backtrace the sequence of calls that lead to the breakpoint, you can examine the values of the arguments of the various calls. In many cases it's not a procedure's code that causes an exception, but the arguments passed to the procedure. Being able to monitor the values of the arguments passed to each procedure will assist you in locating the source of the problem.

To debug large applications, you must understand how structured exception handling works in nested procedures, as any application of a respectable size is made up of a large number of procedures. When a procedure is called from within a Try block, the structured exception handler is in effect. If the procedure has its own error handlers, they will be activated accordingly. When an unhandled exception occurs in a procedure, the error propagates up in the chain of procedures until an exception handler is found. If no exception handler is found, the application will crash.

This page intentionally left blank.

Theoretically, it's possible for an Internet-deployed application to run offline, but we never do that. Typical business applications connect to databases and they need to be online. Users may be able to start an application offline from the local cache, but they will run into problems as soon as they attempt to access a database or any other resource on the server.

Preparing for Internet-Based Deployment

To demonstrate Internet-based deployment, we've developed a simple application, whose main form is shown in Figure 10.1. When the form is loaded, the code behind it populates a DataSet with the rows of the Employees table of the Northwind database. The DataSet is bound to a DataGrid control, which allows the user to edit the table. To commit the changes to the database, the user can click the Save Changes button. The Export To XML button persists the DataSet to an XML file on the client machine, and the third button on the form exercises the members of a custom class.



FIGURE 10.1 The NoTouch-Deployment application manipulates a DataSet with employees.

The DataSet with the employee data was created at design time and is a typed DataSet. What happens to a SqlConnection that's set up at design time when the application is deployed to a client? Basically, you have two options: either make sure that the same database exists at the target machines, or store the connection string to a configuration file and read it from there every time the application starts. If the database is deployed to a different server, or the user's credentials must be modified, an administrator can distribute a new configuration file, or instruct users to edit it. Alternatively, you can store the configuration files to a server, so that administrators can edit them at a single location yet the configuration file will affect all the clients on the local network. Since Internet deployment is used almost exclusively to distribute applications to clients within corporations, you can expect that the clients will have access to the same database servers as the web server. You can also count on a system administrator to distribute and configure the application at the clients.

To make the most of this type of deployment, you should implement a middle tier component as a Web service and install it on the same server from which the application is downloaded to the client machines. The application can be deployed easily on a web server: the users see a Windows forms application and the data access takes place through components that reside on the web server. Changing a business component doesn't entail any changes to the presentation tier (the Windows forms downloaded to the client), and the application can be accessed by users on the same network, or through the Internet.

[Team Fly](#)

 Previous

Next 

One of the limitations of the current version of Internet-deployed applications is that they can't access a Web service that resides on a web server other than the one you used to deploy the application.

Let's return to our sample application. Listing 10.1 shows the code behind the application's buttons. When the form is loaded, the Employees DataSet is populated. Users can edit the data on the DataGrid control and the Export To XML button saves the DataSet's contents to an XML file. The Save Changes button submits the changes to the database by calling the DataAdapter's Update method. The code doesn't handle update errors gracefully, but our goal is to demonstrate how to deploy the application, not how to use the ADO.NET objects.

LISTING 10.1: THE CODE OF THE NOTOUCHDEPLOYMENT PROJECT

```
Private Sub Form1_Load(ByVal sender As System.Object, _
                        ByVal e As System.EventArgs) Handles MyBase.Lo
    Employees1.Clear()
    DAEmployees.Fill(Employees1, "'Employees")
End Sub

Private Sub btnExport_Click(ByVal sender As System.Object, _
                            ByVal e As System.EventArgs) _
                            Handles btnExport.Click
    Dim FS As FileStream
    Try
        FS = New FileStream("C:\EmployeeData.xml", _
                           FileMode.OpenOrCreate, FileAccess.Write)
    Catch
        MsgBox("Could not write to file EmployeeData.xml")
        Exit Sub
    End Try
    Try
        Employees1.WriteXml(FS)
    Catch ex As Exception
        MsgBox(ex.Message)
        Exit Sub
    Finally
        FS.Close()
    End Try
End Sub

Private Sub btnSave_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
                          Handles btnSave.Click
    Try
        DAEmployees.Update(Employees1)
    Catch ex As Exception
        MsgBox(ex.Message)
    End Try
End Sub
```

```
End Sub

Private Sub btnClass_Click(ByVal sender As System.Object, _
                           ByVal e As System.EventArgs) _
    Handles btnClass.Click
    Dim C As New ClassLibrary1.Class1
    MsgBox(''The GetQuote method returned " & C.GetQuote( "ABC" ).ToS
End Sub
```

The project contains a custom class, which will allow us to experiment with the deployment of DLL files. Add a new Class Library project to the solution, and place a single class to the project, the Class1 class:

```
Public Class Class1
    Public Function GetQuote(ByVal stockID As String) As Decimal
        Dim rnd As New System.Random
        Return Convert.ToDecimal(rnd.NextDouble + 5)
    End Function
End Class
```

The Test Class button on the application's main form calls the GetQuote method of the Class1 class and displays the return value on a message box. This value should be a decimal number in the range of 5 to 6. Class1 is the simplest class you can imagine and it exposes a single method. Note that it's implemented as a separate project of the same solution. If you add the class to the same project, it will be included in the same executable file as the main application. By implementing the class in a project of its own, a DLL will be added to the application, which must be deployed along with the main EXE file. As you will see, if we revise the class's code we can simply copy the DLL to the application's virtual folder and all clients will see the new version of the DLL.

It may not be clear at this point why we included all these operations. The sample application will be downloaded from a web server, and as such it will be executed in a context of seriously limited privileges. To persist the DataSet to an XML file, the application needs write-access to the client computer's file system. To update the local database, the application needs access to the database. None of these rights is granted by default to an application that's downloaded from a web server, and we'll see how to grant additional access rights to the application at the client.

Deploying a Windows Application on a Web Server

Let's deploy the application. First, we must create a virtual directory at the local web server to host the application. Open the Internet Information Services snap-in (Control Panel ➤ Administrative Tools) and locate the default web server. Add a new virtual directory by selecting New ➤ Virtual Directory from the Default Web Server item's context menu. Name the new virtual directory *NWEmployees* and set it to the NWEmployees folder under the web server's root folder (if this folder doesn't exist, create it). Then build the application, switch to the application's Bin folder, copy all the files and paste them into the new IIS virtual folder. That's all it takes.

[Team Fly](#)

 Previous

Next 

Here's an example showing how you can add new controls to a form while a program is running. Start by adding a Button to a form.

Let's assume that the user clicked a button asking to search for some information. You then create and display a TextBox for the user to enter the search criteria, and you also put a label above it describing the TextBox's purpose, as shown in Listing 2.1. Type this into the Button's Click event:

LISTING 2.1: GENERATING CONTROLS AT RUNTIME

```
Public Class Form1

    Inherits System.Windows.Forms.Form

    Dim WithEvents btnSearch As New Button()

    Private Sub Button1_Click(ByVal sender As Object, _
        ByVal e As System.EventArgs) Handles Button1.Click
        Dim textBox1 As New TextBox()
        Dim label1 As New Label()

        ' specify some properties:
        label1.Text = "'Enter your search term here..."
        label1.Location = New Point(50, 55) 'left/top
        label1.Size = New Size(125, 20) ' width/height
        label1.AutoSize = True
        label1.Name = "label1"

        textBox1.Text = ""
        textBox1.Location = New Point(50, 70)
        textBox1.Size = New Size(125, 20)
        textBox1.Name = "TextBox1"

        btnSearch.Text = "Start Searching"
        btnSearch.Location = New Point(50, 95)
        btnSearch.Size = New Size(125, 20)
        btnSearch.Name = "btnSearch"

        ' Add them to the form's controls collection.
        Controls.Add(textBox1)
        Controls.Add(label1)
        Controls.Add(btnSearch)

        'display all the current controls
```

To run the application, start Internet Explorer and enter the following URL in the Address box:

```
127.0.0.1/NWEmployees/NoTouchDeployment.exe
```

Notice that the URL contains the name of an EXE file, which is the application. This file will be downloaded and executed at the client. As soon as the application files are downloaded to the client and the application starts, the CLR will attempt to run it. When the first dialog box appears, you'll see an odd icon (a very large icon) on top of it, as shown in Figure 10.2. This icon warns the user that the application runs in a partially trusted context and some of its functionality may be disabled due to security restrictions. The icon is exceptionally large and you can't miss it.

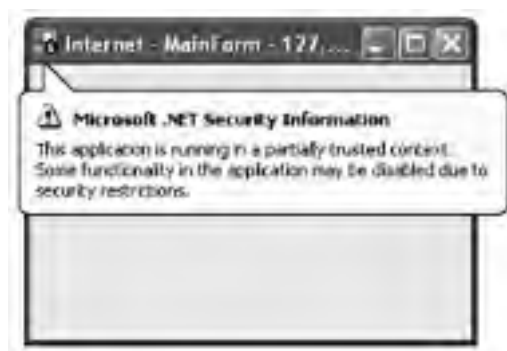


FIGURE 10.2 Applications deployed through a web server run by default in a context of reduced privileges.

The .NET Framework allows users and administrators to specify the privileges for each of the .NET applications they download from a web server. Without some special action by the user, applications downloaded from a web server can do very little. The CLR basically protects users from running a seemingly harmless application that may ruin their computer. If users know what they're doing, they will assign the proper rights to trusted sites and specific applications. .NET applications are deployed on intranets, and there are many mechanisms for identifying the trusted sites and assigning increased privileges to specific applications.

If you continue with the application's execution, when the code in the auxiliary form's Load event handler attempts to read the employees from the Northwind database, the message box shown in Figure 10.3 will appear. This message box tells you that the application attempted to use a member of the System.Data namespace, but the operation was not allowed by the client computer's security policy. Every time the application runs into an operation that can't be completed successfully, a similar message box appears.

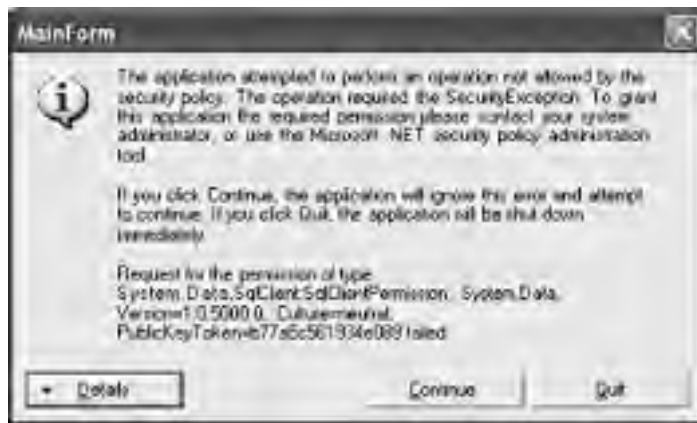


FIGURE 10.3 This message box appears when the application attempts to access a database on the client computer.

Code Access Permissions

The Common Language Runtime grants permissions to an application based on information it collects about the application, which is referred to as *evidence*. To collect evidence about an application, the CLR determines a number of factors, such as whether the site from which the application was downloaded is trusted or not, the zone of the corresponding site (was it the Internet, or the local intranet), the publisher's assembly, and more. Once the CLR has gathered all available evidence, it assigns permissions to the application. Of course, you can interfere with this process and specify custom security settings for an application.

The permissions granted to an assembly are based on the target computer's security policy. The security policy maps sets of permissions to applications, using the evidence of the application. To enable an Internet-deployed application to run on your machine, you create a set of permissions that are required for the application to run and then associate these permissions with certain types of evidence. For example, you may determine that only applications published by a specific manufacturer, or applications coming from certain URLs, have the right to create new files, or delete existing ones. In general, you can create many combinations of evidence and privileges, but the process is quite involved for an end user. That's why Internet deployment is ideal for corporations, where application settings can be controlled by an administrator, and not for the general public, even though it's quite possible to distribute applications from a web server to anyone with a reasonably fast Internet connection.

Security policy can be configured at the enterprise, machine, and user level. The enterprise level describes the security policy for the entire enterprise and is stored in an XML file, whose path is `config\enterprisesec.config`. The machine level describes the security policy for all applications on the local machine and is stored in the XML file `config\security.config`. Finally, the user level describes the security policy for the user running the application and is stored in a third XML file, at the path `Application Data\Microsoft\CLR Security Config\vXXX \security.config`, where `XXX` is the version of the Framework.

A permission set is a collection of named permissions; there are several built-in permission sets for the most typical situations. These built-in permission sets are shown in Table 10.1.

TABLE 10.1: BUILT-IN PERMISSION SETS

| NAME | DESCRIPTION |
|------------|---|
| Nothing | No permissions are given to the application in this permission set, not even the right to execute. |
| Execution | The application is allowed to execute, but has no access to the machine's resources. |
| FullTrust | The application is given unrestricted access to all resources on the machine. This set's permissions can't be modified, because you may disable crucial system applications, which run under this permission set. |
| Everything | The application is given unrestricted access to all resources of the machine. |

| | |
|------------------|---|
| Internet | The application is given the default privileges of an Internet application. |
| LocalIntranet | The application is given the default privileges of an intranet application. |
| SkipVerification | The application is not verified. |

[Team Fly](#)

 Previous

Next 

The permissions we assign to a permission set are called *code access permissions* (what type of operations do we permit the downloaded code to perform). There are several code access permissions you can grant (or deny) to an application, and most permissions have attributes. Table 10.2 shows the available permissions.

It is also possible to define custom permission sets. The .NET Framework Configuration tool provides a wizard to help you create a permission set. Alternatively, you can create an XML file that defines the permission set and add it to the policy using the *caspol* command line tool. This is how an administrator would apply a permission set to a large number of clients.

TABLE 10.2: CODE ACCESS PERMISSIONS

| PERMISSION | DESCRIPTION |
|-------------------------------|--|
| DirectoryServicesPermission | Controls access to Active Directory classes. |
| DnsPermission | Controls access to DNS servers on the network. |
| EnvironmentPermission | Controls access to individual environment variables. |
| EventLogPermission | Controls access to event log services. |
| FileDialogPermission | Allows read-only access to files that have been selected by the interactive user in an Open dialog box. |
| FileIOPermission | Controls access to individual files and directory trees. |
| IsolatedStorageFilePermission | Controls access to the isolated storage file system. Isolated storage provides a unique file system for an assembly. |
| IsolatedStoragePermission | Controls access to the isolated storage. |
| MessageQueuePermission | Controls access to Microsoft Message Queue (MSMQ). |
| OleDbPermission | Controls access to databases using OLE DB drivers. |
| PerformanceCounterPermission | Controls access to performance counters. |
| PrintingPermission | Controls access to printers. |
| ReflectionPermission | Allows access to view assembly metadata using Reflection. |
| RegistryPermission | Controls access to the Registry. |
| ServiceControllerPermission | Controls access to Windows services. |
| SecurityPermission | Controls the use of the security infrastructure itself. |
| SocketPermission | Allows making or accepting connections on a transport address. |
| SqlClientPermission | Controls access to SQL databases. |
| UIPermission | Controls access to user interface functionality such as clipboard, user input, and so on. |
| WebPermission | Controls access to Internet resources. |

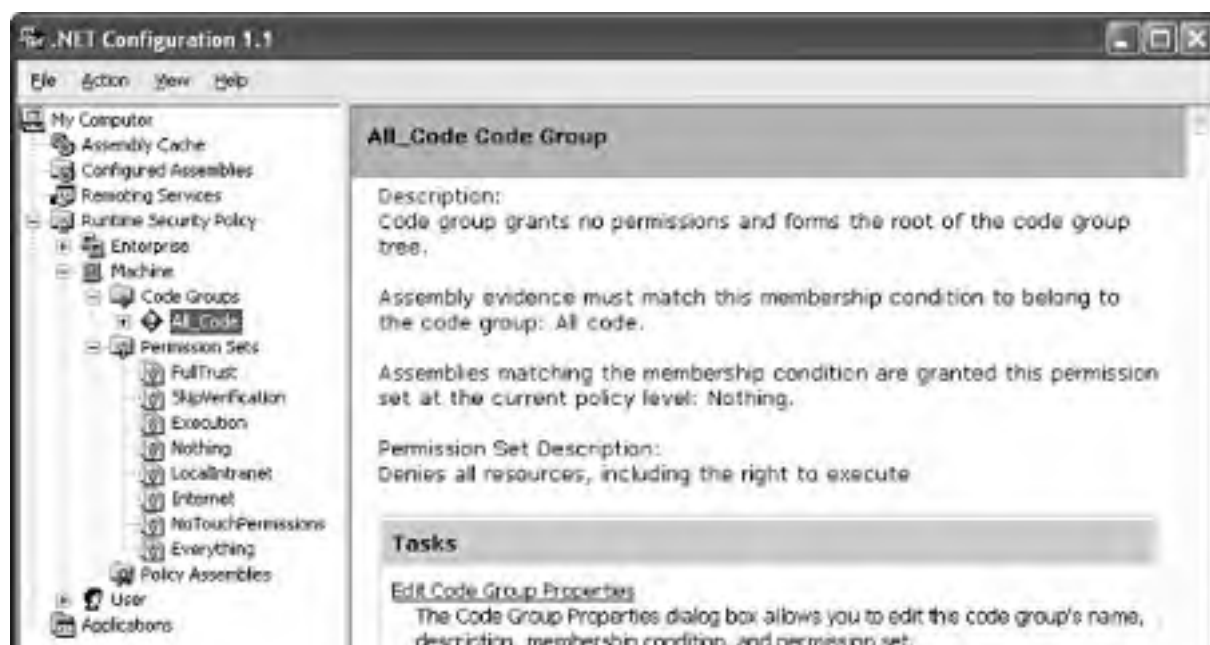
You may wonder why we should prevent an application downloaded from a web server from printing (the `PrintingPermission` code access permission). After all, the user determines when the application will print and what it will print, right? Not quite so. If you give an application unrestricted access to printers, the application may submit printouts to a remote printer, at a competitor's headquarters. The designers of .NET have taken into account all possible security breaches and implemented the appropriate security checks (until someone discovers how to harm your computer through a seemingly harmless operation, of course).

As a developer, you must create a list of permissions required by your application and distribute it to the clients. Even better, you can provide an XML file with the required permissions and distribute this file, which administrators can use to configure the client machines' code access permissions.

CREATING AND CONFIGURING PERMISSION SETS

To assign the proper code access permissions to an application loaded to the target machine from a web server, you must use the .NET Configuration snap-in. Open the Administrative Tools folder in the Control Panel and double-click the Microsoft .NET Framework *X* Configuration shortcut, where *X* is the version of the .NET Framework for which the application was written (since you can have multiple versions of the .NET Framework running in parallel). The window of the snap-in is shown in Figure 10.4.

Expand the Permission Sets item under the Machine branch of the snap-in. Here you see a list of predefined permission sets that you can assign to an application. The `FullTrust` permission set, for example, is used with applications installed by the administrator on the local machine. The `Internet` permission set is used with Internet applications; it grants very few privileges to applications. Let's create a permission set for the applications downloaded from a web server, the `NoTouch-Permissions` set.



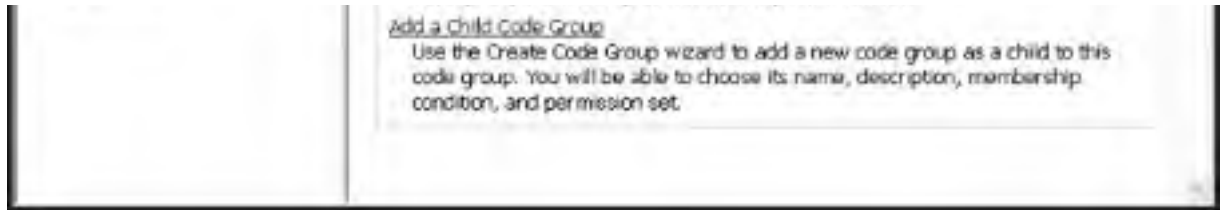


FIGURE 10.4 The .NET Configuration snap-in

Right-click the Permission Sets item and select New from the context menu. The Create Permission wizard will start, and its first window is the Identify The New Permission Set window, where you can enter the name and description of a Permission Set, or import a permission set from an XML file. As you know, permission sets can be persisted in XML files, which you can pass to system administrators to configure the client computers for your application.

Then click the Next button to see the Create Permission Set window, where you can assign permissions to the new set, as shown in Figure 10.5. The left list contains the available permissions, and you can select any of them and move them to the Assigned Permissions list, which contains the current set's permissions.



FIGURE 10.5 Assigning permissions to a permission set

Every time you select a permission from the left list and add it to the set's permissions, a dialog box comes up with specific attributes of the selected permission. Let's start by adding the SQL Client permission. Select it with the mouse and click the Add button to add it to the list of the set's permissions. The Permission Settings dialog box for the SQL Client permission will appear, as shown in Figure 10.6. You can grant an assembly the permission to access SQL Server through ADO.NET, or permission to access SQL Server without any restrictions. Even though it appears that the first option would be adequate for our sample application, you must check the radio button Grant Assemblies Unrestricted Access To Microsoft SQL Servers.

Our sample application also needs permission to access the client computer's file system. Select the File IO permission from the left list and add it to the set. The Permission Settings dialog box for the File IO permission is shown in Figure 10.7. You can grant the assemblies that will run under the specific permission set unrestricted access to the entire file system, or specify certain paths and the type of access for each path, as you can see in Figure 10.7. Let's give our permission set unrestricted access to the client computer's file system.



FIGURE 10.6 Setting the attributes of the SQL Client permission



FIGURE 10.7 Granting FileIO permissions

We've created a new permission set for an application that will be downloaded from a web server, so let's assign it to the sample application. Expand the Code Groups item and you'll see the All Code branch, which contains the various zones from which an application may come: My_Computer_Zone, LocalIntranet_Zone, Internet_Zone, Restricted_Zone, and Trusted_Zone. Add a new code group for the application that will be deployed from a specific

directory of the web server. Right-click the All_Code item and select New. You will see the Create Code Group wizard, which will take you through the steps of setting up a new code group. On the first window you can specify the name of the new code group and its description, or import a code group from an XML file.

Click the Next button to see the Choose A Condition Type window, where you'll specify a membership condition (see Figure 10.8). An assembly must meet the specified membership condition to be assigned the group's permissions; there are many types of conditions, as shown in Table 10.3.

[Team Fly](#)

 Previous

Next 

TABLE 10.3: MEMBERSHIP CONDITIONS

| CONDITION TYPE | DESCRIPTION |
|-----------------------|--|
| All Code | This membership condition is true for all assemblies (not the most practical setting). |
| Application Directory | This membership condition is true for all assemblies in the same directory (or a child directory) of the running application. |
| Hash | This membership condition is true for a specific assembly whose hash code matches the hash code specified on the dialog box. |
| Publisher | This membership condition is true for an assembly that has been digitally signed with a certificate that matches the one specified on the same dialog box. |
| Site | This membership condition is true for all assemblies that come from a specific site. The site is specified by its URL on the same dialog box. |
| Strong Name | This membership condition is true for all assemblies with a strong name that matches the one specified on the same dialog box. |
| URL | This membership condition is true for all assemblies that originate from the specified URL. |
| Zone | This membership condition is true for all assemblies that originate from the specified zone. |

Select the URL condition on the ComboBox control and set the URL to `http://127.0.0.1/NWEmployees/*`. This means that all applications in the NWEmployees virtual folder will belong to the same code group and they will be granted the permissions of the NoTouchDeployment set.





FIGURE 10.8 Setting a Membership Condition for a new code group

[Team Fly](#)

 Previous

Next 

Click the Next button again to see the last window of the wizard, where you're prompted to specify the code group's permission set (or create a new one). Click the radio button Use Existing Permission Set and select the NoTouchPermissions Permission Set. This is all it takes to create a new code group and assign to it a specific permission set. However, specifying the proper permissions is not a simple task and you can't count on end users to determine an application's permissions on their own.

DEMANDING PERMISSIONS THROUGH YOUR CODE

You can demand a specific permission from within your code before attempting to execute a statement that may fail due to lack of proper permissions. To demand a specific permission, create an object that represents the desired permission and call its Demand method. The statements that attempt to open a file for writing, for instance, could be written as:

```
Dim perm As New FileIOPermission(FileIOPermissionAccess.Write, filePath)
Try
    perm.Demand()
    FS = New FileStream(filePath, FileMode.OpenOrCreate, _
        FileAccess.Write)
Catch securityExcpTn As SecurityException
    Console.WriteLine(_
        "'Application was not granted access to file " & filePath)
    Exit Sub
End Try
```

Running the Application

The application's executables are downloaded to the client and are stored in the download cache. This is where the CLR looks for the files every time you start the application. However, it compares their versions to the versions of the files at the server and if it discovers a newer version of a file, it downloads it to the cache and then uses it. Start the NoTouchDeployment project from within your browser and click the Test Class button. You will see a value between 5 and 6 on a message box. Then switch to Visual Studio and change the statement that generates the random value to:

```
Return Convert.ToDecimal(rnd.NextDouble + 10)
```

Compile the class and copy the new DLL from the application's Bin folder to the application's virtual folder on the web server. Switch to the running application and click the Test Class button. You still get a value between 5 and 6, because the new DLL hasn't been downloaded to the client yet. To force the new version of the DLL to be downloaded to the client, close the browser and start it again. Connect to the application's URL and when the main form appears, click the Test Class button. This time you'll get a value in a different range, indicating that the DLL was downloaded automatically by the browser. You can also test the application from a workstation other than the one on which the web server is running. If you do, you will notice a small delay every time the application's files on the web server are updated.

Note that users need not start their browser to connect to an application deployed through a web server. They can create a shortcut to the URL of the application on their desktop and start the application by double-clicking this shortcut. When the shortcut is double-clicked, the browser's window comes up for a moment and then the application's starting form appears.

[Team Ely](#)

 Previous

Next 

You can also design a web page that the user can open to connect to your application on the web server, as shown in Figure 10.9. This is a very simple page, but you can add all the elements you can put on a web page to make it more attractive.

The HTML code of the page shown in Figure 10.9 is shown next. Just replace the URL of the hyperlink's target with the URL of the application on your company's web server:

```
<HTML>
<center>
<h1>No Touch Deployment Demo</h1>
</center>
<h3>Click
<A href=''http://127.0.0.1/NWEmployees/NoTouchDeployment.exe">
here
</A>
to start the NoTouchDeployment application.</h3>
</HTML>
```

Users can save the application's EXE on their hard drives. If they open the page with the hyperlink to the application with their browser, right-click the hyperlink, and select Save Target As, the application's EXE will be copied on the local drive, in a path selected by the user on an OpenFile dialog box. This action will download the main EXE file to the client, but not any of the other executables. These other executable files will be fetched from the web server when you run the application.



FIGURE 10.9 A simple web page to start the application

Downloading Assemblies on Demand

One last option with Internet-deployed applications is to download an assembly from within the application's code, as needed. The assembly could also be downloaded on a separate thread, while the user works with other parts of the application. To download an assembly

from within your code, use the LoadFrom method of the Assembly class, which accepts as argument the URL of the assembly on the web server and returns an Assembly object. Once the assembly has been downloaded to the client, you can extract any part of the assembly and use it in your code. For example, if the assembly contains a form you can extract the appropriate object, cast it to the Form type, and use it in your code as if it were part of the EXE on the local computer.


```
Dim i As Integer, n As Integer
Dim s As String
n = Controls.Count

For i = 0 To n - 1

    s = Controls(i).Name

    Debug.Write(s)

    Debug.WriteLine('')

Next

End Sub

Private Sub btnSearch_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnSearch.Click

    MsgBox("clicked")

End Sub

End Class
```

When adding new controls at design time, you want to at least specify their name, size, and position on the form—especially their Name property. Then use the Add method to include these new controls in the form's controls collection.

Here's how to go through the current set of controls on a form and change them all at once. The following example turns them all red:

```
n = Controls.Count
For i = 0 To n - 1
    Controls(i).BackColor = Color.Red
Next
```

Of course, a control without any events is often useless. To add events to runtime-created controls, you must add two separate pieces of code. First, up at the top (outside any procedure, because this declaration cannot be local), you must define the control as having events by using the WithEvents command, like this:

```
Dim WithEvents btnSearch As New Button()
```

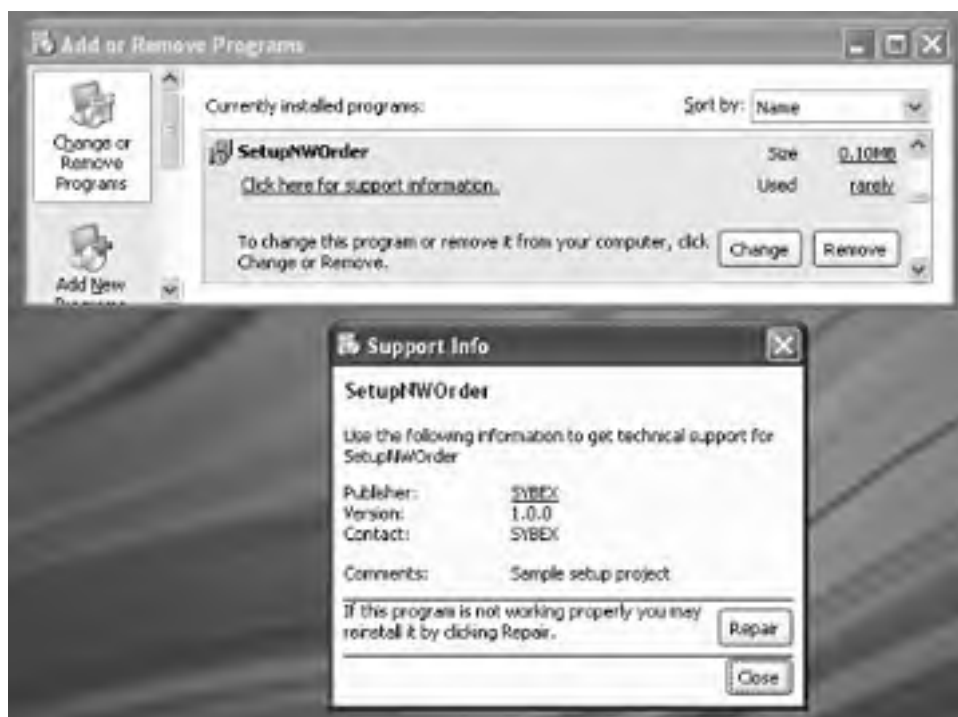


FIGURE 10.10 Windows installer packages create an entry in the Add Or Remove Programs snap-in.



FIGURE 10.11 The NWOrders application for entering new orders to the Northwind database

The NWOrders application uses ADO.NET to access the database, and it expects to find the database on the target computer. You can store the connection string to a configuration file so that administrators can change it. The code of the project is discussed in detail in Chapter 19. In this chapter we'll simply create a setup project for the application and use it to install the application on a client computer.

Let's start by outlining the application's architecture. The application has a single form, which

is shown in Figure 10.11. This form, and the code behind it, is the application's presentation tier: it interacts with the user and uses middle tier components to retrieve product prices, as well as to submit new orders to the database. The application's middle tier consists of two classes, the OrderClass and the BusinessLayer classes. The OrderClass class provides all the functionality needed to interact with the database, with the exception of the component that retrieves product prices. The BusinessLayer class is a "true" business layer, in that it calculates the price of a product when sold to a specific customer. The actual sale price is the product's retail price minus a discount that depends on the number of items of the same product sold to the same customer. This component is downloaded to the client, but it could have been implemented as a Web service and be installed on a remote machine. In our example, both classes belong to the application's namespace.

Copy the NWOrders project from the folder with the projects of Chapter 19 to a new folder and open it with Visual Studio. We're going to add a setup project to the solution. The setup project is no longer an external component. It's a Visual Studio project, and you can include it in the current solution. When you run the setup project in the IDE (with the commands Install and Uninstall of the setup project's context menu), the application is installed on your machine and you can easily test the steps of the installation, make sure that it creates the appropriate shortcuts, copies the auxiliary files (if any) to the application folder, and so on.

Creating a Windows Installer Package

To add a setup project to your solution, open the File menu and select Add Project ➤ New Project; the Add New Project dialog box will appear (see Figure 10.12). Click the Setup And Deployment Projects in the Project Types box and you will see the five setup and deployment projects available with Visual Studio, which are the following:

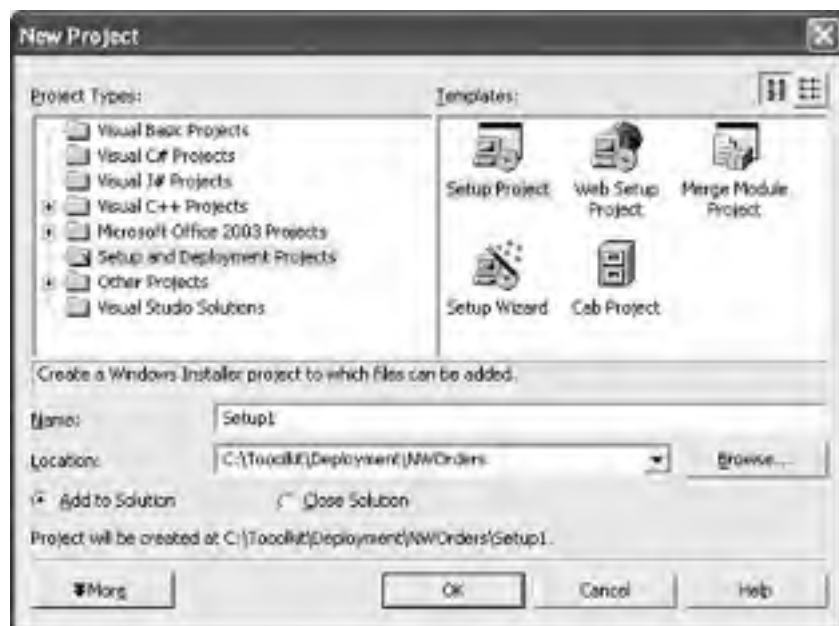


FIGURE 10.12 Selecting a setup project's type

Setup Project This project type creates a Windows installer package (an MSI package) and a bootstrap application that installs the application. Applications installed through setup projects are installed by default in the target computer's Program Files folder and can be later removed through the Add Or Remove Programs snap-in. This is the type of setup project we'll explore in this section.

Web Setup Project This project type creates a Windows installer package, similar to the Setup Project type, but it installs a Web application to the web server's virtual directory. Web applications are usually deployed with a simple copy operation. Besides, there's no real need to install Web applications on a large number of clients, so we won't discuss Web Setup projects.

Merge Module Project This project type creates merge modules, which contain the

files making up a specific component. Merge modules are usually included in other setup projects. The MDAC component, for example, is packaged as a merge module, so that it can be included in the setup project of any application that depends on MDAC.

Cab Project This project type creates CAB (cabinet) files. Cabinet files may contain any number of files; they're compressed to reduce the amount of data to be downloaded to the client.

Cabinet files are often used to package components, and users can choose to install individual components at the target machine.

Setup Wizard This is a wizard that takes you through the steps of creating setup projects of the other four types.

Select the Setup Project template and make sure the Add To Solution button is checked.

NOTE In Visual Studio 2003, this button was eliminated and the new project is always added to the current solution.

The IDE will add a few standard items to the project, as shown in Figure 10.13. On the designer's surface of the setup project you see a simplified view of the target computer's file system. Basically, a setup program modifies the target computer's file system by installing files and other items (shortcuts). While you design the setup program you specify how it will affect the target computer's file system, and this is what you do on the design surface with visual tools: you specify which files will be copied where, indicate the components to be registered, create shortcuts on the desktop, and so on.

To view all the items of the target computer that you can affect from within your setup project, right-click the setup project's name in the Solution Explorer and select View. This command leads to a submenu with the following items. Each item leads you to a different editor, which has the same name as the command.

NOTE The word editor is not part of the command, but it describes the window to which each command leads.

File System editor This is your view of the target computer's file system. You can add items under most folders of the target computer's file system.

Registry editor This is a simple tool that allows you to add keys to the target computer's Registry. These keys can then be read by the application's code.

File Types editor This is a simple tool that allows you to establish associations between your application and files with a specific extension.

User Interface editor This tool allows you to specify the interface of the setup program by adding custom dialog boxes to perform actions that are specific to your application's setup program. To customize the setup program's interface, you can add one or more of a number of predefined dialog boxes. However, you can't include your own dialog boxes; you must use one or more of the predefined ones and customize their captions. You can't change their appearance.

Custom Actions editor This tool is intended for advanced setup projects; it allows

you to perform custom actions, depending on whether any of the following events is fired: Install, Commit, Rollback, and Uninstall. The custom actions are usually performed by separate EXEs, which are invoked automatically when (and if) the corresponding event is fired. A server application's setup program, for example, may automatically start an executable to create a new database upon successful completion of the application's setup.

Launch Conditions editor This tool allows you to define the conditions that must be met at the target machine before the application is installed. For example, you can specify that the application is installed only if the target machine runs under Windows 2000.

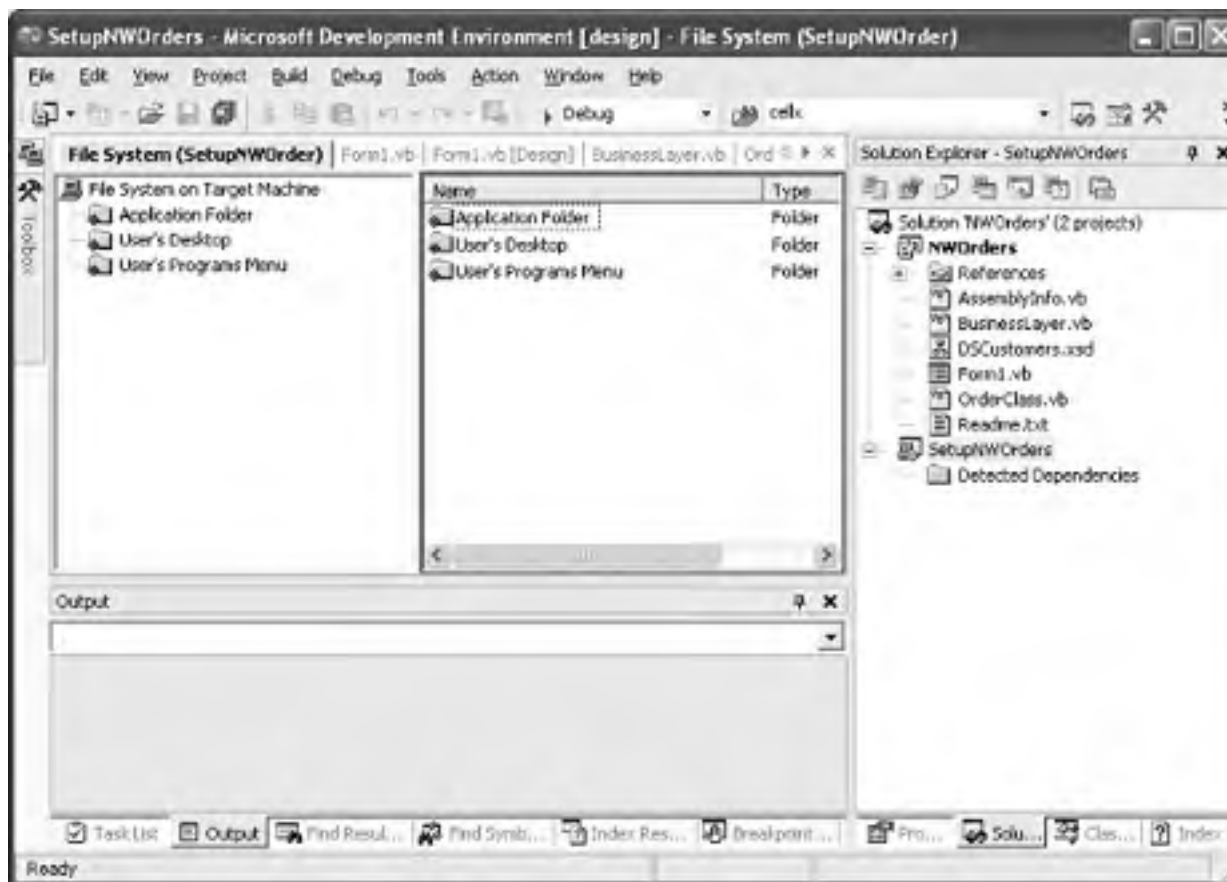


FIGURE 10.13 Working with a view of the target computer's file system

Using the File System Editor

The most important editor is the File System editor, which allows you to specify what will be installed on the target machine and where each item will be installed. At the very least, you must specify the project items that will be copied to the target computer. At the very least, we must copy the application's executable files (EXEs and DLLs). These files constitute the *project's output*. Rightclick the setup project and from the context menu select Add. You will see another menu with the following items, which represent the items you can add to the setup project:

Project Output Use this option to add to the setup program the project's executable files: the EXE and DLL files generated by the compiler. Every setup project should contain at least the output of another project.

File This option leads to a File Open dialog box, where you can locate and select any file to add to the setup program. The files you add to the setup project will be simply copied to the target computer. They are usually auxiliary (images, icons, XML configuration files, and so on).

Merge Module Use this option to select a Merge Module on the local machine and include it in the setup program.

Assembly Use this option to select one of the assemblies installed on the development machine and include it to the setup project.

Right-click the Application Folder item under the File System On Target Machine branch and from the shortcut menu select Add ➤ Project Output. A new dialog box will appear, shown in Figure 10.14, where you can select the project whose output you want to include in the setup program. The ComboBox control at the top of the Add Project Output Group dialog box contains the names

of all projects in the solution (except for the setup project, of course). Select the desired project on the ComboBox control. The outputs of the selected project are displayed on a ListBox control and you can select the desired output (which is usually the Primary Output). The Configuration ComboBox control in the middle of the dialog box lets you select the project's configuration: whether it's the Debug or Release configuration of the project. The default selection is (Active), which is the configuration of the project when you built it for the last time.

The item "Primary output from NWOrders (Active)" will be added under the setup project in the Solution Explorer's window. A dependency is also picked up, namely the Framework redistributable (`dotnetfxredistr_x86.msm`). Our sample project's custom classes are implemented in the same project, so the project's output is a single EXE file. If the classes were implemented as separate projects, the corresponding DLL would have been included in the project's output.



FIGURE 10.14 The Add Project Output Group

The project's dependencies will be detected automatically for you, including their own dependencies and so on. These files will be automatically added to the project's Detected Dependencies list. If you can be sure that certain of the dependencies already exist at the target computer, you can omit them by selecting the Exclude command from the appropriate item's context menu. If an application is deployed and a dependency is missing, the application may be useless. More often than not, developers don't exclude any of the dependencies. The MSI file may be larger than it should be, but the deployed application will always work.

The other types of output you can select from a given project are the following:

Primary Output The EXE and DLL files generated by the compiler for the selected project.

Localized Resources The DLL that contains resources specific to a locale.

Debug Symbols A file with debugging information, created when a project is compiled in Debug mode (a file with extension PDB). This file is used when the application is executed in the context of a debugger.

Common Files The content files (HTML, images, sounds, and so on), used only with ASP applications.

Source Files The project's source files (we usually don't distribute the application's source code).

As soon as you add the primary output of the NWOrders project to the setup project, the names of these files will be added under the Detected Dependencies item of the Solution Explorer. The IDE will pick any dependencies of these files and add them to the setup project's output automatically. If you don't want to include one of the selected dependencies (because you know that a component has already been installed), you can right-click its name in the Solution Explorer window and select Exclude from the shortcut menu. You can also select Refresh from the Detected Dependencies item's context menu to force the IDE to re-evaluate the dependencies.

You're ready to build the setup project and deploy your application to another computer. The setup project, however, will use mostly defaults, which isn't what users expect from a professional setup application. You can test the setup application right in the IDE by selecting the Install and Uninstall commands of the setup project's context menu. Actually, you should run the installation project at this point and print the various dialog boxes that appear during the installation project. Then you can use these printouts as you explore the properties of the setup project and understand better how to customize the appearance of the dialog boxes that appear during the setup.

If you build the entire solution, the process will take a while because the setup project must be built as well. However, you can build each of the applications separately. To see the compiler's output, switch to the setup project's Bin folder and you'll find there three items:

- Setup.exe* A bootstrap program that installs the application
- SetupNWOrders.msi* The MSI file (a Windows installer file) that contains all the data needed to install, fix, and remove the application from the target computer
- Setup.ini* The application's INI file

These three files must be distributed to every computer on which the application will be installed. Users can install the application by running the Setup.exe program.

The windows displayed during the installation process are very simple. The first window welcomes the user to the application's setup and waits for the user to click the Next button. The following window suggests the path of the folder where the application will be installed. Users can accept this location, or click the Browse button to select another folder. The Next button on this window takes you to the Confirm Installation window, where users must click Next to start the installation. The process takes a few moments; when the application's installation completes, users are informed of the completion of the installation and are prompted to close the wizard's last window. You will see later in this chapter how to customize the dialog boxes that appear during the application's setup.

If you start the Add Or Remove Programs utility in the Control Panel, you will see that your application has been installed on the target machine. Users can use this tool to repair or uninstall the application. Locate the application's entry in the Add Or Remove Programs tool's window and you'll see two buttons: Change and Remove. The Change button leads to a dialog box, which gives users the option to repair the installation by writing the original files on top of the existing ones, or to remove it. The Remove button removes the application from the target machine. You should always remove an application before installing a newer version.

THE SETUP PROJECT'S PROPERTIES

Each setup project has a few basic properties, which you can set in the Properties window if you select the name of the Setup project in the Solution Explorer's window. These properties identify the

manufacturer of the application and determine some basic characteristics of the setup application, such as whether it should detect and remove an earlier version of the same application. The most important properties of the setup project are the following:

Author Set this property to the name of the company (or individual) that developed the application; its default value is the same as the Manufacturer property's value.

Description Set this property to any helpful message about the application.

Manufacturer A string with the application's manufacturer. This setting is used by MSI to create the name of the default folder during installation, so you must set it to a value that users can easily associate with your application.

ManufacturerURL This is your company's URL, which is displayed as a hyperlink on the product's support page.

SupportPhone This is the phone of a telephone support department for the application.

SupportUrl This is the URL of the application's support website.

Version This is the version number of the Windows installer package.

These settings are used to compose the Support Info window, which is shown in Figure 10.10. To see this window, start the Add Or Remove Programs tool from the Control Panel, select the SetupNWOrder entry, and click the Click Here For Support Information hyperlink, also shown in Figure 10.10.

There are two more interesting properties of the setup project, which determine to some extent the behavior of the setup program, and they are the following:

DetectNewerInstalledVersion This property is True by default; it tells the Windows installer to detect any newer version of the same application that has already been installed and, if it finds one, prevent the installation of the older version.

RemovePreviousVersion Set this property to True if you want the setup program to remove an older version of the application (if one exists) and then install the newer version.

Creating Shortcuts

A simple task common among nearly all setup programs is the creation of a shortcut to the application on the user's desktop. To create a shortcut from within your setup application, click the Application Folder item in the File System editor and in the pane with this folder's contents, right-click the target file (Primary Output From NWOrders). One of the commands of the context menu is the Create Shortcut To Primary Output From NWOrders. Select this command and a new shortcut will be created. Change the shortcut's name to "Shortcut to NWOrders" and then drag the shortcut and drop it on the User's Desktop item on the File System pane. The installation program will create a shortcut to the application's EXE file and place it on the user's desktop.

If you look at the properties of the newly created shortcut, you'll see two particularly interesting ones. The Icon property lets you set the icon that will appear on the user's desktop and the Working-Folder property is the application's working folder. These are two basic properties you set every time you create a shortcut on your desktop.

[Team Ely](#)

 Previous

Next 

The User's Programs Menu item represents the user's programs menu; you can add a new menu item for your application. You can place your application directly on the user's menu, or create a subfolder with a descriptive name that will contain the application. Right-click the User's Programs Menu item in the File System editor and from the shortcut menu select Add Folder. A new folder will be created under the selected item; rename it to VBToolkit. Right-click the new item and from the context menu select Add ➤ Project Output. When the application is installed, a new item will be added to the user's programs menu, and the application's EXE will be added under this item. If you're installing multiple applications with the same setup program, place their EXEs in the same item of the user's programs menu.

You can also handle the special folders on the user's file system. Right-click the item File System On Target Machine and you will see a list of special folders, from which you can select one. These special folders are listed next:

| | | |
|-------------------------|--------------------------------|------------------------------|
| Common Files Folder | Fonts Folder | Program Files Folder |
| System Folder | User's Application Data Folder | User's Desktop |
| User's Favorites Folder | User's Personal Data Folder | User's Programs Menu |
| User's Send To Menu | User's Start Menu | User's Startup Folder |
| User's Template Folder | Windows Folder | Global Assembly Cache Folder |
| | Custom Folder | |

You can add items to any of these folders. To install a font at the target computer, for example, place the font into the target computer's Fonts Folder. If you want your application to run every time the computer is started, place the EXE file in the user's Start Menu.

The Registry Editor

Use the Registry Editor to add keys and values to the target computer's Registry. When you open the Registry Editor on the designer surface, you will see the organization of the Registry, but you can only access the HKEY_CURRENT_USER and HKEY_LOCAL_MACHINE branches of the Registry. The key HKEY_LOCAL_MACHINE\Software\[Manufacturer] key is created by default, but it hasn't been assigned a value yet. The [Manufacturer] key will be replaced by the value of the Manufacturer property of the setup project when the setup application is executed at the target machine.

Adding new keys is straightforward. Right-click the desired branch in the left pane of the Registry pane and from the context menu select New Key. The new key will be named New Key #1; you can change this name to anything. To set the key's value, right-click the key's name and from the shortcut menu select a value type: String value, Environment String value, Binary value, or DWORD value.

Using the User Interface Editor

A setup program has a simple user interface, which consists of a number of dialog boxes that guide the user through the installation process. It's possible to add custom dialog boxes to the interface, and this is done through the User Interface editor. Unfortunately, you can't create your own forms and include them in the project's UI.

[Team Ely](#)

 Previous

Next 

To view the User Interface editor, switch to the Solution Explorer's window and select View ➤ User Interface Editor from the setup project's context menu. On the designer's surface, you'll see a tree view of the dialog boxes that will appear on the user's desktop during the installation, in the order in which they will appear. Each dialog box usually has a few properties that you can use to customize the dialog box's appearance.

To add a new dialog box, right-click the designer and select Add Dialog. The dialog box of Figure 10.15 will appear, where you see the available dialog boxes. If you double-click one of these dialog boxes, a new item will be added to the tree with the installation application's user interface.

To customize the setup program's interface, set the properties of the dialog boxes. Select each dialog box on the left pane and look up its properties. Most dialog boxes have very few properties, such as the bitmap that will be displayed on them and a copyright warning.

To add a custom dialog box to the user interface, select one of the steps in the installation process (Start, Progress, End) and from its context menu select Add Dialog. The Add Dialog box that will appear contains a number of predefined dialog boxes. Most of them are made up of a small number of radio buttons, check boxes, and text boxes, as shown in Figure 10.16. Unfortunately, you can't edit these dialog boxes; you can't even view them at design time. You can only set their properties and view them while the application is actually being installed. To see what each custom dialog box looks like before using it in your project, look it up in the documentation.

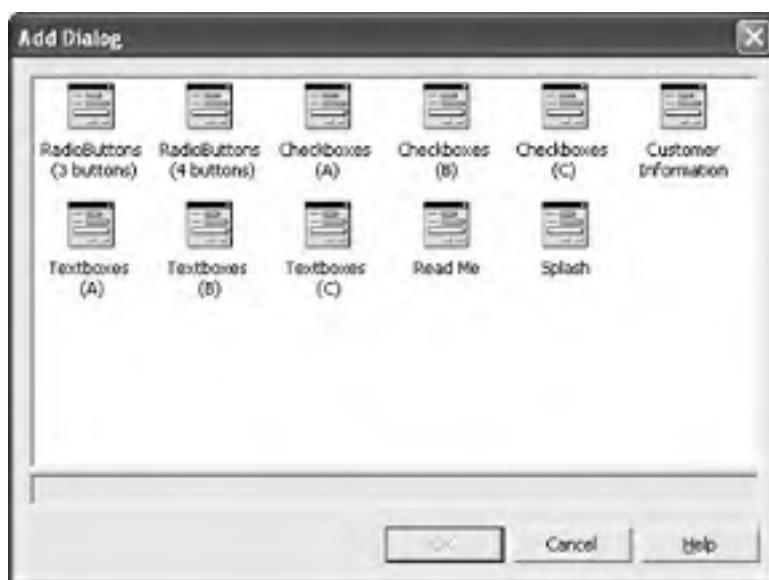


FIGURE 10.15 Use one or more of these dialog boxes to customize the setup project's UI.

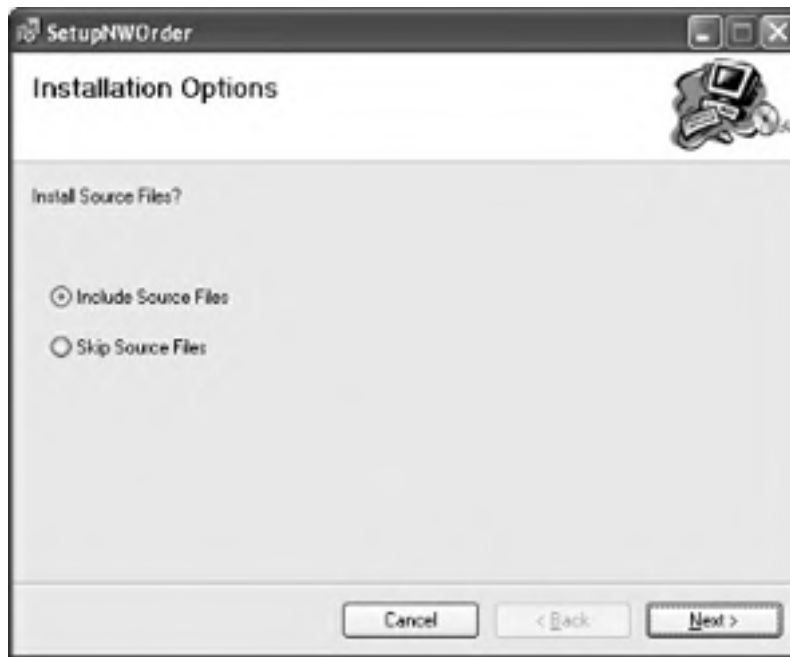


FIGURE 10.16 Getting user data during installation through a custom dialog box

Some of the custom dialog boxes are quite simple. The License Agreement dialog box displays the application's license agreement (which is a text file) and the usual "I Agree" and "I Don't Agree" radio buttons. If the user doesn't check the Agree radio button, the installation process is aborted.

Let's add a custom dialog box to our sample setup application. We'll add a dialog box with two radio buttons that will prompt the user to specify whether the project's source files should be installed or not. Select the dialog box with the two radio buttons and then set the new dialog box's properties as follows:

| Property | Setting |
|-----------------------|-----------------------|
| <i>BannerText</i> | Installation Options |
| <i>BodyText</i> | Install Source Files? |
| <i>Button1Label</i> | Include Source Files |
| <i>Button1Value</i> | 1 |
| <i>Button2Label</i> | Skip Source Files |
| <i>Button2Value</i> | 2 |
| <i>ButtonProperty</i> | SOURCEFILES |
| <i>DefaultValue</i> | 1 |

The names of the properties are self-explanatory. The ButtonProperty property is a basically a variable name, which is set to the value of the checked radio button. This value is used by another part of the setup program to control an action. In our case we'll use the input provided on the custom dialog box to determine whether the source files will be installed or not. Figure 10.16 shows what the custom dialog box looks like during the application's installation.

Switch to the File System editor and add the application's source files to the setup project's output. From the Application Folder item's context menu select Add ➤ Project Output, and when the Add Project Output Group dialog box appears, select the Source Files item. This action will cause the source files to be included in the package and a new item will be added to the Application Folder of the target machine.

To install the source files conditionally, based on the user's selection on the custom dialog box, open the Property Browser of the item that represents the source files and locate the Condition property. This property determines the condition that must be met for the selected component to be installed. Set the Condition property of the Source Files From NWOrder item to:

```
SOURCEFILES = 1
```

The application's source files will be copied by the installation program to the target computer's application folder if the user checks the Include Source Files radio button on the custom dialog box.

If the custom dialog box contains check boxes, you must take into consideration that users may check multiple check boxes. Each check box has a `CheckBoxValue` property, whose value is either `Checked` or `Unchecked`. The `Condition` property in this case should be set to a logical expression such as:

```
CheckBox1Value = Checked And CheckBox2Value = Unchecked
```

[Team Fly](#)

 Previous

Next 

Then, in the location where you want to provide the code that responds to an event, type the following:

```
Private Sub btnSearch_Click(ByVal sender As System.Object, _  
ByVal e As System.EventArgs) Handles btnSearch.Click  
    MsgBox(''clicked")  
End Sub
```

This event code is indistinguishable from any other "normal" event in the code. Notice that unlike the all-in-one event shared by all the controls in the entire control array in VB6, VB.NET expects you to give each newly created control its own name and then use that name to define an event that uses "Handles" Name.Event, as illustrated by the Click event for btnSearch in the previous example.

Multiple Handles

If you're wildly in favor of the all-in-one event, you can do some curious things in VB.NET that permit it. Listing 2.2 is an example that handles the Click events for three different controls by declaring their names following the Handles command.

LISTING 2.2: HANDLING MULTIPLE CONTROLS WITH A SINGLE EVENT

```
Private Sub cmd_Click(ByVal sender As System.Object, ByVal e As  
System.EventArgs) Handles cmdOK.Click, cmdApply.Click, cmdCancel.Clic  
    Select Case sender.Name  
        Case "cmdOK"  
            'Insert Code Here to deal with their clicking the OK  
        Case "cmdApply"  
            'Insert Code Here for Apply button clicking  
        Case "cmdCancel"  
            'Insert Code Here for Cancel button clicks  
    End Select  
End Sub
```

In addition to the multiple objects listed after Handles, notice another interesting thing in this code. There's finally a use for the sender parameter that appears in every event within VB.NET. It is used in combination with the .NAME property to identify which button was, in fact, clicked. This event handles three Click events. Sender tells you which control raised (triggered) this event at any given time. In most VB.NET code, the Sender object is the same as both the name of the event and the name following the Handles:

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

You'll find a variety of interesting and helpful new tools and techniques available to you in ASP.NET. If you're familiar with ASP, or its best-forgotten predecessors, you're likely to be delighted with what ASP.NET can do for you, and what it allows you to do in your code. For example, here's a simple bit of code that tests whether a client's browser permits VBScript execution, among other things, by using the `Browser` property of the `HttpRequest` class, like this:

```
Private Sub Page_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    If Request.Browser.JavaScript Then
        Response.Write('Accepts JavaScript")
    End If
    If Request.Browser.VBScript Then
        Response.Write("<BR>Accepts VBScript")
    End If
    Dim s As String = Request.Browser.Browser
    Response.Write("<BR>Browser:" & s)
    s = Request.Browser.Version
    Response.Write("Version" & s)
End Sub
```

Sending Entire Files

In ASP.NET, you can provide entire files as arguments to the `Response` object. If you have a text file you want to slap onto the user's browser, simply provide its filepath on your server, like this:

```
Private Sub Page_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    Dim s As String = "C:\test.txt"
    Response.WriteFile(s)
End Sub
```

Note also that when a web page is requested, its ASP.NET file (.aspx) is compiled into a temporary .dll that is used for subsequent requests. Clearly this approach improves scalability, security, and overall performance. Best of all, you, the programmer, don't have to do anything about it. It all happens behind the scenes, like so many aspects of ASP.NET and ADO.NET—that's why I said that these technologies are high-level. You forget about lots of messy details, and write your instructions using abstract, powerful commands.

Using Server Controls

Also new in .NET are server controls, or as they're sometimes called, *server-side* controls. As their name suggests, these controls reside on the server, but they "compose" a plain-vanilla HTML page that is sent to the client. The code-behind page is also composed into HTML, if necessary, which is why you sometimes see HTML formatting instructions mixed in with traditional VB.NET, such as "
".

```
        Response.Write(' ' <BR> ")
    End While

    datReader.Close()
    conString.Close()
End Sub
```

Try running this example by pressing F5. You should see the results shown in Figure 11.1.



FIGURE 11.1 A database's contents directly displayed in the client browser

If you get an error message, such as "This page cannot be displayed," "Permission denied," or "Login failed," the attempted connection to the sample database "Pubs" probably failed to connect. Perhaps you never installed the sample database, or you don't have a version of SQLServer running, or your level of permissions hasn't been properly set to access the data or to use the SQLServer. To solve these and other possible problems, look at the various suggestions in Chapter 13.

The code in this example is essentially the same as you would use in an ordinary Windows application to connect to a database and get data via a query. The only really unusual aspect is the "
," the HTML line break command, and the `Response.Write` to print strings on the client's browser.

Now let's look at several excellent ASP.NET controls that you can use to quickly and effectively create database-driven web pages.

[Team Fly](#)

 Previous

Next 

The DataList, Repeater, and Templates

The DataGrid and other WebForm controls are quite an improvement over traditional HTML tables and other HTML GUI effects—an improvement for both the user and the programmer. The programmer has far less work with server-side WebForm controls, and the results are superior.

The DataList is a somewhat less advanced form of DataGrid. The DataList is also similar to the Repeater control discussed later, but the DataList offers a default format and some tools you can use to specify how the data will be displayed. However, as with the Repeater, there's no support for paging.

The DataList is somewhat less powerful than the DataGrid. You can manipulate several of the DataList's properties by adjusting templates for such elements as header, footer, AlternatingItem, EditItem, Item, SelectedItem, and Separator. The only required template is the Item. In addition, you can specify how the display is built visually by setting repeatcolumns, repeatdirection (vertical, horizontal), and repeatlayout (flow, table).

The DataGrid includes all eight templates; the DataList includes all except the PagerTemplate; the Repeater doesn't have the Pager, SelectedItem, or EditItem templates.

TEMPLATES

Templates are used with DataGrid, DataList, and Repeater controls, but only the Repeater requires that you use a template. A template is a collection of HTML elements (including as well any controls you place into those elements) that collectively define, at least in part, the appearance and behavior of a control that contains the template. One way to use a template is to allow it to specify the appearance of every row in a DataGrid (templates can repeat within their container control).

It's easy to confuse templates with styles because both perform essentially the same jobs and do it in similar ways. However, styles generally describe such properties as FontSize and colors, and can override the defaults for these types of properties in a given control. Templates, by contrast, define such features as headers, alternating items in a table or list, separators, and so on.

You can create and modify templates using XML declarations in .aspx files via the `<template>` element. Or you can use the Edit Template feature, described in the following section.

THE DATALIST

Put a DataList control on a WebForm. The control's icon itself has a description telling you to right-click the icon. Choose the Edit Template option. You can manipulate templates to specify what additional controls are included in the DataList, as well as adjusting the appearance of the dividers, header, footer, and individual cells. You can also add any ASP.NET control you wish to a template.

Select the ItemTemplate area in the DataList (you may have to left-click this area in the icon repeatedly until you see the insertion cursor blinking). Double-click a TextBox in the Toolbox. Now you've done it; the client will now view a TextBox in each item displayed in the DataList. Now select the AlternatingItem Template area and add a TextBox there as well, but change this TextBox's Back-Color property to light blue.

Go ahead and put a TextBox in the currently selected item, changing its BorderStyle to inset and the BorderColor to red, and type the words **Edited Here** in the Edited Item area. You can if you wish add other controls (except timers and such) to the template. Now right-click the DataList icon again and choose End Template Editing.

THE REPEATER

The Repeater control is simpler than the DataGrid or DataList. For one thing, it's read-only. It also doesn't have any built-in visual structure and it also requires a template. It's possible to use it via template adjustments for various list formats: numbered, bullets, tabs, and so on. When you add a Repeater control to a WebForm, you're told to go to HTML view and do the editing there. There is no default formatting, no RAD or visual tools to assist you in creating the format. It's all totally up to you; you have to type in any templates by hand in the HTML editor.

Drop a Repeater control onto an empty WebForm, then switch to the HTML code window. Move the element's end tag (`</asp:Repeater>`) down several lines so you have some room to add a couple of templates.

Now type in the following boldface templates:

```
<li><%# Container.DataItem('au_lname') %>  
<%# Container.DataItem("au_fname") %>  
</li>
```

That's the format you use to bind each item template to the container (WebForm page) data source. Each field must be specified by name for each item, as illustrated in the code above. And you have to use the code-behind window to bind the repeater to a data store:

```
Dim ds As DataSet = New DataSet  
    Dim pubConn As SqlConnection = _  
New SqlConnection("DataSource=localhost;Integrated Security=SSPI;Initial Catalog=  
    Dim selectCMD As SqlCommand = _  
New SqlCommand("SELECT * FROM Authors", pubConn)  
    Dim da As SqlDataAdapter = New SqlDataAdapter  
    da.SelectCommand = selectCMD  
    pubConn.Open()  
    da.Fill(ds, "Authors")  
    pubConn.Close()  
    Repeater1.DataSource = ds  
    Repeater1.DataBind()
```

Using the DataGrid

The DataGrid is the most powerful of the three data-related WebForm controls. It has all the features available to the DataList control (discussed later), but also supports paging and sorting. All three data-related controls—DataGrid, DataList, and Repeater—are based on templates that you can customize. Templates, too, are discussed later.

You can dump a table or other data into the DataGrid and format it as you wish. Here's a simple example that displays the same authors table used in the previous example. By leaving the DataGrid's

AutoGenerateColumns property to the default True, the data table you specify as the DataSource simply flows into your grid all by itself, complete with headers.

From the WebForms tab on the Toolbox, drag a DataGrid to the WebForm. Then, using the same Imports statements as in the previous example, type Listing 11.2 into the Form_Load event.

LISTING 11.2: FLOWING DATA INTO A DATAGRID

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

    Dim conString As String = _
    ''Data Source=localhost;Integrated Security=SSPI;Initial Catalog=pubs
    Dim con As New SqlConnection(conString)
    Dim cmd As New SqlCommand("select * from Authors", con)

    con.Open()

    Dim dReader As SqlDataReader = _
    cmd.ExecuteReader(CommandBehavior.CloseConnection)

    DataGrid1.DataSource = dReader
    DataGrid1.DataBind()
    dReader.Close()
    con.Close()

End Sub
```

The DataSet object is useful when you're permitting the client to update your database. If you merely want a read-only set of data, use the faster DataReader as in the previous example.

There are usually several ways to accomplish a given task in .NET, so here's an alternative way (Listing 11.3) to fill the DataGrid by creating a DataSet, filling it with the table, then binding the DataGrid to the DataSet.

TIP You could also bind to one of the tables in the DataSet collection: Tables(0).

LISTING 11.3: BINDING A DATASET TO A DATAGRID

```
Dim ds As DataSet = New DataSet

    Dim pubConn As SqlConnection = _
    New SqlConnection( "Data Source=localhost;Integrated Security=SSPI;Ir

    Dim selectCMD As SqlCommand = _
    New SqlCommand( "SELECT * FROM Authors", pubConn)

    Dim da As SqlDataAdapter = New SqlDataAdapter
```

```
da.SelectCommand = selectCMD  
pubConn.Open()  
da.Fill(ds, "Authors" )  
pubConn.Close()  
  
DataGrid1.DataSource = ds  
DataGrid1.DataBind()
```

SPECIFYING BEHAVIORS

The DataGrid permits you to define five different types of columns for different jobs. You can also permit paging, define borders, and specify sorting and other facets of the grid's behavior. Click the ellipsis next to the Columns collection in the properties window (or right-click the DataGrid itself and choose Property Builder) and you'll see a dialog box pop out where you can specify these behaviors.

MAKING PRETTY

The previous examples can use some fixing up. They work fine, but to make them appear nice to the user, add some color and reduce the point size of the font. Click the HTML tab on the design window and add the boldface lines below to the description of the DataGrid:

```
<form id=" Form1" method="post" runat="server" >  
<asp:DataGrid id="DataGrid1" style="Z-INDEX:  
101; LEFT: 208px; POSITION: absolute; TOP: 88px"  
runat="server" Width="680px" Height="400px"  
BackColor="#ddffff"  
BorderColor="darkblue"  
Font-Name="Arial"  
Font-Size="8pt"  
CellPadding="2"  
CellSpacing="1"  
HeaderStyle-BackColor="a0b0c0">  
</asp:DataGrid></form>
```

***TIP** You may have problems when pasting code into the HTML window; you may get all kinds of strange additional text. If you see this happening, just delete the problem code from the HTML window, then recopy it and paste it into Notepad. That will strip off any extraneous codes and give you raw text when you copy and paste it from Notepad to the HTML window.*

By making the adjustments, you change the results from the crowded, hard-to-read version in Figure 11.2.

After you add the boldface formatting adjustments in the previous example, the design shown in Figure 11.3 is more professional and makes customers more likely to trust your company and its products.



The screenshot shows a Microsoft Internet Explorer window displaying a table of customer data. The table has 8 columns: au_id, au_name, au_fname, phone, address, city, state zip, and contract. The data is presented in a plain text format with no borders or styling, making it difficult to read.

| au_id | au_name | au_fname | phone | address | city | state zip | contract |
|-------------|------------|-------------|--------------|----------------------|---------------|-----------|----------|
| 172-32-1176 | White | Johnson | 408 496-7223 | 10932 Bigge Rd. | Menlo Park | CA 94025 | True |
| 213-46-8915 | Green | Marjorie | 415 986-7020 | 309 63rd St. #411 | Oakland | CA 94618 | True |
| 238-95-7766 | Carson | Cheryl | 415 548-7723 | 589 Darwin Ln. | Berkeley | CA 94705 | True |
| 267-41-2394 | O'Leary | Michael | 408 286-2428 | 22 Cleveland Av. #14 | San Jose | CA 95128 | True |
| 274-80-9391 | Straight | Dean | 415 834-2919 | 5420 College Av. | Oakland | CA 94609 | True |
| 341-22-1782 | Smith | Meander | 913 843-0462 | 10 Mississippi Dr. | Lawrence | KS 66044 | False |
| 409-56-7008 | Bennet | Abraham | 415 658-9932 | 6223 Bateman St. | Berkeley | CA 94705 | True |
| 427-17-2319 | Dull | Ann | 415 836-7128 | 3410 Blonde St. | Palo Alto | CA 94301 | True |
| 472-27-2349 | Gringlesby | Burt | 707 938-6445 | PO Box 792 | Covelo | CA 95428 | True |
| 486-29-1786 | Locksley | Charlene | 415 585-4620 | 18 Broadway Av. | San Francisco | CA 94130 | True |
| 527-72-3246 | Greene | Morningstar | 615 297-2723 | 22 Graybar House Rd. | Nashville | TN 37215 | False |
| 648-92- | Blotchet- | | 503 745- | | | | |

FIGURE 11.2 BEFORE: Without formatting attributes, the browser does its best to fit everything in, but the result is a bit of a jumble.



The screenshot shows the same Microsoft Internet Explorer window, but the table data is now properly formatted with a grid border, making it much easier to read.

| au_id | au_name | au_fname | phone | address | city | state | zip | contract |
|-------------|------------|----------|--------------|----------------------|---------------|-------|-------|----------|
| 172-32-1176 | White | Johnson | 408 496-7223 | 10932 Bigge Rd. | Menlo Park | CA | 94025 | True |
| 213-46-8915 | Green | Marjorie | 415 986-7020 | 309 63rd St. #411 | Oakland | CA | 94618 | True |
| 238-95-7766 | Carson | Cheryl | 415 548-7723 | 589 Darwin Ln. | Berkeley | CA | 94705 | True |
| 267-41-2394 | O'Leary | Michael | 408 286-2428 | 22 Cleveland Av. #14 | San Jose | CA | 95128 | True |
| 274-80-9391 | Straight | Dean | 415 834-2919 | 5420 College Av. | Oakland | CA | 94609 | True |
| 341-22-1782 | Smith | Meander | 913 843-0462 | 10 Mississippi Dr. | Lawrence | KS | 66044 | False |
| 409-56-7008 | Bennet | Abraham | 415 658-9932 | 6223 Bateman St. | Berkeley | CA | 94705 | True |
| 427-17-2319 | Dull | Ann | 415 836-7128 | 3410 Blonde St. | Palo Alto | CA | 94301 | True |
| 472-27-2349 | Gringlesby | Burt | 707 938-6445 | PO Box 792 | Covelo | CA | 95428 | True |
| 486-29-1786 | Locksley | Charlene | 415 585-4620 | 18 Broadway Av. | San Francisco | CA | 94130 | True |



| Phone | Address | City | Phone | Address | City | State | Zip | Case |
|-------------|--------------|----------|--------------|--------------------|----------------|-------|-------|-------|
| 546-93-1872 | Blotner-Hale | Reginald | 503 745-8402 | 55 Riverside Bl. | Coryalla | OR | 97336 | True |
| 872-71-3048 | Yokomizo | Akira | 415 925-4225 | 3 Silver Ct | Walnut Creek | CA | 94595 | True |
| 712-45-1887 | del Castillo | Innes | 815 926-3275 | 2286 Cran Pl 988 | Ann Arbor | MI | 48105 | True |
| 722-51-5459 | DeFrance | Michel | 215 547-9880 | 3 Basting Pl. | Dairy | IN | 45401 | True |
| 724-68-8901 | Shinger | Del | 415 943-2951 | 5420 Telegraph Av. | Oakland | CA | 94609 | False |
| 724-88-9391 | MacFeather | Stamps | 415 354-7128 | 44 Oakland Hts | Oakland | CA | 94612 | True |
| 756-33-7361 | Karsen | Lyle | 415 524-9219 | 5720 McAuley St | Oakland | CA | 94608 | True |
| 807-61-6054 | Perkeley | Sylvia | 301 945-8853 | 1356 4-wington Pl | Rockville | MD | 20853 | True |
| 846-92-7158 | Hummer | Sheryl | 415 826-7128 | 3410 Skouse St. | Palo Alto | CA | 94301 | True |
| 893-72-1158 | McBadden | Heather | 707 448-8880 | 301 Putnam | Watville | CA | 95888 | False |
| 899-48-2035 | Rinzer | Ann | 801 826-0750 | 87 Seventh Av. | Salt Lake City | UT | 84152 | True |
| 899-72-3587 | Rilger | Albert | 801 826-0750 | 87 Seventh Av. | Salt Lake City | UT | 84152 | True |

FIGURE 11.3 AFTER: A few formatting instructions and the results are easier to read and more professional.

You can create additional useful effects, helping make the grid more readable. The DataGrid in the examples in this chapter has not been bound to a data store until runtime. For this reason, the Properties window cannot be used during design time to adjust headers and data using the Columns collection. You must make these adjustments programmatically.

This next example illustrates how to first set the DataGrid's `AutoGenerateColumns` property (also known as an attribute, if you're in an XML mood). When you make this change, the grid will not automatically display a table's column names (the field names) and the rows of data. Make this change in the properties window.

Now, type in this HTML code (in boldface) to specify how you want the data displayed:

```
<asp:DataGrid id='DataGrid1' autogeneratecolumns="False" style="Z-INDEX: 101;
 76px; POSITION: absolute; TOP: 36px runat="server"
 Width="243px" Height="280px">
<AlternatingItemStyle BackColor="#FFE0C0"></AlternatingItemStyle>
<ItemStyle BackColor="#C0C0FF"></ItemStyle>
<Columns>
<asp:BoundColumn HeaderText="Name"
 DataField="ContactName" ItemStyle-Font-Size="14"></asp:BoundColumn>
<asp:BoundColumn HeaderText="Title"
 DataField="ContactTitle" ItemStyle-Font-Size="11"></asp:BoundColumn>
<asp:BoundColumn HeaderText="City" DataField="City"
 ItemStyle-Font-Size="11"></asp:BoundColumn>
<asp:BoundColumn HeaderText="Phone Number"
 DataField="Phone" ItemStyle-Font-Bold="True"></asp:BoundColumn>
</Columns>
</asp:DataGrid>
```

As you see, you can use this technique to freely manipulate which columns are displayed (DataField), the header names, font size, boldface, and many other properties of each column. Before running this example, try using a different sample database just for practice. Try the Northwind sample. To do that, make these changes to the previous example code:

```
Initial Catalog=Northwind
FROM Customers
da.Fill(ds, "Customers")
```

Press F5 to see the adjustments you made to the format of the individual columns.

***TIP** You'll find a variety of predesigned quality formats for the DataGrid that you can add with a click of the mouse. Just right-click the DataGrid on the design window, choose Auto-format from the context menu, then select a format.*

FURTHER ADJUSTMENTS

You can make further changes to improve the readability of a DataGrid. Try now changing the Alternating BackColor property so that every other row is displayed in the new color. The AlternatingItemStyle property's BackColor property (yes, properties can have their own properties) remains the default color, making the table easier to read. Use the properties window to change the AlternatingItemStyle's BackColor property and you can see the results in the design window immediately.

Detecting Postback

When information comes from the client back to your server, it's known as *postback*. What, for example, happens when a client clicks one of the items in your DataGrid or a ListBox? Perhaps the user wants to view details about the clicked data? Or wants to order it from your catalog? In any case, you often will need to respond on the server to a user action.

As you know, an HTML page is composed on your server based on what's in the Page_Load event (shown in the previous examples in this chapter), then sent to the client. How, though, can you respond to a user clicking in their browser? How does this information get to your server, and how can you respond to it and send back a new, composed HTML page?

Here's an example. Remove the DataGrid from the previous example and replace it with a ListBox from the WebForms tab of the Toolbox. You're going to print a message at the very top of the user's browser, so in your WebForm make sure that the ListBox has been dragged a few inches down from the top of the page to leave some room for the printing.

Now you have to add an autopostback attribute to the HTML code defining the ListBox. Click the HTML tab at the bottom of the design window and add this code shown in boldface:

```
<asp:ListBox id='ListBox1" autopostback="true"
  style="Z-INDEX: 101; LEFT: 216px;
  POSITION: absolute; TOP: 160px" runat="server" "Width"="248px"
Height="320px"></asp:ListBox>
```

Alternatively, you can adjust this property in the Properties window.

In your code you check the IsPostBack property to see if this is the first time the client is viewing this page (`IsPostBack = False`) or whether the client has sent a message back and you may want to respond to that message. Type Listing 11.4 into the Form_Load event:

LISTING 11.4: DETECTING POSTBACK

```
Private Sub Page_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
    If Not IsPostBack Then
        Dim MyArray As New ArrayList
        MyArray.Add("Mary")
    End If
End Sub
```

```
MyArray.Add("Had")  
MyArray.Add("A")  
MyArray.Add("Little")  
MyArray.Add("Lamb")
```

[Team Fly](#)

 Previous

Next 

LISTING 11.5: INTERACTION VIA THE MIDDLE TIER

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

    Dim s As String = Request.Form.Item("TextBox1").ToString

    Dim r As String = Regex.IsMatch _
(s, ("^([\w-\.]*)@((\[[0-9]{1,3}\. [0-9]{1,3}\. [0-9]{1,3}\.) |
([\w-]+\.)+)([a-zA-Z]{2,4}|[0-9]{1,3})(\?)$"))

    If r = True Then "correct email format
        s &= "<BR> is a correct email address. Thank you!"

    Else
        s &= "<BR> is not a correct email address."& _
Please retype it and click the button again."

    End If

    Response.Write(s)

End Sub
```

Validation Controls

In addition to programmatic validation, you can use the new .NET validation controls. Here's an example. Add a RangeValidator control to your WebForm. Then add a TextBox, a label, and a button. Select the RangeValidator control in the design window and enter the following values in the RangeValidator's Properties window: Maximum Value: 39, Minimum Value: 3, ErrorMessage: Your number must be between 3 and 39, ControlToValidate: TextBox1, Type: Integer.

Here is a description of the job performed by each validator:

RequiredFieldValidator Ensures that the user fills in a required entry.

RangeValidator Tests for upper and lower boundaries and data type. Can specify a range of dates or an alphabetic range as well as numeric.

CompareValidator Tests against a property value in a different control on the page, a value from a database, or a literal. You use the comparison operators for this test: >, =, and so on.

RegularExpressionValidator Tests against a given regular expression.

CustomValidator Allows programmatic testing. Specifies a procedure that validates the input (either client-side or server-side).

In the following sections we describe in detail the two most important validation controls: the RangeValidator and the RegularExpressionValidator.

THE RANGE VALIDATOR

Change the Label's ID property to lblResponse. Double-click the button to get to its Click event in the code-behind window and type this in:

```
Private Sub Button1_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles Button1.Click  
    RangeValidator1.Validate()  
    If Page.IsValid = True Then  
        lblResponse.Text = "Your entry is valid"  
    End If  
End Sub
```

Notice the strange double reference following the Handles command: both the Click and Validate events are handled by this procedure. The Validate method of the RangeValidator control is first fired, and then the Page object's IsValid property is tested. If it's True (meaning that the user entered a valid number), you display a message in your label telling the user they did well. If the user types in a number that's out of range (such as 2 or 55), the RangeValidator's ErrorMessage property is displayed instead. Go ahead and press F5 and see the effect of entering valid and invalid numbers.

Perhaps you're thinking "What the heck? Why not just write the following code in the button's Click event?"

```
If CInt(TextBox1.Text) < 3 Or CInt(TextBox1.Text) > 39 Then  
    Response.Write("Must be between 3 and 39")  
End If
```

This would work, but you get more flexibility using validation controls, and when you have to validate more than one control on a page it's efficient to use a validation control. Also, validation controls give you the ability to check either individual data (individual TextBox entries, for instance, using `RangeValidator1.IsValid`) or the validity of the entire page at once (using `Page.IsValid`). You can also specify the text, location, and appearance of error messages generated by the validation controls. Finally, and perhaps best of all, the validation process is adaptable and decides for itself whether it can run as a client-side script (if the browser permits this) or, if necessary, on the server. You, the programmer, need not intervene in this process. The best efficiency is achieved automatically.

If you want to consolidate error messages, the ValidationSummary control displays any error messages from all validation controls on a given WebForm. You can choose to display these messages via a message box, or as above directly in the browser.

Warning: All the validation controls check data only if there is some. In other words, an empty TextBox is *not* validated (put another way, the empty response is treated as valid). This could cause problems unless you add a RequiredFieldValidator control to ensure that the tested control isn't empty.

You have to add a separate validation control for every input control you want to validate. During postback, the page object sends the text to the validation control that has its ControlToValidate property set to a particular TextBox or other control.

However, it's possible to point more than one validation control to a single TextBox (or other control). Perhaps you must ensure both a correct range as well as ensuring that the TextBox is not empty. In that case, you would set the ControlToValidate properties of both a RangeValidator and a RequiredFieldValidator to point to that TextBox.

THE REGULAR EXPRESSION VALIDATOR

What, though, should you do if you want to allow more than a single valid pattern? How about these two valid zip code patterns: 14486 or 14486-6303? In this case, you don't add multiple validation controls; instead, use the `RegularExpressionValidator` and specify that two valid patterns are permitted. The `RegularExpressionValidator` compares what the user enters against a Regex pattern and can be used for various common situations such as e-mail address, social security number, credit card number, zip code, phone number, and so on. Also, the programmer can specify two or more acceptable patterns simultaneously.

A set of commonly used Regex expressions are included with the `RegularExpressionValidator`. In the properties window, click the ellipsis next to the `ValidExpression` property and you'll see the handy Regular Expression Editor dialog box, as shown in Figure 11.4.

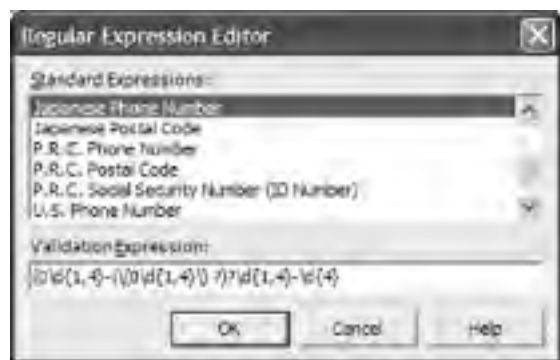


FIGURE 11.4 With this editor, the problem of figuring out several common Regex expressions is solved.

Now you can just select it, rather than having to construct `(0\d{1,4}-|\(0\d{1,4}\)\)?\d{1,4}-\d{4}` as the correct validation pattern match for Japanese phone numbers.

The page object sends data to a validation control for testing, and that control then checks the input and sets a property specifying whether or not the test passed. In the example above, the `IsValid` property of the page object is tested to see if it's set to `True` or `False`. If you have multiple validation controls on a given page, and *any* them rejects some data, the `Page.IsValid` property will be set to `False`. For this reason you normally first test the `Page.IsValid` property in your code, then if it turns out to be `True`, you can just accept the user's input without further testing.

If the `Page.IsValid` property tests `False`, you would probably want to figure out which validation controls are individually set to `False`:

```
If RangeValidator1.IsValid = False Then
```

Or you can loop through the `Validators` collection, like this:

```
x = page.validators(0).IsValid
```


Note that some validation controls test more than a single factor at a time. This example RangeValidator tests both for a range and that the data type is an integer:

```
<asp:RangeValidator id='RangeValidator1'  
  style="Z-INDEX: 105; LEFT: 640px; POSITION: absolute; TOP: 304px"  
  runat="server" Width="136px" Height="80px"  
  ErrorMessage="Boo Boo!" MaximumValue="3" MinimumValue="39"  
  Type="Integer"></asp:RangeValidator>
```

[Team Fly](#)

 Previous

Next 

Making this problem even more annoying is the fact that *not all* collections in .NET begin with zero; some collections begin with one! You have to memorize the various exceptions to the rule, or more likely, you just have to add fudge factors when the old `IndexOutOfRangeException` pops up now and then in your code.

You can use the `UBound` function (or the `Count` property of an `ArrayList`) to get the *accurate* highest element number:

```
UBound(myarray)
```

Or you can use the `Length` property to find out how many actual elements are in the array, like this:

```
myarray.Length
```

In this next example, the `UBound` function returns 10, but the `Length` property is 11:

```
Dim MyArray(10) As String
    Console.WriteLine('UBound is: ' & UBound(MyArray))
    Console.WriteLine("Length Property is: " & MyArray.Length)
```

The `Option Base 1` statement was made available in earlier versions of VB precisely to repair the absurdity of zero-based collections.

Nonetheless, the .NET Common Language Specification requires zero-based arrays. The zero-based array is one example of how .NET requires VB to conform to the way the C language and its offspring—C++, C#, Java, and so on—have always done things. In my view, it would have been better to add a commonsensical one-based lower boundary to collections in .NET (and to force the C-type languages to get logical) than to make VB.NET use zero-based collections (with some annoying exceptions).

The zero-based annoyance can be found in various areas of .NET. For example, the following VB6 function returns `ABC` because when you specify that you want the string starting at the first character (`, 1`), you *mean* the first character—`A` in this case:

```
Dim x As String
x = Mid("ABC", 1)
MsgBox(x)
```

When you run this example, you get `ABC` as you would expect.

Now, what do you suppose happens when you use the equivalent VB.NET function `Substring` and specify `1` as the first character?

```
Dim x As String = "ABC"
MsgBox(x.Substring(1))
```

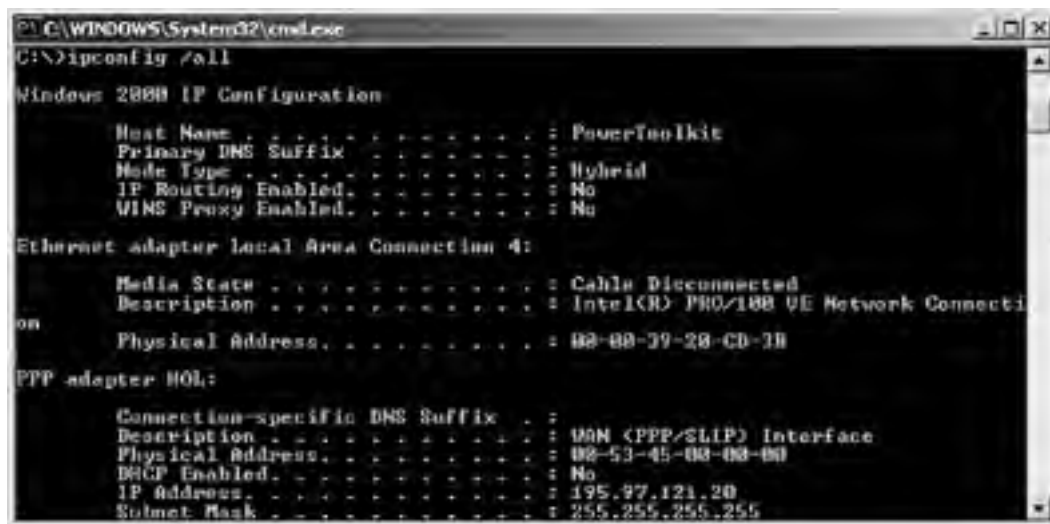
Perhaps you're not stunned, but I was. When I first tried this example, I thought I was saying to VB.NET that I wanted a string starting at the first character (`1`), which would give me back `ABC`. Nope. You get `BC`. In VB.NET, this `(1)` actually means `(2)`. The string that is returned

to you starts with the second character, even though you used (1) in the source code. This bizarre phenomenon, they tell me, is deliberate, by design, and, in some way, good. *How* it's good escapes me.

When ergonomics is, at long last, introduced into computer-language design, we'll get rid of inane, human-unfriendly usages like the zero-based set. All collections should begin with 1, obviously. Why should we have to memorize exceptions to a rule that is a very bad rule in the first place?

to 255, such as 192.140.39.101. This is known as *dotted-quad* notation. When you connect to the Internet through an Internet Service Provider (ISP), your computer is assigned an IP address automatically by the ISP. This address remains the same for the duration of the session; the next time you connect to the Internet, you get a new IP address. It is possible to request a fixed IP address from your ISP, and more and more people have a fixed IP address on the Internet. Having a fixed IP address means that people can always find you by your IP address.

To find out your computer's IP address, connect to the Internet as usual and then run the IPCONFIG utility, which is shown in Figure 12.1. Open the Start menu, select Run, and enter CMD in the dialog box that will pop up. When the command prompt window appears, type **IPCONFIG /all** and press Enter.



```
C:\WINDOWS\System32\cmd.exe
C:\>ipconfig /all

Windows 2000 IP Configuration

Host Name . . . . . : PowerTookIt
Primary DNS Suffix . . . . . :
Node Type . . . . . : Hybrid
IP Routing Enabled. . . . . : No
WINS Proxy Enabled. . . . . : No

Ethernet adapter Local Area Connection 4:

Media State . . . . . : Cable Disconnected
Description . . . . . : Intel(R) PRO/100 VE Network Connecti
on
Physical Address. . . . . : 08-00-29-28-CD-3B

PPP adapter H0L:

Connection-specific DNS Suffix . . . . . :
Description . . . . . : WAN (PPP/SLIP) Interface
Physical Address. . . . . : 00-53-45-00-00-00
DHCP Enabled. . . . . : No
IP Address. . . . . : 195.97.121.20
Subnet Mask . . . . . : 255.255.255.255
```

FIGURE 12.1 The IPConfig utility

In addition to their IP address, computers on the Internet have friendly names too, which are known as hostnames. You can find your computer's name in the Network Identification tab of your computer's Properties dialog box. If you're on a network with a registered domain name, your computer's friendly name is something like `name.domain.com`, where `name` is your computer's name and `domain.com` is the company's domain name. Users on a network with its own domain don't need fixed IP addresses, because their hostnames are unique.

Beyond the IP address you're assigned by your ISP when you connect to the Internet, your computer has another IP address on the local network (if you're on a local area network). The local network IP address was assigned to each computer on the local network by the administrator, or by a special program that runs on a server and assigns an IP address to each computer as they're turned on. This is what the DHCP (Dynamic Host Configuration Protocol) utility does; it assigns a unique IP address to each workstation of the network, as they connect.

If you're on a corporate network, you probably get on the Internet via a proxy server. The proxy server is an intermediary computer between the computers on the local network and the Internet. All the computers on the local network access the Internet through the proxy server. In addition, the proxy server can protect the local network from outside attacks.

It's also possible to share a fast Internet connection among multiple computers on the network by sharing one computer's Internet connection (the computer that's connected to the Internet through a DSL or an even faster line). No matter how you connect to the Internet from a local network, your computer has an IP address for the Internet, and another one for the local network. You can connect

to other clients on the same local network with either IP address, or simply with the client's host-name. Users on the Internet can connect to your computer only if they know the IP address assigned to your computer when you connected to the Internet. Of course, knowing a computer's address on the Internet doesn't mean that you can do anything more than request a connection to it. Whether you will actually connect and see its resources depends on the remote computer's security settings. The classes we'll discuss in this chapter enable two computers to communicate with one another on a specific port. The computer that accepts the request will decide whether to honor the request or to reject it. An application should be running on both machines that will monitor the specific port to which the other computer is sending information.

Finally, there's a special IP address that identifies the local computer, and it's 127.0.0.1. This address refers to the local computer (the computer on which the program is running), regardless of whether the computer is on the Internet, a local area network, or both. You will use this address to test all applications presented in this chapter if you're using a stand-alone computer.

To deal with Internet addresses, the .NET Framework provides the `System.Net.Dns` class, which exposes several methods. All methods of this class make use of the `IPAddress` and `IPHostEntry` classes. The two classes are similar and they allow you to discover IP addresses from hostnames, or hostnames from IP addresses. The `IPAddress` class represents IP addresses and the `IPHostEntry` represents a remote host. To create an `IPAddress` object, call the class's constructor passing as argument an IP address in long format, or an array of 4 bytes whose values correspond to the four groups of a dotted-quad address.

To create an `IPHostEntry` object we usually call the `GetHostByName` method of the `Dns` class, passing as argument the name of the remote host:

```
Dim remoteHost As IPHostEntry = Dns.GetHostByName(www.servername.com)
```

Then you can use the `remoteHost` object's properties and methods to retrieve information about the remote host. The most important member of the `IPHostEntry` class is the `AddressList` property, which is an array of IP addresses to which the server responds. Most servers are associated with a single IP address, but some servers may respond to multiple addresses.

Another interesting class is the `IPEndPoint` class, which represents a specific port on a specific server (an IP address and a port number). Two applications running on remote computers that talk to each other use two `IPEndPoint` objects to bind a socket to a specific port on the remote computer. The following statements create an `IPEndPoint` object that represents a specific port on the local machine. The first statement creates an `IPHostEntry` object, the `IPHost` variable that represents the local machine. The second statement retrieves the IP address associated with the local host (127.0.0.1) and combines it with a port number to create an `IPEndPoint` object:

```
Dim IPHost As IPHostEntry = Dns.Resolve("localhost")  
Dim IPEP As New IPEndPoint(IPHost.AddressList(0), 5001)
```

The first 5,000 ports are reserved, but you can use any port number after 5000. The IPHostEntry class exposes the following properties:

AddressList property This property returns (or sets) a list of IP addresses associated with a hostname, which is passed to the method as a string. This property returns an array of addresses, should the host have multiple addresses. You can't set the address of the local client, but you can create an IPHostEntry that represents a remote computer and sets its AddressList property.

[Team Fly](#)

 Previous

Next 

negative acknowledgment about the successful delivery of the packet. TCP connections, on the other hand, will verify the successful delivery of each and every packet and they will retransmit it in case of delivery failure. As a rule of thumb, the UDP protocol is used to transmit small packets in situations where reliability is not of critical importance. The TCP protocol has a significant overhead and is used in applications that transfer large packets of data. You can use the UDP protocol and still implement your own mechanism to confirm the arrival of each packet at its destination.

To create a socket, you must use the Socket class's constructor, which accepts three arguments:

```
Dim listener As new Socket(AddressFamily, SocketType, ProtocolType)
```

We call the instance of the Socket class *listener*, because we'll use it to listen to incoming requests on a specific port. The AddressFamily argument determines the type of address the socket is bound to and its value is a member of the AddressFamily enumeration. For a complete listing of the members of the AddressFamily enumeration, see the documentation. Table 12.1 lists the most common address types.

TABLE 12.1: SELECTED MEMBERS OF THE ADDRESSFAMILY ENUMERATION

| MEMBER NAME | DESCRIPTION |
|----------------|--|
| AppleTalk | AppleTalk address |
| Atm | ATM services address |
| DecNet | DECnet address |
| Ecma | European Computer Manufacturers Association (ECMA) address |
| InterNetwork | Address for IP version 4 |
| InterNetworkV6 | Address for IP version 6 |
| Ipx | IPX or SPX address |
| Irda | IrDA address |
| NetBios | NetBIOS address |
| Sna | IBM SNA address |
| Unix | Unix local to host address |

In this chapter we'll use only the InterNetwork type for our examples. This type of address is used by the Internet Protocol (IP) and represents dotted quad addresses, such as 192.168.0.100. In the future, it's likely that InterNetworkV6 addresses will dominate, as we're running out of IP version 4 addresses. The best approach is not to specify the type of the address, but start with the actual address and request its type.

The second argument is the type of the socket, which is a member of the SocketType enumeration, shown in Table 12.2.

[Team Fly](#)

 Previous

Next 

TABLE 12.2: THE SOCKETTYPE ENUMERATION

| MEMBER | DESCRIPTION |
|-----------|---|
| Dgram | Supports short messages that do not require a dedicated connection (UDP protocol); should be used with relatively small messages. |
| Raw | Supports the implementation of custom communication protocols; you're responsible for forming the IP headers of the packages. |
| Rdm | Supports messages that do not require a dedicated connection. Rdm (Reliably-Delivered Messages) messages arrive in the order in which they were sent and the sender is notified if messages are lost. |
| Seqpacket | Supports messages that require a dedicated connection. It's a reliable two-way transfer of byte streams across a network. |
| Stream | Supports messages that require a dedicated connection (TCP protocol). |
| Unknown | Specifies an unknown Socket type. |

For UDP sockets you'll set the socket's type to *Dgram*, and for TCP sockets you'll set the socket's type to *Stream*.

The last argument is the protocol that will be used by both the local and remote application and its value is a member of the *SocketType* enumeration, which includes 15 members. In this chapter we'll use the *Udp* and *Tcp* members, which correspond to a socket type of *Dgram* and *Stream*, respectively.

Once you've set up a socket, you must bind it to a specific port. Any remote application that needs to talk to your application must send its requests to this port, using the same protocol as the socket. To bind a socket to a port, use the *Socket* class's *Bind* method, which accepts as arguments an IP address and the port number. When accepting requests, the IP address is that of the current machine. The port must be any free port's number. Port 80, for example, is used for HTTP requests. The first 5,000 port numbers are reserved (not that they're used, of course), so we'll use port numbers starting with port number 5001. The following statements bind the `listener` socket to the `localEP` endpoint:

```
Dim localEP As new IPEndPoint(IPAddress.Any, 5001)  
listener.Bind(localEP)
```

The `localEP` variable represents an IP addressable destination and is constructed by specifying the current machine's IP address and a port number. The *Any* member of the *IPAddress* enumeration indicates that the server should listen for client requests on all network interfaces.

UDP and TCP sockets are used differently, so we'll demonstrate their use (and their differences) with two similar examples. We're going to build two pairs of applications that exchange information. The UDPServer and UDPClient applications exchange information through UDP classes, while the TCPServer and TCPClient applications exchange information through the TCP class. The first couple of examples are very simple: the client sends data to the server, which simply receives it and displays it.

In real life, both computers need to send and receive data at the same time. This exchange of data must also take place asynchronously. We'll show you how to build a chatting application toward the end of the chapter, but we'll start with a simpler example that will help you understand the basics of socket programming.

Using UDP Sockets

To enable two applications to talk to one another through the UDP protocol, you must create two UDP sockets. Then you bind them to a specific port on the local machine. This is the port to which they'll be listening for incoming requests. In other words, each application will use the other application's port to send requests. Once the two applications start listening to their designated ports, they can send messages to one another. To simplify the discussion, we'll assume that the first application sends data to the other (the UDP client) and the second application (the UDP server) reads the incoming data and displays them to the user.

To send a message, the client must call the Send method of the Socket class passing as argument an array of bytes that holds the data to be transmitted and the destination. This operation can be executed synchronously, because it doesn't take long (unless the remote server isn't accepting requests, of course). The server is a bit more complicated, because we can't call a synchronous method and then wait for some data to arrive. There are two methods to accept incoming data with the UDP protocol without freezing our application's interface until some data arrives at the port: by starting a new thread that listens for incoming messages or by calling the socket's Poll method, which waits for a specified number of microseconds and then returns a True/False value to indicate whether there are data to be read. To actually read the data, we use the Receive method of the socket. The Receive method reads all the data waiting at the specified port and returns them in an array of bytes.

Start a new project, the UDPServer project, and place a TextBox and a Button control on the form, as shown in Figure 12.2. The TextBox is where the incoming messages will be displayed.

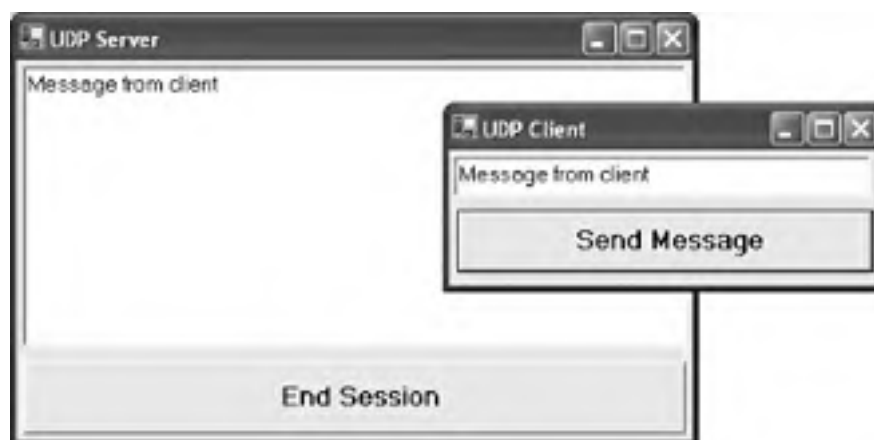


FIGURE 12.2 The UDPServer application receives messages from the UDPClient application and displays them on a TextBox control.

Then import the following namespaces to simplify your code:

```
Imports System.Net  
Imports System.Net.Sockets
```

The button on the form creates a Socket object that listens for requests to a specific port. Listing 12.1 is the code behind the Start Listening button.

[Team Fly](#)

 Previous

Next 

LISTING 12.1: THE UDPSERVER APPLICATION'S CODE

```
Dim listener As New Socket(AddressFamily.InterNetwork, _
                           SocketType.Dgram, ProtocolType.Udp)
Private Sub btnListen_Click(ByVal sender As System.Object, _
                            ByVal e As System.EventArgs) _
    Handles btnListen.Click

    Dim localEP As New IPEndPoint(IPAddress.Any, 5001)
    listener.Bind(localEP)
    While True
        If listener.Poll(1000, SelectMode.SelectRead) Then
            Dim bytesToRead As Integer = listener.Available
            Dim buffer(bytesToRead) As Byte
            listener.Receive(buffer, bytesToRead, SocketFlags.None)
            Dim message As String = _
                System.Text.Encoding.ASCII.GetString(_
                    buffer, 0, bytesToRead)
            If (message.ToUpper = "QUIT" ) Then
                Exit While
            Else
                Console.WriteLine(message)
                TextBox1.AppendText(message & vbCrLf)
            End If
        End If
        Application.DoEvents()
    End While
    listener.Close()
    Application.Exit()
End Sub
```

To terminate the session, the client must send the string "QUIT." The server has no way to terminate the session, because it's not sending anything to the client. You can shut down the server, but this won't even cause a runtime exception at the client. The client will happily continue to send messages to the server, which will never see them or acknowledge them.

Notice that the server application uses an endless loop to monitor for incoming requests and that it calls the `DoEvents` method to give the UI a chance to react to user events. Later in this chapter, you'll see how to monitor for incoming messages from within a separate thread.

Let's switch to the `UDPClient` application. Since the UDP protocol doesn't require a dedicated connection, we simply establish a connection to the remote server and send a message. Listing 12.2 shows the code behind the Send Message button. The Send Message button is the default one on the form. You can send the message by hitting Enter at the end of each line.

LISTING 12.2: THE UDPCCLIENT APPLICATION'S CODE

```
Private Sub Button1_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
    Handles btnSend.Click
    Dim RemoteServerName As String = "localhost"
    Dim message As String = TextBox1.Text
    Dim localSocket As New Socket(AddressFamily.InterNetwork, _
                                  SocketType.Dgram, ProtocolType.Udp)
    Dim localEP As New IPEndPoint(IPAddress.Any, 5002)
    localSocket.Bind(localEP)
    Dim buffer() As Byte = _
        System.Text.Encoding.ASCII.GetBytes(message)
    Dim ServerAddress As IPAddress = _
        Dns.Resolve(RemoteServerName).AddressList(0)
    Dim remoteEP As New IPEndPoint(ServerAddress, 5001)
    localSocket.SendTo(buffer, 0, buffer.Length, _
                       SocketFlags.None, remoteEP)
    localSocket.Close()
    TextBox1.Text = ""
End Sub
```

The client starts by creating a socket, the `localSocket` object, which it will use to send data to the server. The `localSocket` object is bound to port 5002 of the local computer. Notice that you must change the value of the `RemoteServerName` variable to the name of the machine on which the server is running, if you're testing the applications on two different machines. Then it creates a new `IPEndPoint` object, the `remoteEP` variable, that represents the remote port. This variable is passed as argument to the `SendTo` method of the local socket to transmit the data to the server.

In summary, the server application listens at a specific port for incoming requests, from any address on port 5002. The client creates a socket and uses it to send data to the client. Since we're using the same computer for both the server and the client, the client and server sockets are bound to two different ports. The client needs to know the port to which the server is listening and send the messages there. If sent to another port, the server will never receive the messages.

Using TCP Sockets

With TCP you must first establish a link between the two computers. This link must remain alive during the session and all messages sent to the other machine must go through this link. To exchange data with a remote computer using the TCP protocol, both the client and server applications must create a new socket and bind it to a port. You already know how to do this. Then you must call the `Listen` method of the server's `Socket` object to start listening to incoming requests.

To use the two applications, run the TCPServer project and click the Start Listening button on the form. This will cause the server to start listening for incoming requests. Start the client application and click the button Establish Connection To Server to create a link to the designated port on the remote server. Then start typing messages in the TextBox control of the client application and press Enter to transmit each message to the server.

[Team Fly](#)

 Previous

Next 

The code behind the Start Listening button on the form of the TCPServer project is shown in Listing 12.3. First, we create an `IPEndPoint` to represent the port at which the program will listen for incoming data. Then we create the `TCPServer` socket, bind it to this `IPEndPoint`, and call the `Listen` method of the `Socket` object. The `Listen` method accepts as argument the maximum number of requests that may be pending. This argument lets you specify the number of requests that can be queued while the socket accepts a request, and its value is usually an integer from 1 to 5.

At this point the server is ready to accept a connection request from a client with the `Accept` method. The `Accept` method of the `Socket` class accepts a connection request on a specific port. The method doesn't authenticate the requests; it simply accepts the first request that arrives at the port from any client. It's your responsibility to pass authentication information to the server from the client.

LISTING 12.3: THE TCPSERVER APPLICATION'S CODE

```
Private Sub Button1_Click(ByVal sender As System.Object, _
                        ByVal e As System.EventArgs) _
    Handles Button1.Click
    Dim IPEndPoint As IPEndPoint = Dns.Resolve("localhost")
    Dim IPEP As New IPEndPoint(IPEndPoint.AddressList(0), 5001)
    Dim TCPServer As New Socket(IPEP.AddressFamily, _
                                SocketType.Stream, ProtocolType.Tcp)

    TCPServer.Bind(IPEP)
    TCPServer.Listen(2)
    Dim msgSocket As Socket = TCPServer.Accept()
    Console.WriteLine("Server accepting TCP requests")
    Dim DataIn(8192) As Byte
    Dim bytesRead As Integer
    While True
        Application.DoEvents()
        Try
            bytesRead = msgSocket.Receive(DataIn)
        Catch exc As Exception
            Console.WriteLine(exc.Message)
        End Try
        If bytesRead > 0 Then
            Dim message As String = _
                Encoding.Unicode.GetString(DataIn, 0, bytesRead)
            If message.ToUpper = "QUIT" Then
                TCPServer.Close()
                End
            Else
                Console.WriteLine(message)
                TextBox1.AppendText(message)
            End If
        End If
    End While
    msgSocket.Close()
End Sub
```

The server's interface will freeze until a request from a client arrives, because the `Accept` method is synchronous. Once the incoming request has been accepted, the code enters an infinite loop, where it keeps calling the `Receive` method to read data sent by the client. The `Receive` method is also synchronous, so the server application's interface will freeze until a message arrives from the client. The server application's interface will be updated every time a new message is received. The program prints the incoming messages on a `TextBox` control, as well as on the `Output` window. The client application declares a `Socket` variable at the form level, because it must be accessed from two different procedures: the procedure that establishes a connection to the server and the procedure that sends data to the server.

```
Dim TCPSocket As Socket
```

The code behind the "Establish Connection to Server" is shown in Listing 2.4. The code creates an `IPEndPoint` to represent a specific port on the remote server and then uses it to construct a `Socket` object, the `TCPSocket` object. The `Socket` class's `Connect` method establishes a connection to the remote server. This is another synchronous method, which is usually called from within a separate thread.

LISTING 12.4: THE TCPCLIENT APPLICATION'S CODE

```
Private Sub btnConnect_Click(ByVal sender As System.Object, _  
                             ByVal e As System.EventArgs) _  
                             Handles btnConnect.Click  
    Dim ipEnd As New IPEndPoint(_  
                             Dns.Resolve("localhost").AddressList(0), 5001)  
    TCPSocket = New Socket(ipEnd.AddressFamily, _  
                           SocketType.Stream, ProtocolType.Tcp)  
    Try  
        TCPSocket.Connect(ipEnd)  
        btnConnect.Enabled = False  
    Catch exc As Exception  
        Console.WriteLine(exc.Message)  
    End Try  
End Sub
```

To send a message to the server, the client application calls the `Socket` class's `Send` method, passing as argument an array with the bytes to be sent to the server. This method is called every time the user presses `Enter` in the `TextBox` control on the top of the form, as shown in Listing 12.5.

LISTING 12.5: SENDING MESSAGES TO THE TCPSERVER

```
Private Sub TextBox1_KeyUp(ByVal sender As Object, _  
                           ByVal e As System.Windows.Forms.KeyEventArgs) _  
                           Handles TextBox1.KeyUp  
    If e.KeyCode = Keys.Enter Then  
        Dim buffer As Byte() = Encoding.Unicode.GetBytes(TextBox1.Text)  
        TCPSocket.Send(buffer)  
        If TextBox1.Text.ToUpper = "QUIT" Then
```

Initialization

In VB.NET you can assign a value to a variable when you declare it. This is referred to as using "initializers": `Dim s As String = "This"`. This same feature is available to arrays, but you must use braces:

```
Dim MyArray() As String = {"Clark", "Lois Lane", "Jimmy"}
```

You can't specify an upper boundary when initializing array values in this fashion. The `()` must be left empty, as this example illustrates.

Arrays of Objects

All data in .NET are technically objects. Also, you can create an array of objects (in this way you can store different data types within the same array). To create an object array, you first declare an object variable, then instantiate each object in the array. The example in Listing 2.3 creates an array holding six book objects.

LISTING 2.3: CREATING AN ARRAY OF OBJECTS

```
Public Class Form1

    Inherits System.Windows.Forms.Form

    Dim arrBook(6) As Book 'create the array object variable
    (Windows Form Designer generated code)

    Private Sub Form1_Load(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles MyBase.Load
        Dim i As Integer

        'instantiate each member of the array:
        For i = 0 To 6
            arrBook(i) = New Book()
        Next

        ' set the two properties of one of the array members
        arrBook(3).Title = "Babu"
        arrBook(3).Description = "This book is large."

        Dim s As String = arrBook(3).Title
        Console.WriteLine(s)

    End Sub
```

Both the server and the client applications use threads to perform the operations that would normally freeze the UI. The server uses two threads that run in the background: one that listens for incoming requests and another one that monitors the designated port for incoming requests. The client application uses a background thread to monitor for incoming messages. Basically, all the operations on the server take place in the background from within threads, while the clients run a single background thread each to monitor incoming messages from other users. New messages are sent by the client in the foreground thread.

To test the TCPChat application, start the TCPChatServer and click the Start Listening button. The server application is ready to accept requests. Then start one or more instances of the TCPChatClient application. The TCPChatClient application will prompt you to specify a username by which you'll be known to the other users. The name should be unique, but the application won't reject duplicate names. You can add the appropriate code to the server application to reject connection requests if the client name is not unique.

The TCPChatServer Application

The server application uses two classes that implement the two different threads: the ListenClass, which exposes the StartListening method, and the ChatClass, which exposes the StartChat method. The two methods are the procedures that run in separate threads. In addition, the two classes provide a number of properties that allow them to exchange information with the main application and one another. They also raise events to notify the application about the connection of a new user (the Connected event of the ListenClass class and the LineArrived event of the ChatClass class). Listing 12.6 shows the implementation of the ListenClass.

LISTING 12.6: THE LISTENCLASS OF THE TCPSEVER APPLICATION

```
Class ListenClass
    Public client As TcpClient
    Public Event Connected(ByVal tcpClient As TcpClient)
    Public localListener As TcpListener

    Public Sub StartListening()
        If localListener Is Nothing Then
            localListener = New TcpListener(Dns.Resolve
                ("localhost").AddressList(0), 5001)
        End If
        localListener.Start()
        client = localListener.AcceptTcpClient
        Console.WriteLine(localListener.LocalEndpoint)
        RaiseEvent Connected(client)
    End Sub

    Public Sub StopListening()
        localListener.Stop()
    End Sub
End Class
```

The StartListening method sets up the localListener object, which is a TCPListener object. To create this object, it passes to its constructor the address of the remote host and the remote host's port that will be used to intercept incoming requests (port 5001). Don't forget to change the name of the "localhost" host if you plan to deploy the server application on a different computer.

To start listening for incoming requests, the TCPListener class provides the Start method, which doesn't require any arguments. This is equivalent to the Listen method of the Socket object. Then the code calls the AcceptTcpClient method, which is synchronous and accepts a request from a client. The AcceptTcpClient method returns a TcpClient object, which is stored in one of the ListenClass class's public properties. The same object is passed to the main application as argument of an event handler. As soon as the server accepts a client request, it fires the Connected event.

As you can see, the StartListening method accepts a single request and then exits. In the code that handles the Connected event (which you'll see shortly), we start another instance of the same thread, which keeps listening for the next client request.

The ChatClass class is a bit more involved, because it doesn't simply accept a request. The StartChat method keeps monitoring each client stream for incoming data. Listing 12.7 shows the implementation of the ChatClass.

LISTING 12.7: THE CHATCLASS OF THE TCPCHATSERVER APPLICATION

```
Class ChatClass
    Public clientStream As NetworkStream
    Public clientStreams As New ArrayList
    Public Event LineArrived(ByVal UserID As Integer, _
                            ByVal txtLine As String)
    Public Event ConnectionClosed(ByVal clientID As Integer)

    Public Sub StartChat()
        Dim line As String
        Dim buffer(9999) As Byte
        Dim iClient As Integer
        For iClient = 0 To clientStreams.Count - 1
            Dim user As Integer = iClient
            Dim i As Integer = 0
            Try
                If iClient < clientStreams.Count Then
                    While clientStreams(iClient).DataAvailable
                        buffer(i) = clientStreams(iClient).ReadBy
                        i += 1
                    End While
                    If i > 0 Then
                        line = System.Text.Encoding.UTF8. _
                            GetString(buffer, 0, i)
                        RaiseEvent LineArrived(user, line)
                        If line.ToUpper = "QUIT" Then
                            RaiseEvent ConnectionClosed(iClient)
                        Exit For
                    End If
                End Try
            End For
        End Sub
```



```
                End If
            End If
        End If
        Catch exc As Exception
            RaiseEvent ConnectionClosed(iClient)
            MsgBox("Connection closed" & vbCrLf & exc.Message)
            Exit Sub
        End Try
    Next
End While
End Sub
End Class
```

Each client is assigned a `NetworkStream` object when it connects to the server. This stream is used to exchange data with the specific client. The `StartChat` method monitors the `NetworkStream` object associated with each client. If there are data to be read (the `DataAvailable` property is `True`), it reads them one byte at a time with the `ReadByte` method. For each line of text that arrives from a client, the method raises the `LineArrives` event. We pass the name of the user who sent the message and the message itself to the main application through the arguments of this event.

When the server is started, you must click the `StartListening` button on the application's form. This button's code starts the `StartListening` method of the `ListenClass` class in a separate thread, as shown in Listing 12.8:

LISTING 12.8: LISTENING FOR REQUESTS ON A SEPARATE THREAD

```
Private Sub btnStartListening_Click(ByVal sender As System.Object, _
                                   ByVal e As System.EventArgs) _
    Handles btnStartListening.Click
    Dim TSThread As Thread
    TSThread = New Thread(AddressOf LC.StartListening)
    TSThread.Priority = ThreadPriority.Normal
    TSThread.Start()
End Sub
```

When a client request is accepted by the server, the `Connected` event is fired. In this event's handler we read the name of the user at the client (the first line transmitted by the client is a username), and we create a new `Participant` object and add this object to the `Participants` collection. The `Participant` object is an instance of a custom class that holds information about a chat participant; its definition is as follows:

```
Public Class participant
    Public Name As String
    Public ClientStream As NetworkStream
End Class
```

For each participant we store a name and a `NetworkStream` object that represents the stream of the client. We'll use this stream to send data to the specific participant. After creating a `Participant` object, the code starts the `StartListening` method again on a new thread to listen for additional client requests and the `StartChat` method (on a separate thread) to monitor for incoming messages. The code that handles the `Connected` event is shown in Listing 12.9.

LISTING 12.9: HANDLING THE CONNECTED EVENT

```
Private Sub LC_Connected(ByVal client As TcpClient) Handles LC.Connected
    Dim remoteClient As TcpClient
    remoteClient = client
    Dim CThread As New Thread(AddressOf CC.StartChat)
    CC.clientStream = LC.client.GetStream
    ' Read client's name
    While Not CC.clientStream.DataAvailable
    End While
    Dim i As Integer
    Dim buffer(99) As Byte
    While CC.clientStream.DataAvailable
        buffer(i) = CC.clientStream.ReadByte
        i += 1
    End While
    ' Create a new participant
    Dim p As New participant
    p.Name = System.Text.Encoding.UTF8.GetString(buffer, 0, i)
    p.ClientStream = CC.clientStream
    participants.Add(p)
    ' Add new participant's stream to CC object (ChatClass's instance
    CC.clientStreams.Add(CC.clientStream)
    txtMessages.AppendText(''User " & p.Name & _
        " joined the chat" & vbCrLf)
    ' Start the thread that receives and broadcasts messages
    If participants.Count = 1 Then CThread.Start()
    ' Start the StartListening procedure in a new thread
    ' to monitor the port at which clients make requests
    chatThread = New Thread(AddressOf LC.StartListening)
    chatThread.Priority = ThreadPriority.Normal
    chatThread.Start()
End Sub
```

Once the thread of the `StartChat` method is up and running, the server application listens for incoming messages. Notice that the `StartChat` method is called only once; only one instance of the method is needed to monitor messages sent by any client, because the messages arrive to the same port on the server, no matter which client sent them. The `CThread` thread is started only if the number of participants is 1. This condition may become true as users leave the chat session, so you'll

probably need a more reliable technique to find out whether an instance of the StartChat method is already running (you could use another global variable, for example). The StartChat method raises the LineArrived event every time a new message arrives, and this event is handled by the main application with the code shown in Listing 12.10:

LISTING 12.10 HANDLING INCOMING MESSAGES AT THE SERVER

```
Private Sub CC_LineArrived(ByVal userID As Integer, _
                          ByVal message As String) _
    Handles CC.LineArrived
    txtMessages.AppendText(message & vbCrLf)
    Dim buffer() As Byte
    ' Extract the sender's name
    Dim senderName As String =
        CType(participants(userID), participant).Name
    Dim P As participant
    buffer = System.Text.Encoding.UTF8.GetBytes(senderName & _
        "'> " & message)

    For Each P In participants

        P.ClientStream.Write(buffer, 0, buffer.Length)
    Next
End Sub
```

The event handler of Listing 12.10 is straightforward: it iterates through the members of the Participants collection and sends the same message to all clients. Because we maintain a NetworkStream object for each participant, we call the stream object's Write method to broadcast the message to the clients.

The TCPChatClient Application

The client part of the TCPChat application is the TCPChatClient application. This application monitors a specific port on the local computer for incoming messages by running a background thread that monitors a specific port. The code uses the ChatClass class as the TCPChatServer application and starts the StartChat method in a background thread from within the form's Load event handler, which is shown in Listing 12.11.

LISTING 12.11: STARTING THE BACKGROUND THREAD ON THE CLIENT APPLICATION

```
Private Sub Form1_Load(ByVal sender As System.Object, _
                      ByVal e As System.EventArgs) _
    Handles MyBase.Load
    Dim remoteEP As IPEndPoint
    Try
        remoteEP = New IPEndPoint(
            Dns.Resolve("localhost").AddressList(0), 5001)
```

```
        client.Connect(remoteEP)
    Catch exc As Exception
        MsgBox(''Chat server not found, application will terminate.'"
            vbCrLf & exc.Message)
    End
End Try
clientWStream = client.GetStream
CC.clientStream = clientWStream
' Prompt the user for a name that will be used
' to identify the current client in the chat
Dim userName As String
userName = InputBox("Enter the name with which " & _
    "you want to participate in the chat")
Me.Text = "CLIENT " & userName
' Transmit the user's name to the server.
Dim buffer() As Byte
buffer = System.Text.Encoding.UTF8.GetBytes(userName)
CC.clientStream.Write(buffer, 0, buffer.Length)
' and start a new thread in the background to monitor
' for messages sent by the chat server
CThread = New Thread(AddressOf CC.StartChat)
CThread.Priority = ThreadPriority.Normal
CThread.Start()
End Sub
```

The `client` variable is a `TCPClient` object that's declared at the form's level with the following statement:

```
Dim client As New TcpClient
```

This object is used to establish a connection to the server. The connection is established when the client object's `Connect` method is called. The `Connect` method accepts as argument an `IPEndPoint` object that represents the chat's port on the remote computer. As soon as the connection is established, the client program prompts the user for a name, with which the user will participate in the chat. The `TCPChat` application doesn't check the user's name. You can improve the application by adding some code at the client to reject connections without a unique username. When a client attempts to connect with a name that's already in use, the server application could send a specific string to the client application (a string like "Non-unique name") and terminate the connection. The client application should detect the rejection and prompt the user for another username.

After sending the user's name to the server, the client is ready to engage in a chat. It does so by starting the `StartChat` method in a background thread. Every time a new message arrives from the server, the `StartChat` method fires the `MessageArrived` event. This event's handler is trivial; it simply displays the new message on the application's main form. The `MessageArrived` event's handler is shown in Listing 12.12.

LISTING 12.12: HANDLING THE MESSAGEARRIVED EVENT

```
Private Sub MessageArrived(ByVal message As String) _  
    Handles CC.LineArrived  
    ' New message arrived, display it  
    txtParticipants.AppendText(message & vbCrLf)  
End Sub
```

The two projects that make up the TCPChat application demonstrate how to use the TCPClient and TCPListener classes to enable two computers to talk to one another with the TCP protocol. To summarize, here are the steps for exchanging data using the TCP-related classes of the .NET Framework.

On the server:

1. Create a TcpListener object and associate it with the port to which the clients will connect. Then call the TcpListener object's Start method.
2. To accept a new request, call the TcpListener method's AcceptTcpClient method, which returns a TcpClient object. Then call the TcpClient object's GetStream method to create a NetworkStream object, from which you can read the incoming messages or send messages to the specific client.
3. To find out if a NetworkStream object contains data, use the DataAvailable property. If this property is True, call the Read or ReadBytes method to read the data from the stream.
4. To send data to a client, call the Write or WriteBytes method of the same NetworkStream object, passing as argument an array of bytes with the data to be transmitted.

On the client:

1. Create a new TcpClient object and call its Connect method to establish a connection to the remote server. The Connect method accepts as argument an IPEndPoint object that represents the port at which the remote server is listening. Then call the TcpClient object's GetStream method, which returns a NetworkStream object.
2. Call the NetworkStream object's Write or WriteBytes method to send data to the client that corresponds to the NetworkStream object.
3. Monitor the same stream to find out if new data has arrived from the server. If the DataAvailable property is True, use the same stream object's Read or ReadBytes method to read the data sent by the server.

This concludes our discussion of strictly peer-to-peer programming. In the second half of this chapter we're going to explore a few more classes of the System.Net namespace, which allow you to interact with web resources. As you will see, a browser is not the only way for a Windows client to contact a web server. You can write Windows applications that contact applications running on a web server and request, or upload, data.

OpenRead method The third method for downloading data from a web server returns a Stream object, which you can use to read data just as you would with a local file. Once the Stream object has been created, you can use its Read methods to retrieve the data. The syntax of the OpenRead method is:

```
WebClient.OpenRead(documentURL)
```

The return value of the OpenRead method must be assigned to a Stream object:

```
Dim wClient As New WebClient  
Dim webStream As Stream = wClient.OpenRead(''142.18.191.10/File.html")
```

After that, you can use the Stream object's reading methods to read the data, one line at a time or the entire document. This method is convenient when you're downloading binary data, such as images. Note that none of the three methods for downloading data will return until the entire document has been downloaded.

There are also methods to upload a file to a web server. Of course, the methods for uploading a file aren't of much use on their own. You need a program at the receiving end to intercept the data and process them. The methods for uploading data to a web server are the following:

OpenWrite method The OpenWrite method creates a Stream object that acts like a channel between your application and the web server. Once the Stream object is in place, you can use its Write methods to upload data. The simplest form of the OpenWrite method accepts a single argument, the URI of the resource that will accept the uploaded file:

```
WebClient.OpenWrite(URI)
```

A second overloaded form of the method allows you to specify the method to be used for sending the data to the server; its value can be the "POST" or "GET" string:

```
WebClient.OpenWrite(URI, method)
```

The following statements create a Stream object for a specific document on a web server:

```
Dim wclient As New WebClient()  
Dim outputStream As Stream = wclient.OpenWrite(uri, "POST")
```

To send a short string to the web server, use a statement like the following:

```
outputStream.WriteLine("Data from a client")  
outputStream.WriteLine("End of data")  
outputStream.Close
```

You must not forget to close the stream, because this is how the server knows that the client is done uploading data.

UploadData method The UploadData method sends an array of bytes to the web server and returns another array of bytes with the server's response (if any).

```
WebClient.UploadData (URI, data)  
WebClient.UploadData (URI, method, data)
```

[Team Fly](#)

 Previous

Next 

```
End Class

Public Class Book
    Private _Title As String
    Private _Description As String

    Public Property Title() As String
        Get
            Return _Title
        End Get
        Set(ByVal Value As String)
            _Title = Value
        End Set
    End Property

    Public Property Description() As String
        Get
            Return _Description
        End Get
        Set(ByVal Value As String)
            _Description = Value
        End Set
    End Property
End Class
```

End Class

Array Search and Sort Methods

Happily, .NET arrays have search and sort methods. This example illustrates both methods. The simplest syntax for sorting is:

```
Array.Sort(myArray)
```

And for searching:

```
anIndex = Array.BinarySearch(myArray, "Penni Goetz")
```

Try this next example (Listing 2.4) to see results in TextBox1.

LISTING 2.4: SEARCHING AND SORTING AN ARRAY

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

    Dim myarray(4) As String
```

The first argument is the URI of the resource that will accept the data. The second argument of the send overloaded form is the name of the method to be used for uploading the data ("GET" or "POST"). The last argument of both forms of the method is an array of bytes with the information to be uploaded.

UploadFile method The UploadFile method is a convenient method for uploading a file to the web server. Its syntax is:

```
WebClient.UploadFile(URI, path)
```

where the first argument is the URI of the resource that will receive the file and the second argument is the path of the local file that will be uploaded. Like the other two methods for uploading data, the UploadFile method returns an array of bytes with the server's response (if any). An overloaded form of the method accepts another argument that specifies the method that will be used to transmit the file to the server, and it's the following:

```
WebClient(URI, method, path)
```

UploadValues method This method uploads a collection of name/value pairs to the web server and returns an array of bytes with the server's response (if any). To call this method you must first create a NameValueCollection collection with the name and value pairs you want to upload and pass it along with the URI of the resource that will receive the data on the server. This is similar to submitting a form with several fields to the server (each field's name is the corresponding control name and its value is the data entered by the user). The syntax of the method is:

```
WebClient.UploadValues(URI, data)
```

To upload a collection of three named values, use the following statements:

```
Dim data As New NameValueCollection()  
data.Add("First Name", Joe)  
data.Add("Last Name", "Doe")  
data.Add("EMail", "user@domain.com")  
Dim wClient As New WebClient()  
WebClient.UploadValues("192.210.27.119", data)
```

The UploadValues method has no equivalent download method; it's used to submit a virtual form to the web server. You can find out the names of the controls on a form and then submit this form to the server from within your application using the UploadValues method. You can create numerous combinations of data from within your code and submit them quickly to the server, as opposed to filling out the form manually. Consider a site that prompts the user for a zip code and returns the temperature for the corresponding area. You can quickly retrieve the temperatures throughout the country, without entering each zip code on a form and submitting the form manually every time. Just call the same ASP application that the original web application calls using the UploadValues method to pass a different zip code every time.

All three methods for uploading data return an array of bytes with the server's response. This response can be used as a confirmation of the successful completion of the operation. The methods are synchronous, which means that your application will appear to be busy while it's downloading or uploading data. As you have noticed, the data are transmitted as arrays of bytes (unless you're

[Team Fly](#)

 Previous

Next 

uploading a local file, or downloading directly to a local file, of course). To convert strings to byte arrays (or byte arrays into strings), you must use the methods of the `System.Text.Encoding.ASCII` class. The `System.Text.Encoding` class contains a number of useful classes, such as the `Unicode`, `UTF7`, and `UTF8` classes, which handle `Unicode`, `UTF7`, and `UTF8` characters, respectively. All of these classes expose the same methods, so we'll present the methods of the `System.Text.Encoding.ASCII` class.

To convert a string to an array of bytes and vice versa, use the following two methods:

GetBytes method Call the `GetBytes` method to convert a string to an array of bytes. It accepts as argument a string and returns an array of bytes. The following statements convert a string variable (the `cmdString` variable) to a series of bytes and store them in the buffer array:

```
Dim cmdString As String = "Uploaded Filename"  
Dim buffer() As Byte  
System.Text.Encoding.ASCII.GetBytes(cmdString)
```

GetString method Likewise, the `GetString` method converts an array of bytes to a string. This method accepts as arguments an array of bytes and returns a string:

```
System.Text.Encoding.ASCII.GetString(buffer)
```

We're showing here the simplest form of both methods. The `GetBytes` and `GetString` methods are overloaded, and you can look up the other forms of the methods in the documentation.

Downloading Documents with WebClient

To download a document from a web server, create an instance of the `WebClient` class and call its `Download` method, passing as argument the URI of the resource you want to download. The resource can be a filename or the name of an executable (an ASP application) that generates data on the fly. The following statements request the main page of a website and display it on a `TextBox` control. What you will see on the control is the page's HTML code. The file need not be an HTML page; you can specify any document's URI with the argument of the `DownloadData` method. The document's contents will be returned in an array of bytes, as shown in the following code segment:

```
Imports System.Net  
Imports System.Text  
Dim WClient As New WebClient()  
Dim remoteUrl As String = "http://www.sybex.com"  
Dim buffer As Byte() = WClient.DownloadData(remoteUrl)  
TextBox1.Text = Encoding.ASCII.GetString(myDataBuffer)
```

The `DownloadData` method is synchronous: the application's interface will freeze until the entire document has been downloaded to the client. Of course, you can execute this method on a thread so that the interface will remain responsive. Even so, the data doesn't become available before the last byte has been downloaded to the client.

If you would rather download one line at a time and process the lines as they arrive, you can create a Stream object with the OpenRead method and use the Stream object's ReadLine method, as shown in the following code segment:

```
Dim WClient As New WebClient()
```

[Team Ely](#)

 Previous

Next 

```
Dim WStream As Stream = WClient.OpenRead('http://www.sybex.com")
Dim SR As New StreamReader(WStream)
Dim txtLine As String
txtLine = SR.ReadLine
While Not txtLine Is Nothing
    Application.DoEvents
    TextBox1.AppendText(txtLine & vbCrLf)
    txtLine = SR.ReadLine
End While
WStream.Close
```

Finally, you can download the data directly to a file with the `DownloadFile` method. This method accepts two arguments, the URI of the document to be downloaded and the path of the file where the data will be stored. The following statements will download the HTML code of the Sybex main page and store it to the `C:\SYBEX.HTML` file:

```
Dim fileName As String = "C:\SYBEX.HTML"
Dim WClient As New WebClient()
WClient.DownloadFile ("http://www.sybex.com", fileName)
```

Uploading Documents with WebClient

Uploading a file doesn't mean that it will also be saved somewhere on the server's file system. Quite the opposite. When you upload a file, you also need a Web application running on the server, which will intercept the file and process it. The address to which the document will be uploaded (the destination URI) is actually the URI of an application that can process the incoming data.

If the data you want to upload resides in a file, you can upload the entire file with the `UploadFile` method, which accepts three arguments: the destination URI, the method to be used for upload, and the path of the file to be uploaded. The method that will be used for uploading the data is a string argument and its value can be either "POST" or "GET." The POST method allows you to upload a larger amount of data, but both methods are limited when it comes to really long files.

To experiment with file uploading, you must create a client application that will upload some information to the server, as well as a Web application that will accept the data on the server. Start Visual Studio and create a new Web application, the `GetFile` project. Then enter the statements in Listing 12.13 in the WebForm's Load event handler:

LISTING 12.13: AN ASP.NET APPLICATION TO ACCEPT DATA UPLOADED BY A WEB CLIENT

```
Private Sub Page_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    Dim str As IO.Stream
    str = Request.InputStream
    Dim strLen As Integer = CInt(str.Length)
    Dim buffer(strLen) As Byte
    str.Read(buffer, 0, strLen)
    Response.Write("You submitted " & buffer.Length.ToString & _
        " bytes of data")
End Sub
```


This Web application retrieves the data submitted by the client, stores them to an array of bytes, and returns a string with the length of the array. This is the server's response and we'll use it in our client application to verify that the operation has completed successfully.

Then switch to the WebForm's HTML tab and delete everything except for the header of the page:

```
<%@ Page Language='vb' AutoEventWireup="false" Codebehind="WebForm1.aspx.vb"
Inherits="GetFile.WebForm1"%>
```

Our application shouldn't have a visible interface, because we want to contact it remotely from within our client and upload the file. Now you can compile the project and contact it from a client. Create the appropriate EXE file with the Build GetFile command of the Build menu. Then start a new instance of Visual Studio, create a Windows application, and place the statements in Listing 12.14 behind a button's Click event handler:

LISTING 12.14: UPLOADING A FILE TO A WEB SERVER

```
Private Sub Button1_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
    Handles Button1.Click
    Dim URI As String = "http://127.0.0.1/GetFile/WebForm1.aspx"
    Dim myWebClient As New WebClient()
    Dim fileName As String = "c:\products.xml"
    Dim buffer As Byte() = myWebClient.UploadFile(URI, "POST", fileName)
    MsgBox(System.Text.Encoding.ASCII.GetString(buffer))
End Sub
```

If you run the Windows project and click the button on the form, the specified file will be submitted to the GetFile.aspx application and a few moments later you will see the server's response on a message box. The application running on the server doesn't process the data, but you can edit the code to perform any type of processing, or store the buffer array to a file on the server. It's a matter of writing simple VB code, as you would do for a Windows application.

You can also use this technique to pass serialized objects to the server. Let's consider a Windows application that accepts orders and stores them into instances of the Order class. If the application needs to send a copy of each order to a remote system, you can serialize the order objects into a MemoryStream and then submit this stream to an ASP application running on a web server. The ASP application should have access to a copy of the class from which the objects are derived and deserialize the stream into instances of this class. Of course, there's a better method to pass serialized objects to a server application, using a message queue (message queues are discussed in detail in Chapter 14). When this is not possible, or desirable, it's almost trivial to write a server and a client application that exchange serialized objects using the WebClient class.

The WebClient class is fairly straightforward to work with and should be used with simple applications that don't require extensive interaction between the client and the server. For more demanding applications, you can use the WebRequest and WebResponse classes, which are discussed next.

[Team Ely](#)

 Previous

Next 

The WebRequest and WebResponse Classes

Using the WebClient to exchange files with a web server is almost trivial, but the operation is synchronous: once you start reading the incoming stream (or sending the outgoing stream), your application will freeze until the entire document has been moved. The .NET Framework provides the WebRequest and WebResponse objects, which are more flexible and allow asynchronous transfers.

Behind the scenes, the WebClient class uses the WebRequest and WebResponse classes to connect to the remote server and retrieve its response. The WebClient class is used for very simple applications and textbook examples. Any real application that needs to access resources on the Internet should use these two classes and their descendants. One very good reason for using the WebRequest and WebResponse classes is that the WebClient class's methods are synchronous—you can't use them to download, or upload, a resource asynchronously. You can always use the WebClient class from within a separate thread, but as you will soon see, the WebRequest and WebResponse classes have built-in asynchronous capabilities.

The WebRequest and WebResponse classes are abstract classes and can't be used directly in your code. They abstract the operations of connecting to a server and requesting resources. The two descendant classes that are implemented in the .NET Framework are the HttpWebRequest/HttpWebResponse classes, which we use to access resources with the `http://` URI scheme, and the FileWebRequest/FileWebResponse classes, which we use to access resources with the `file://` URI scheme. In the section we'll discuss in detail the HttpWebRequest and HttpWebResponse classes.

To access a web page (or any document that can be returned by a web server) with the WebRequest class's members, create an instance of the WebRequest class by calling its Create method, whose syntax is:

```
HttpWebRequest.Create (URI)
```

where the URI argument is a string representing the document's URL. This method returns a WebRequest object; you can use it to set the properties of the connection, or call its GetResponse method to retrieve a WebResponse object. The WebResponse object represents the response of the server to the client. Its GetResponseStream returns a Stream object, which you can use to read the data sent by the server. The following statements establish a connection to a remote server and read the specified document (the main website page of this book's publisher):

```
Dim URI As Uri = New Uri ("http://www.sybex.com")
Dim wReq as HttpWebRequest = HttpWebRequest.Create (URI)
Dim wResp As HttpWebResponse = wReq.GetResponse ()
Dim InStream As Stream = wResp.GetResponseStream ()
Dim reader As StreamReader = New StreamReader (InStream, Encoding.ASCII)
Dim HTMLdoc As String = reader.ReadToEnd ()
Console.WriteLine (respHTML)
wResp.Close ()
```

Notice that the `HttpWebResponse` object created by the `GetResponse` method must be closed, or else you won't be able to make additional requests using this object.

To upload data to a server, call the `GetRequestStream` object and then use the `Stream` class's methods to write data onto the stream. The data will be uploaded to the remote server, where they

[Team Fly](#)

 Previous

Next 

must be processed by a Web application (or by a plain old ASP application). The following statements upload a local file to a remote web server:

```
Dim Data() As Byte
'statements to populate Byte array
Dim wReq As HttpWebRequest = _
    HttpWebRequest.Create('http://127.0.0.1/GetFile.aspx')
wReq.Method = "POST"
wReq.ContentLength = Data.Length
Dim OutStream As Stream = wReq.GetRequestStream()
OutStream.Write(Data, 0, Data.Length)
OutStream.Close()
```

The two classes are straightforward to use and we'll look at an example in a moment. To complete this introduction to the basic members of the two classes, we should discuss the methods that provide asynchronous support. They're the `BeginGetResponse` and `EndGetResponse` methods. The `BeginGetResponse` accepts two arguments: a callback delegate and an object containing information about the request. The syntax of the `BeginGetResponse` method is:

```
HttpWebResponse.BeginGetResponse(callback, state)
```

where `callback` is a delegate to the subroutine to be called when the `HttpWebResponse` object is available and `state` is an object variable (you can store in it information you want to pass to the delegate). The `BeginGetResponse` method returns an `IAAsyncResult` object, which is a reference to the asynchronous request.

In the callback delegate, you must call the `EndGetResponse` method explicitly. Once the `HttpWebResponse` object that represents the server's response becomes available, you can call its `GetResponseStream` method to retrieve the `Stream` object with the server's data. The delegate is the address of a subroutine that accepts a single argument, which is the object returned by the `BeginGetResponse` method. Reading the data from the remote server is also a potentially slow process (the server may take a while to respond, or the client's connection to the server may be slow). This operation can also be performed asynchronously with the `BeginGetRequestStream` and `EndGetRequestStream` objects.

THE WEBREQUEST PROJECT

To demonstrate the members of the `WebRequest` and `WebResponse` object, we've designed a simple application that downloads a document from a web server using these two classes (see Figure 12.4). As you know, downloading a file from a remote server is a slow process and we can't allow the interface of our application to stop responding while the client computer waits for the server to start submitting data. This example demonstrates how to contact a web server and request a document asynchronously.

The Asynchronous Request button calls the `HttpWebRequest.BeginGetResponse` method, which initiates the asynchronous download of the document specified with the `Create` method. This method accepts as argument a delegate, which is the address of the subroutine that must be invoked as soon as the server starts transmitting the document. The code behind the Asynchronous Request button is quite simple, as shown in Listing 12.15.

[Team Fly](#)

 Previous

Next 



FIGURE 12.4 The WebRequest demonstrates how to contact a web server asynchronously.

LISTING 12.15: DOWNLOADING A FILE ASYNCHRONOUSLY WITH HTTPWEBREQUEST

```
Private Sub AsynchronousDownload(ByVal sender As System.Object, _  
                                ByVal e As System.EventArgs) _  
                                Handles btnAsync.Click  
    Dim WReq As HttpWebRequest = HttpWebRequest.Create(txtURL.Text.Tr  
    Dim ReqCallback As New AsyncCallback(AddressOf RequestComplete)  
    WReq.BeginGetResponse(ReqCallback, WReq)  
End Sub
```

The RequestComplete() subroutine is an asynchronous callback and it accepts a single argument, which is of the IAsyncResult type (see Listing 12.16). This argument basically identifies the asynchronous operation that completed its execution. In the RequestComplete subroutine's code you must call the EndGetResponse method, passing the same argument that was passed to the RequestComplete subroutine by the system. The EndGetResponse method returns an HttpWebResponse object, just like the GetResponse method. Once the instance of the HttpWebResponse object has been created, you can use it to read the document, either in its entirety or one line at a time.

LISTING 12.16: THE REQUESTCOMPLETE DELEGATE

```
Private Sub RequestComplete(ByVal ar As System.IAsyncResult)  
    Dim webGet As HttpWebRequest = CType(ar.AsyncState, HttpWebRequest)  
    Dim wResp As HttpWebResponse  
    Dim RStream As IO.StreamReader  
    wResp = CType(webGet.EndGetResponse(ar), HttpWebResponse)  
    RStream = New IO.StreamReader(wResp.GetResponseStream())  
    Dim data As String = RStream.ReadLine  
    While Not data Is Nothing
```


This page intentionally left blank.

```
myarray.SetValue("one", 1)
myarray.SetValue("two", 2)
myarray.SetValue("three", 3)
myarray.SetValue("four", 4)

Dim cr As String = vbCrLf 'carriage return
Dim show As String

Dim i As Integer

For i = 1 To 4
    show = show & myarray(i) & cr
Next

TextBox1.Text = show & cr & cr & "SORTED:" & cr

Array.Sort(myarray)

show = ""
For i = 1 To 4

    show = show & myarray(i) & cr

Next

Dim anIndex As Integer

anIndex = Array.BinarySearch(myarray, "two")
Dim r As String = CStr(anIndex)

show += cr & "The word two was found at index number " & _
r & within the array"

TextBox1.Text += show

show = ""

For i = 1 To 4

    show = show & myarray(i) & cr

Next

TextBox1.Select(0, 0) 'turn off selection

End Sub
```

LISTING 13.1: BUILDING AND TESTING A WEB SERVICE

```
<WebMethod()> Public Function ReverseWords(ByVal s As String) As String

    Dim delimStr As String = ' '
    Dim delimiter As Char() = delimStr.ToCharArray()
    Dim split As String() = s.Split(delimiter)
    split = s.Split(delimiter)

    split.Reverse(split) 'reverse the array

    s = " "

    For i As Integer = 0 To split.Length - 1
        s &= split(i) & " "
    Next

    Return s

End Function
```

Also, it's nice to include some metadata, describing the service, so add a description attribute (shown here in boldface):

```
<System.Web.Services.WebService
  (Description:= "Reverses the words in a submitted string" , _
  Namespace:= "http://tempuri.org/services/Service1")> _
Public Class Service1
```

Press F5. Internet Explorer (or your default browser) starts. You see the description attribute displayed first when you attempt to consume this service, as shown in Figure 13.1:

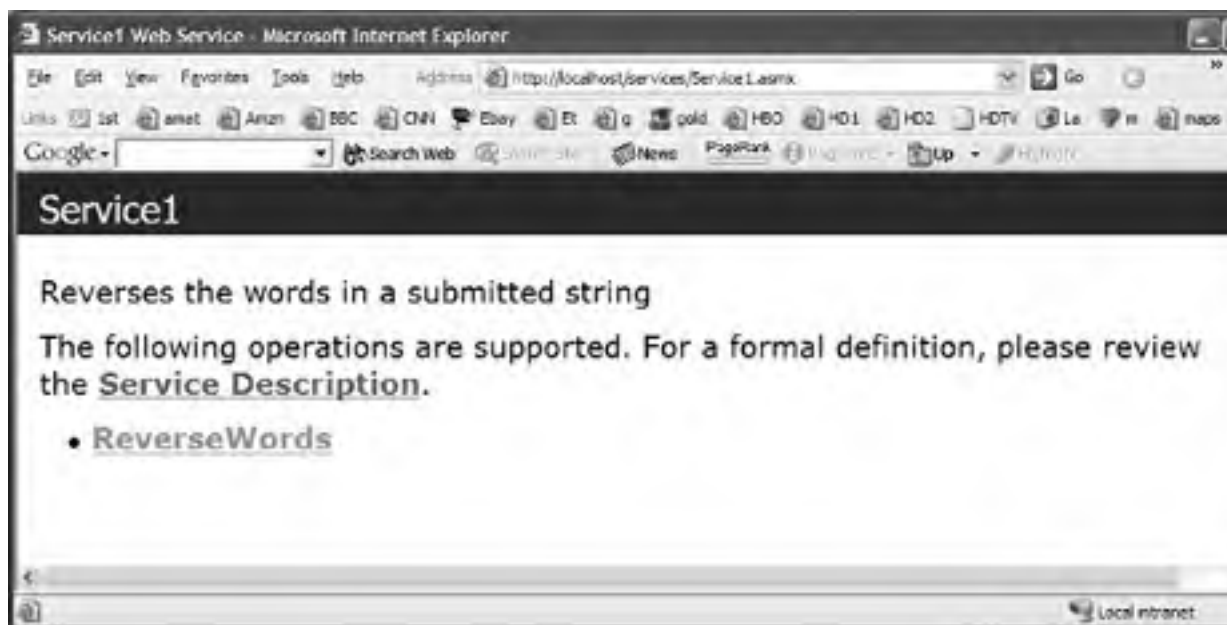


FIGURE 13.1 This display is the first step when testing a Web service.

[Team Fly](#)

 Previous

Next 

Click the Service Description link to see all the metadata and the structure of a SOAP Web service message. XML files that support the messaging between client and Web service are, fortunately, handled automatically for you. ASP.NET deserializes the incoming SOAP message into a usable object that your Web service can react to. Likewise, after your Web service's response message object is composed, it is serialized into a SOAP message, then sent back to the consumer of your service.

Now click the Back button to return to your service's main test page, and click the ReverseWords link in Internet Explorer, type in some words and submit the words (click Invoke). You see the returned string, reversed and in boldface, as shown in Figure 13.2:



FIGURE 13.2 Your Web service consumed, by you.

The first line of a Web service begins with `<WebService (Namespace := 'http://tempuri.org/')>`, an element that specifies that this class is special; it's a Web service. The arrow `<>` symbols have been used in computer programming to delimit HTML elements. (You can actually leave out the `WebService` element; ASP.NET looks at the filename extension `.ASMX`, and knows this is a Web service. Nonetheless, as you saw in the previous example, you can make use of the attributes of this element, such as the `Namespace` and the `Description` attributes.

```
<WebService (Description := "Provides the current date and time" ,  
Namespace := "http://tempuri.org/")> Public Class Service1
```

The `<WebMethod()>` element isn't optional. It specifies that a function belongs to a Web service. A `Description` attribute can be added to the `WebMethod` element. As you see, a Web service's source code blends HTML qualities such as attributes and elements along with typical VB.NET source code.

Caching Web Service Data

Computing often differs only in its terminology from traditional information handling. Consider the school nurse who keeps a stack of flu information sheets right on her desk during flu season, rather than having to waste time going to the file cabinet for a sheet each time a student comes in hacking and wheezing.

She calls it keeping something close at hand; we call it *caching*.

Clearly, if you run a quote-of-the-day service, you don't keep today's quote in a remote database and fetch it each time a client asks for it. Or if your Web service provides this week's sale items list, you don't regenerate that list for each request. Instead, in these and many similar scenarios, you want to gather the data only once a day (or week or whatever), then hold the data in quick, local memory rather than searching for and extracting it for each response.

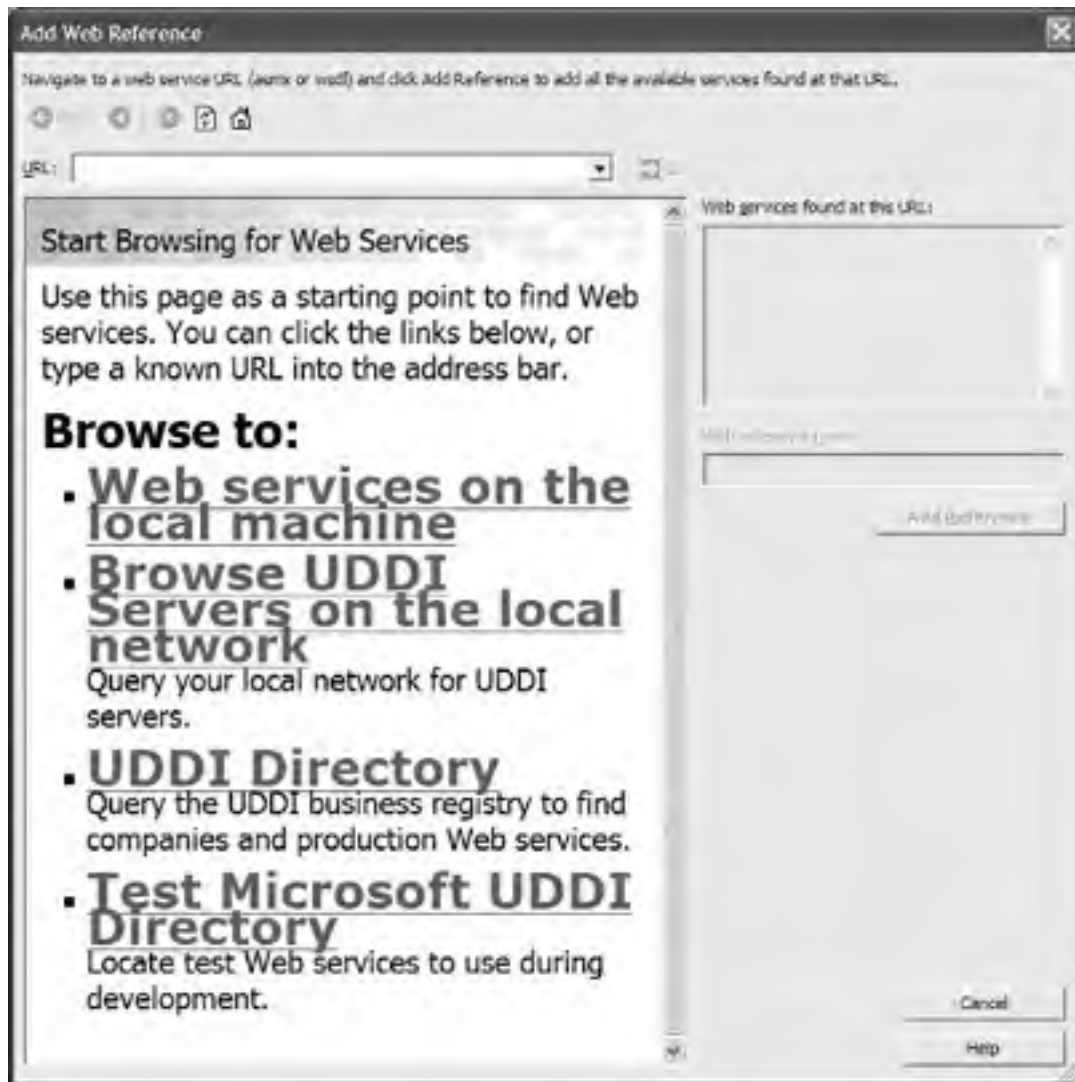


FIGURE 13.3 Use this "browser" to discover Web services.

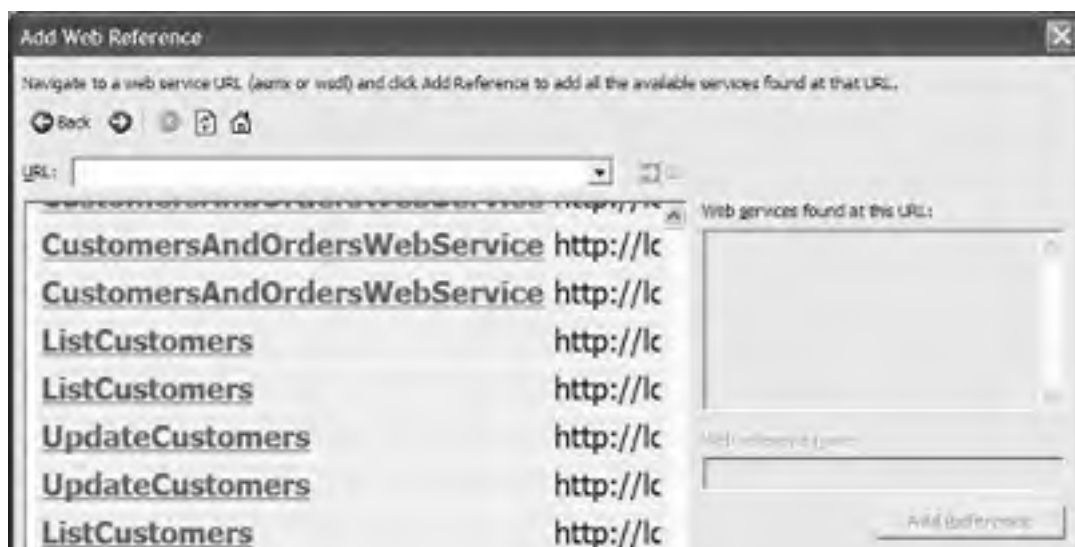




FIGURE 13.4 Here's a list of Web services on a local computer.

you don't see the Pubs database listed under SQL servers in VB.NET's Server Explorer, you need to install the samples that come with VB.NET. To do that, perform the instructions in the following section.

Using the Pubs Sample Database

Here's how to get a connection to the Pubs sample database so you can experiment with the examples in this and other chapters in this book.

Choose Start ➤ Programs ➤ Microsoft .NET Framework SDK ➤ Samples and QuickStart Tutorials. Follow the steps to install the .NET Framework Samples Database. If the installation fails, double-click the SQL Server icon on your tray at the bottom of the Windows desktop, then click the red button to stop SQL Server from running. Rerun InstMSDE.exe. Then reboot your computer to restart SQL Server.

After installing the sample database, you have to create a connection to it. Expand the Data Connections node in Server Explorer and see if there's a connection to Pubs. If not, you can create the connection by right-clicking Data Connections in Server Explorer and selecting Add Connection from the context menu. Follow the steps in the Data Link Properties dialog box. Click the Provider tab and choose Microsoft OLE DB Provider For SQL Server in the list box. Click Next and open the list under Select Or Enter A Server Name. Choose your SQL Server's name. Click the radio button next to Use Windows NT Integrated Security. Drop the list under Select The Database On The Server. Choose Pubs. Then close the dialog. With this data connection, you can experiment with various ways to bind controls to databases, to test database access within Web or Windows applications, and to access databases programmatically.

Getting an XML Dataset

Now you can try connecting your Web service to a database and extracting a dataset from it. Start a new Web service project by choosing File ➤ New ➤ Project, then double-clicking the ASP.NET Web Service icon in the New Project dialog box.

You need these Imports statements, so type them into the code window:

```
Imports System.Web.Services
Imports System.Data.SqlClient
```

In this example, you get information from the database and send it back to the client, and by happy "coincidence," the dataset is already packaged in XML, ready for platform-independent consumption.

Listing 13.4 is the code to type in to create your service (you add what's shown in boldface).

LISTING 13.4: CONNECTING TO A DATABASE

```
<WebService (Description:="Provides a list of jobs from the Pubs data
```

```
Namespace:="http://tempuri.org/">> Public Class Service1
```

```
Inherits System.Web.Services.WebService
```

[Team Ely](#)

 Previous

Next 

Web Services Designer Generated Code

```
<WebMethod()> Public Function ShowJobs() As DataSet

    Dim connPubs As New SqlConnection('server=localhost;Initial
Integrated Security=SSPI")
    Dim Datacmd As New SqlDataAdapter("select * from Jobs", connPubs)
    Dim ds As New DataSet()
    Datacmd.Fill(ds, "Jobs")

    Return ds

End Function

End Class
```

Press F5, click the link to your Web service in Internet Explorer, then click the Invoke button to see the results, as shown in Figure 13.5.

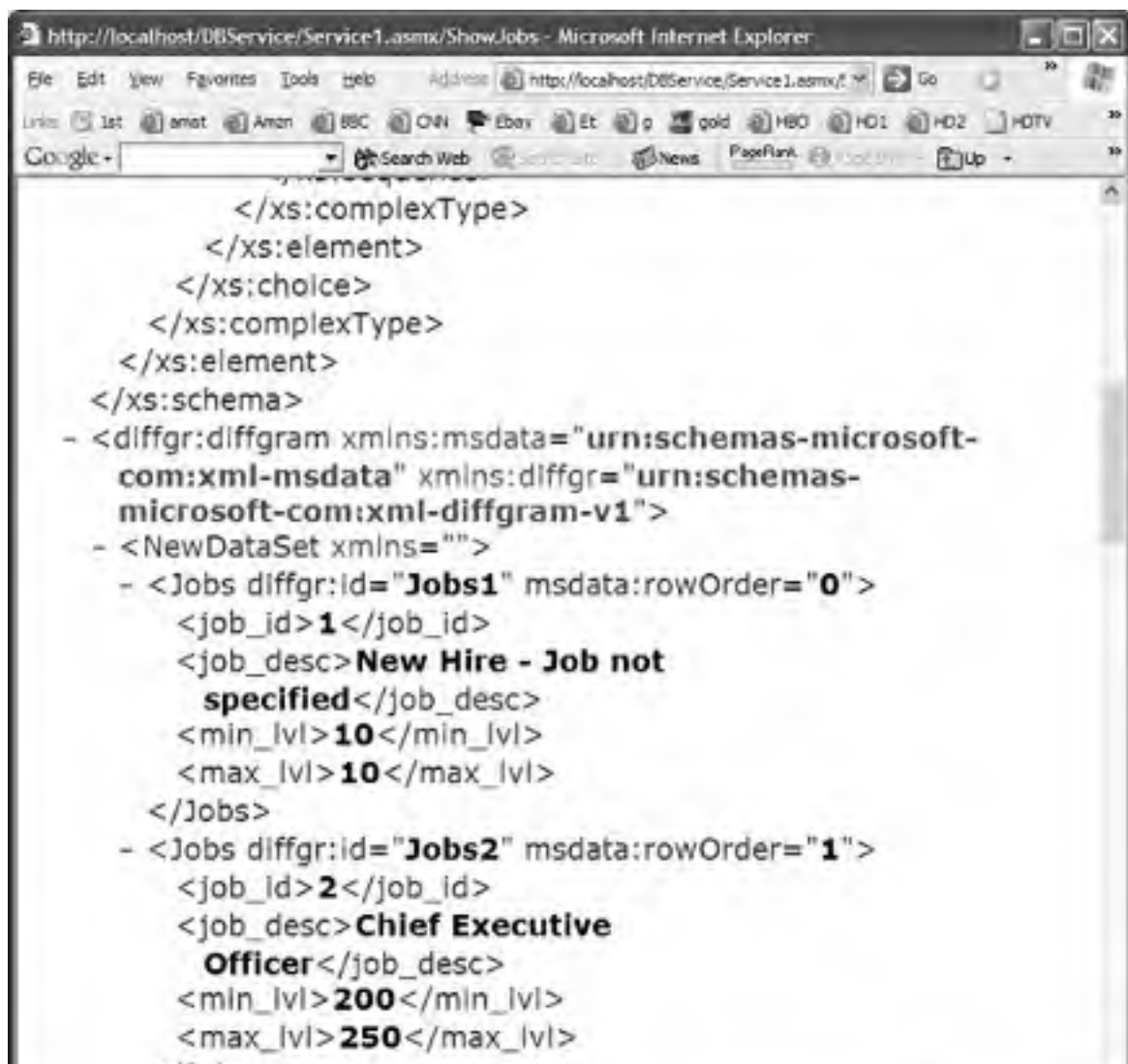




FIGURE 13.5 This is an ADO .NET dataset in XML format, with nested elements representing the fields in each record.

Potential Problems with MSDE

It's easy to get messed up when working with MSDE (Microsoft SQL Server 2000 Desktop Engine, the decapitated version of SQL Server that ships with .NET). It's decapitated because it doesn't include the usual SQL Server tools, such as Enterprise Manager (which can be necessary if you need to adjust permissions and other security features for MSDE). Many programmers have gotten error messages when trying to use code, like the previous example, that attempts to use SQL Server (MSDE version). The usual error messages are "permission denied" or "login failed." This can occur for various reasons. If you have this problem, right-click your Inetpub directory in Windows Explorer and select the Security (or Sharing And Security) option. Check to ensure that you haven't set up prohibitive security barriers for these directories. Also use Server Explorer in VB.NET to see if the Pubs database (or whatever you're trying to connect to) actually exists within the data connections you've defined. You may have to create a new data connection, reinstall the sample databases, or replace this reference to localhost:

```
Dim connPubs As New SqlConnection("server=localhost;Initial Catalog=pub
Integrated Security=SSPI")
```

With a the name of your computer\data connection, such as:

```
Dim connPubs As New SqlConnection("server=DELL\NetSDK;Initial Catalog=pubs; _
Integrated Security=SSPI")
```

As a last resort, you can sometimes solve MSDE problems by downloading and installing the latest version of MSDE, or you can install the trial version of SQL Server itself at <http://www.microsoft.com/sql/evaluation/trial/2000/default.asp>.

Looking at the Results

Take a look at the XML message returned in this example. The data is extracted from the Pubs database, then transformed into XML format (elements are created to hold each record, with other elements delimiting each field). Notice that the XML is divided into two main sections. First, the *schema* (structure) describes each field and some details about it, such as its data type:

```
<xs:element name="Jobs" >
<xs:complexType>
<xs:sequence>
  <xs:element name="job_id" type="xs:short" minOccurs="0" />
  <xs:element name="job_desc" type="xs:string" minOccurs="0" />
  <xs:element name="min_lvl" type="xs:unsignedByte" minOccurs="0" />
  <xs:element name="max_lvl" type="xs:unsignedByte" minOccurs="0" />
</xs:sequence>
</xs:complexType>
</xs:element>
```

After this you find the dataset itself, containing the records and their data. Aren't you happy that ASP.NET generates HTML, XML, SOAP, and other such files for you? Working with WebForms and Web services would indeed be dreary if we had to generate these verbose structures ourselves:

```
<Jobs diffgr:id="Jobs2" msdata::rowOrder= "1">
```

[Team Fly](#)

 Previous

Next 

The SetValue method in this example allows you to add or replace an item anywhere within an array by specifying the index number (however, the index number will be off by 1, thanks to the zero-base problem described earlier in this chapter):

```
myarray.SetValue("one" , 2)
```

The Sort method is overloaded with eight variations, including one that sorts only a subset of the array.

```
Array.Sort(myarray, StartIndex, LengthOfSubset)
```

Customized Sorting

Here's a useful technique. You can sort one array based on the elements in *another* array. This lets you devise your own sorting rules, a technique that comes in handy more often than you might think.

Here's an example. You have a single-dimension array containing first names and last names: Mary Jones, Bob Smith, and so on. You can't sort this array by the last names, by a simple sort command. They would be sorted by first name instead.

The solution is to create a second array that holds only the last names, sort it and at the same time sort the original array in tandem. Listing 2.5 is an example.

LISTING 2.5: SETTING YOUR OWN CUSTOM SORTING RULES

```
Private Sub Form1_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles MyBase.Load  
    Dim cr As String = vbCrLf 'carriage return  
  
    Dim myarray(5) As String  
    Dim lastnames(5) As String  
  
    myarray(0) = "Monica Lewis"  
    myarray(1) = "Georgio Apples"  
    myarray(2) = "Sandy Shores"  
    myarray(3) = "Dee Lighted"  
    myarray(4) = "Andy Cane"  
    myarray(5) = "Darva Slots"  
  
    TextBox1.Clear()  
  
    'create an array of the last names:  
  
    Dim i As Integer  
    Dim x As Integer  
    Dim s As String  
  
    For i = 0 To UBound(myarray)
```


Documentation Can be included wherever you want inside the WSDL. It's like a comment in an ordinary computer language, explaining to humans what's going on.

Definitions The root element of any WSDL. Ordinarily this element includes a declaration of the namespaces used in the WSDL area.

Types An XML schema that defines the types used by the messages defined in the document.

portType Describes a collection of *operation* elements, which themselves specify the input/output message defining the operation.

Binding Specifies the protocol, bindings for HTTP, SOAP, GET, and POST, and MIME.

Service Lets the client know the address where it can send messages to the Web service. Normally this maps a set of bindings for a single portType, specifying the URL of the Web service.

Viewing WSDL

To peruse an example of a WSDL contract, start a new VB.NET ASP.NET Web service–style project. Add a Description attribute to the line just beneath the Imports statement, like this:

```
<System.Web.Services.WebService(Description:= 'Adds or Multiplies two Integers  
Namespace:= "http://tempuri.org/WebService2/Service1")> _
```

Then replace the green commented lines with the two functions shown in Listing 13.5.

LISTING 13.5: EXAMINING WSDL CONTENT

```
<WebMethod(Description:="Please supply two Integers for addition")>  
Public Function AddThem(ByVal n As Integer, ByVal n1 As Integer) As  
  
    Return n + n1  
  
End Function  
  
<WebMethod(Description:="Please supply two Singles for Division")>  
Public Function DivideThem(ByVal n As Single, ByVal n1 As Single) As  
  
    Return n / n1  
  
End Function
```

Press F5 to run the Web service. After Internet Explorer displays the Web service, you can see its WSDL content by clicking the Service Description link. At the top you see a list of namespaces, and namespace abbreviation definitions that are employed throughout the entire document to specify the various schema. For example, here the abbreviation's in `xmlns:s="http://www.w3.org/2001/XMLSchema"` is identified. Then, later, in the `<types>` block, the abbreviation is used to specify the schema location: `<types> <s:schema elementFormDefault="qualified".`

The `targetNamespace` specifies a container namespace (a more general one) that all components of the element (`<definitions>` in this example) belong to. In other words, all names declared in this document belong, in this example, to your Web service at "tempuri" (your computer's internal site). This is, in effect, the URL of the Web service.

The types include `Types`, `Messages`, and `PortType` sections within the WSDL section, each describing various data content of the Web service message. The actual `<Types>` element specifies the variable types used in the Web service (parameters passed to functions, and the type passed back in the response). The permitted data types and details about them are described by XML schemas referred to via the URIs, or, optionally, they can be described other ways.

In our example, the `AddThem` method takes two integers as arguments, and returns an integer in what WSDL calls the `AddThemResponse` and `AddThemResult`.

Notice in the WSDL code that each variable type is qualified by the abbreviation `s:` ("`type='s:int'`"). The `s:` was defined previously in the WSDL as the abbreviation for the URI of the schema where the data types are defined. However, if you get curious and try to go there with your browser, you'll find that the "open source" is *closed*. You have to provide passwords and logon names to get in—presumably those in the know decided that this XML data type information required security. Some speculate that the schemas are still under "review" by committee members and thus are not for public consumption at this time. Others say that some academic interests are competing with each other. In any case, you may be refused entry. If you're not a member of the W3C, you might be able to join and get a password!

Following the `Types` section comes the `Messages` area, which is more abstract a description of your methods than in the `Types` area (no data types are defined here):

```
<message name="AddThemSoapIn">
  <part name="parameters" element=="s0:AddThem" />
</message>
<message name="AddThemSoapOut">
<part name="parameters" element="s0:AddThemResponse" />
</message>
```

Each `<part>` element here represents a parameter coming in to the Web service, or a value sent back as a response. The data type of each `<part>`, confusingly named the "element" attribute, either can be a type named in the `Type` area of the document or can be found in a schema (an XSD base type, a SOAP-defined type, or a WSDL-defined type).

As you would expect, the "specifications" for XML and daughter "languages" like WSDL and SOAP are in constant flux, and suffer from debate and bifurcation (what open-source people like to call *forking*). Just as there are thousands of specific, mutually exclusive, implementations of XML currently competing, at least four different data type schemas currently hang around as the "official" one. A new data type list floats up on average about once a year. In some cases, the schema version is identified by its year in the URI, such as this version from 2001:

`xmlns:s= "http://www.w3.org/2001/XMLSchema"`

If you're deeply interested in seeing a complicated set of suggested data types, take a look at the specifications (or *recommendations*, as some XML committee members prefer you say) at these locations:

<http://www.w3.org/TR/xmlschema-2/>

<http://www.w3.org/TR/2001/PR-xmlschema-2-20010330>

SOAP Too

In addition to the WSDL, you get this SOAP version of the same Web service, thanks to .NET. Listing 13.6 gives the SOAP request and response for your DivideThem method.

LISTING 13.6: SEEING THE SOAP

```
POST /xx/Service1.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: 'http://tempuri.org/xx/Service1/DivideThem'

<?xml version="1.0" encoding==" utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <DivideThem xmlns="http://tempuri.org/xx/Service1">
      <n>float</n>
      <n1>float</n1>
    </DivideThem>
  </soap:Body>
</soap:Envelope>
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding=="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001
/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">[
  <soap:Body>
    <DivideThemResponse xmlns="http://tempuri.org/xx/Service1">
      <DivideThemResult>float</DivideThemResult>
    </DivideThemResponse>
  </soap:Body>
</soap:Envelope>
```

Complex Types

In addition to the simple variables you've seen so far, you can also use enums, structures, and other complex types. Listing 13.7 is an example.

LISTING 13.7: EMPLOYING COMPLEX TYPES

```
Public Structure ThisType
    Public ID As Integer
    Public LastName As String
    Public Money As Single
    Public PersonalityFlaws As String
End Structure

<WebMethod()> Public Function Test(ByVal s As String) As ThisType
    Dim TT As New ThisType

    TT.ID = 12422
    TT.LastName = "Floriasta"
    TT.Money = "12.22"
    TT.PersonalityFlaws = "cheap as a ten-cent cigar " & s

    Return TT

End Function
```

Here's part of the resulting SOAP that .NET generates to describe the structure:

```
<CountCharsResult>
  <ID>int</ID>
  <LastName>string</LastName>
  <Money>float</Money>
  <PersonalityFlaws>string</PersonalityFlaws>
</CountCharsResult>
```

As you see, our VB.NET data type Single has been mapped to the SOAP Float type. Although XML and daughters are said to be "language independent," what this often means in practice is that the preferred language dominates. What's the preferred language by people in academia and therefore on XML committees? C, C++, and daughter languages. Float is a C data type.

Listing 13.8 shows how an Enum is translated into SOAP and WSDL.

LISTING 13.8: USING ENUMS

```
Public Enum DaysOfTheWeek
    Sunday
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
End Enum
```

```
<WebMethod()> Public Function Test() As DaysOfTheWeek

    Dim e As New DaysOfTheWeek
    Return e

End Function
```

This translates into the following SOAP:

```
<soap:Body>
  <TestResponse xmlns= 'http://tempuri.org/xx/Service1'>
    <TestResult>Sunday or Monday or Tuesday or Wednesday
      or Thursday or Friday or Saturday</TestResult>
  </TestResponse>
</soap:Body>
```

and the following WSDL, in its Type section:

```
<s:simpleType name="DaysOfTheWeek">
  <s:restriction base="s:string">
    <s:enumeration value="Sunday" />
    <s:enumeration value="Monday" />
    <s:enumeration value="Tuesday" />
    <s:enumeration value="Wednesday" />
    <s:enumeration value="Thursday" />
    <s:enumeration value="Friday" />
    <s:enumeration value="Saturday" />
  </s:restriction>
</s:simpleType>
```

PortType

The PortType section of a WSDL document is yet another abstraction of the Web service being contracted.

```
<portType name="Service1Soap">
  <operation name="Test">
    <input message="s0:TestSoapIn" />
    <output message="s0:TestSoapOut" />
  </operation>
</portType>
```

PortType describes incoming and outgoing parameters, and additional optional documentation, function names, or other data the author of the Web service might want to include. Think of this section as specifying the interfaces that are exposed. This can encompass more than a single message exchange. Also note that changes you make to the message element can cause changes in both the PortType and binding elements as well. Use as many portType elements as you want. This is a way to describe overloaded functions, for example.

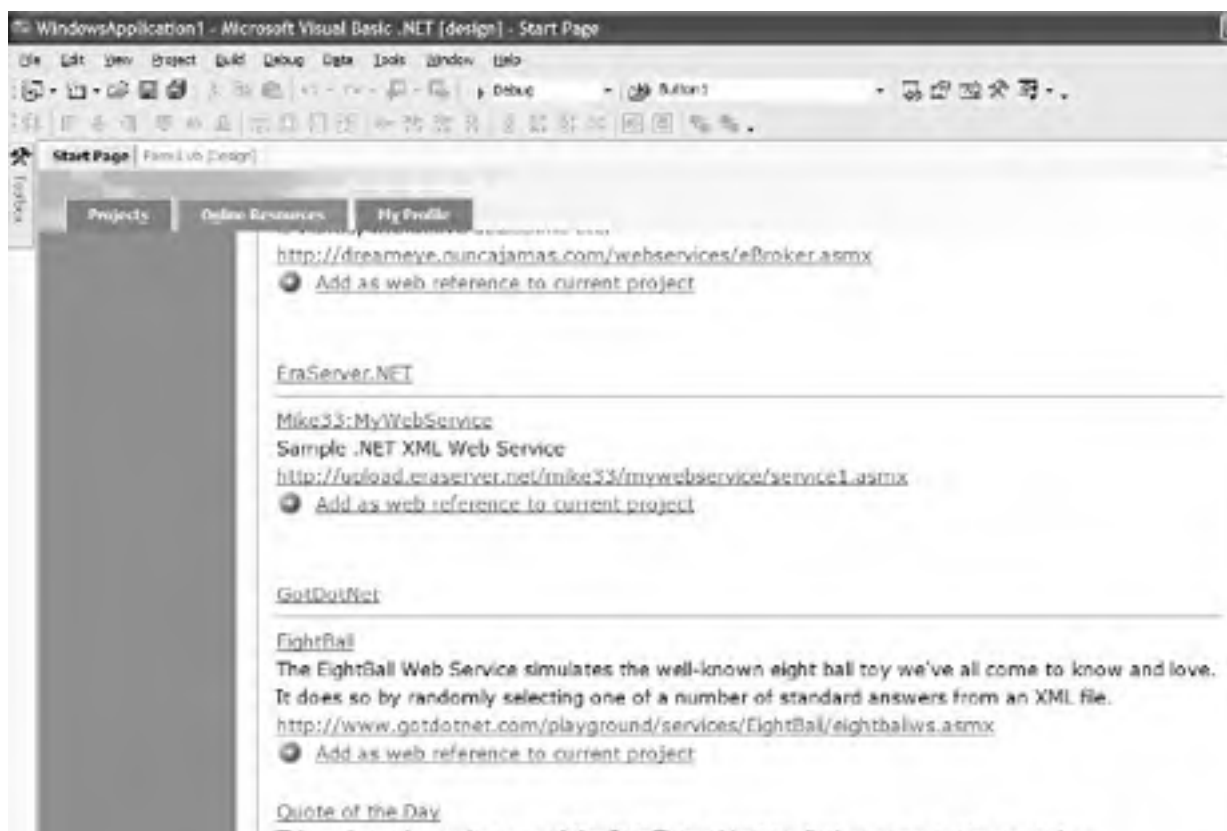
Here's one published service that worked. Start a new VB.NET Windows-style project and choose Help ➤ Show Start Page. Click the Online Resources tab, then click XML Web Services in the left pane. Leave the UDDI Production Environment radio button selected, then drop the list box under Category and choose Miscellaneous. Click the Go button. A list of available Web services appears. Locate the *Quote of the Day* service, shown in Figure 13.6.

Click the Add As Web Reference To Current Project link. The Solution Explorer now contains a Web Reference to this URL: `com.gotdotnet.www`. You use that address to create your Imports statement. Go to the code window now and type this Imports statement:

```
Imports WindowsApplication1.com.gotdotnet.www.Quote
```

If your project name isn't `WindowsApplication1`, replace that with your project's name. Here's the code you use to access this Web service:

```
Private Sub Form1_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles MyBase.Load  
  
    Dim cService As New com.gotdotnet.www.Quote  
  
    Dim s As String  
  
    s = cService.GetQuote()  
  
    MsgBox(s)  
  
End Sub
```



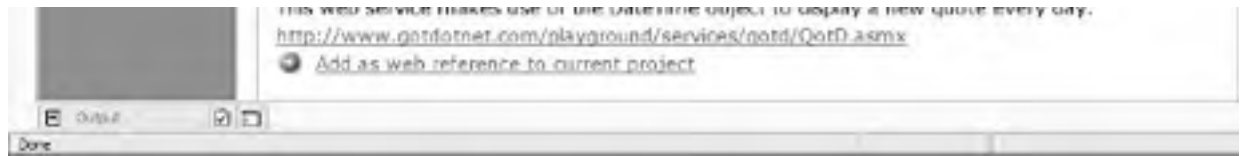


FIGURE 13.6 This UDDI Registry contains published services you can experiment with.

[Team Fly](#)

◀ Previous

Next ▶

```
s = myarray(i)
x = s.IndexOf(' ') find blank space
lastnames(i) = myarray(i).Substring(x) 'get last name
TextBox1.Text += lastnames(i) & cr
Next

TextBox1.Text += cr & "Sorted by last name:" & cr

myarray.Sort(lastnames, myarray)

For i = 0 To UBound(myarray)
    TextBox1.Text += myarray(i) & cr
Next

TextBox1.Select(0, 0) 'turn off selection
End Sub
```

Many Properties and Methods

In VB.NET, arrays have many members. There are several members unique to the array class (reverse, `GetUpperBound`, and so on). The simplest syntax for the reverse method reverses all the items in an array, like this:

```
Array.Reverse(myarray)
```

Or to reverse only a subset of items within the array, here the reversing begins with the item at index number 1 and reverses three items:

```
Array.Reverse(myarray, 1, 3)
```

Here is a list of all the public properties of the array class: `IsFixedSize`, `IsReadOnly`, `IsSynchronized`, `Length`, `Rank` (number of dimensions), `SyncRoot`.

Here are the public methods: `BinarySearch`, `Clear`, `Clone`, `Copy`, `CopyTo`, `CreateInstance`, `Equals`, `GetEnumerator`, `GetHashCode`, `GetLength`, `GetLowerBound`, `GetType`, `GetUpperBound`, `GetValue`, `IndexOf`, `Initialize`, `LastIndexOf`, `Reverse`, `SetValue`, `Sort`, `ToString`.

`LastIndexOf` searches an array (or a portion of an array) for a value, and returns the index of the final (highest) occurrence of that value. Alas, the index is off by one. `GetLowerBound` is a rather odd method because every array in VB.NET must have a lower bound of zero. `GetLowerBound` is vestigial, left over from an original plan to permit us programmers to specify a lower bound other than zero. There seems to be a political fight going on somewhere—two camps continuing to dispute over the base issue. The .NET Framework *does* support adjustable lower bounds for arrays, but you aren't supposed to mess with them. You're not permitted to create instances of the array class, then adjust the lower bound. As the VB.NET help documentation says on this topic: "Users should use the array constructs provided by the language."

This page intentionally left blank.

reaches the destination queue. The message can be sent directly to the remote queue (if it's possible), sent to a node between the local machine and remote computer where the destination queue resides, or stored at the local machine and be delivered when the computer is connected to a network. In any case, sending a message does not block the local machine. The source and destination computers need not be on the same network, and it's possible that messages will be routed through intermediate computers. MSMQ guarantees the delivery of the message. If the message can't be delivered for any reason, MSMQ can generate automatic acknowledgment messages. Let's start with an overview of the tools for creating and administering queues and then we'll see how to send and receive messages.

Types of Queues

There are several types of queues you can install. At the highest level, queues can be classified as *private* and *public*. A private queue is available only on the machine on which it was installed, but can be accessed by applications running on a remote machine, as long as the applications know the queue's name. Public queues are replicated through the network and applications can access the public queue at their site. It doesn't make any difference on which machine a public queue resides, because they all contain the same messages. Any message written to a public queue is replicated to all other public queues with the same name by MSMQ without any programming effort on your behalf.

In addition to private and public queues, there are two more types of queues: system queues and outgoing queues. The system queues are maintained by the system; there are three types of system queues:

- ◆ Journal messages
- ◆ Dead-letter messages
- ◆ Transactional dead-letter messages

Journal messages store copies of the messages you send to a private or public queue. On the sending computer, only a single queue is required for all messages sent from that computer. On the receiving computer, a separate queue with journal messages is created for each individual queue, which keeps copies of the messages removed from that queue.

Dead-letter messages store copies of messages that couldn't be delivered successfully. Transactional messages that couldn't be delivered successfully are stored as transactional dead-letter messages. A message is not considered successfully delivered if it expires before its delivery (the expiration interval is specified by the application that sends the message). If a message expires before its delivery, its copy is stored in the queue with the dead-letter messages of the computer on which the message expired.

One last category of queues are the outgoing queues. When you send a message to a queue that's not reachable at the time, the message is stored in an outgoing queue. Outgoing queues are created automatically by MSMQ under the Outgoing Queues folder; there's one outgoing queue for each remote queue to which your application has attempted to send a message. The outgoing queues are named by the corresponding remote queue's FormatName property and

they store the messages sent to the remote queues. The messages in the outgoing queues will be delivered to the proper remote queue automatically as soon as the local computer is connected to the network. Notice that outgoing queues are not permanent; once the messages in an outgoing queue are delivered, the queue itself is

[Team Ely](#)

 Previous

Next 

removed. It will be created again when MSMQ needs to temporarily store some messages that can't be delivered to the remote queue.

Messages aren't written to disk by default. This indicates that MSMQ was designed on the premise that messages will be read and processed (and therefore removed from the queue) as soon as possible. Even if the computer is shut down properly, the contents of the queues will disappear. You can change the default behavior by setting the Recoverable property of a message before sending it to a queue. Normally, messages will be read off the queue as soon as they arrive. If it's crucial that no message is lost, you should set the Recoverable property of the messages you send to the queue to True. This is especially advisable for messages that will be relayed to a remote system. These messages are stored to the local machine until they can be delivered to their destination, or to another node on the route to the destination computer.

Creating New Queues

Before you can work with a queue you must create it. There are two methods to create new queues: You can create (and administer) queues with visual tools in Visual Studio's IDE, or with the MSMQ Microsoft Management Console (MMC). By the way, it's possible to create, as well as administer, queues programmatically from within your application, as you will see later in this chapter.

To follow along, create a new queue with the MMC snap-in. Open the Administrative Tools folder in the Control Panel and start the Computer Management snap-in. Expand the branch Services And Applications and then expand the Message Queuing branch under it. You will see four folders that correspond to the various types of queues you can create and use: Outgoing, Private, Public, and System queues. Under each folder you'll find a number of queues. Under each queue there are two items: Queue messages and Journal messages. These two items contain the messages sent to the specific queue and copies of them. As messages are being processed, they're also removed from their queues, except for journal messages, which remain in their corresponding queues. You will not see any messages in the outgoing and public queues unless you have configured the corresponding queues.

To create a new private queue, right-click the Private Queues folder and, from the context menu, select New ➤ Private Queue. You'll be prompted to enter the name of the new queue, which has the prefix "private\$," as you can see in Figure 14.1.



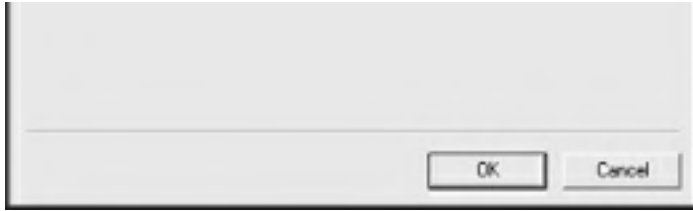


FIGURE 14.1 Creating a new queue with the Message Queuing MMC snap-in

The new queue has no messages and you can't add messages using visual tools. You can only delete a queue, or purge its messages (if it has any). There's no mechanism for deleting isolated messages. The Purge command in the context menu of the Queue Messages item will remove all the messages from the queue. Even though there are no messages in the new queue, you can see the names of the basic properties of each message; they're the headers of the columns in the right pane of the Computer Management window. Each message has a label (optional), a priority, and an ID. The ID is a GUID (globally unique identifier) and it's made up of a unique identifier for the queue and a sequence number for each message in the queue. In the following sections we'll discuss methods of referencing existing queues and creating new ones programmatically.

Administering Queues

The basic class for administering and using queues is `System.Messaging.MessageQueue`. This class contains members that allow us to explore existing queues and create new ones, as well as send and receive messages.

***NOTE** To access the functionality of the MSMQ component from within your VB applications, you must first add a reference to the System.Messaging component (use the Project ➤ Add Reference command in the IDE). Then import the System.Messaging namespace to your application so that you won't have to fully qualify each member of this class.*

To reference a queue, you can use the queue's path, label, or format name. The queue's pathname is a combination of the name of the computer on which the queue resides and the name of the queue. If the queue is public, use the following pathname form:

```
computer_name\Queue_name
```

If the queue is private, you must insert the string `Private$` between the computer name and the queue name:

```
Computer_name\private$\queue_name
```

If the queue is on the local machine, you can use the following pathname:

```
.\private$\queue_name
```

The queue's format name is a string that resembles a `ConnectionString` and contains connection details as well as the queue's pathname. The following is the format name of a queue on the PowerToolkit machine:

```
DIRECT=OS:PowerToolkit\private$\order_queue$
```

The last method is to use the queue's label. Each queue can have a label, which you assign to the queue by setting its `label` property from within your code, or through the MSMQ snap-in. If you specify a label that's not unique, a runtime exception will be thrown. Each queue has a unique identifier (a GUID value), which you can also use to reference the queue. This identifier is assigned to the queue when it's created by the system.

Journal queues must be accessed with a different notation. Journal queues that are specific to private or public queues are referenced by the name of the queue they refer to and the suffix `Journal$`. Let's say the orders queue receives messages about orders. You can associate a journal queue with the

orders queue and request that copies of all messages sent by the `orders` queue are stored in the journal queue associated with the orders queue. This queue is private and can be referenced with the following path:

```
ComputerName\Orders\Journal$
```

The server's journal queue can be accessed with the following pathname:

```
ComputerName\Journal
```

as there's only one journal queue for the server.

In this chapter you'll learn how to access and configure message queues from within your applications. We'll discuss the topic of programming against the MSMQ objects in detail in the following sections, but let's start with a quick overview of the basic VB code for sending and receiving messages. The simplest method of adding a reference to a queue to your application is through the Server Explorer window. Start Visual Studio, switch to Server Explorer, expand the Servers branch, and select the server on which the desired queue is installed. If the name of the server doesn't appear in the Servers branch, right-click the Servers item and select Add Server from the context menu. Under the selected server's name you will see the Message Queues item. Expand this item, select the desired queue, and drop it on the project's form. A new component will be added to the form, the `MessageQueue1` component. Select the new component, and in the Properties window you will see the queue's properties, as shown in Figure 14.2. The Path property is one of them; this property identifies the queue by its FormatName. We'll discuss the queue's properties in the following sections. You can refer to the selected queue in your code as `MessageQueue1`. You can also create new queues in the Visual Studio IDE by selecting the Create Queue command from the Private Queues or Public Queues item of the Server Explorer.

Let's experiment with the new queue. Add a button to the form of the current project and enter the following code in its Click event handler:

```
Private Sub Button1_Click(ByVal sender As System.Object, _  
                          ByVal e As System.EventArgs) _  
    Handles Button1.Click  
    MessageQueue1.Send("The message's body" , "Simple Message")  
    MsgBox("Message written to queue " & _  
          MessageQueue1.QueueName & " successfully")  
End Sub
```

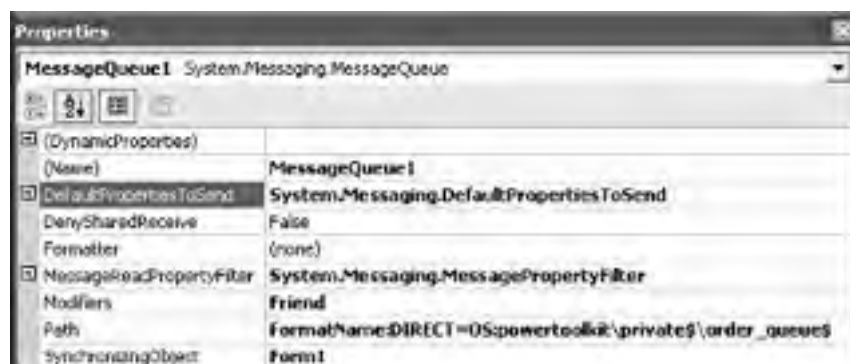




FIGURE 14.2 Setting a queue's properties in the IDE

that accepts a second argument, which is a Boolean value that determines whether the new queue will be transactional or not. We'll discuss transactional queues later in this chapter. The Delete method accepts as argument a queue's format or pathname and deletes the specified queue. The following statements demonstrate how to create the orders private queue, if it doesn't already exist:

```
Dim Msq As New MessageQueue
Dim queuePath As String = " .\private$\orders"
If Not Msq.Exists(queuePath) Then
    Try
        Msq.Create(queuePath)
    Catch
        MsgBox("Could not create queue")
    End Try
End If
```

Exploring a Computer's Queues

To explore the queues on the local computer (or any other computer on which you have the necessary rights to manipulate queues), you can use the following methods. All methods listed here return an array of MessageQueue objects and each element of the array references a specific queue:

GetPrivateQueuesByMachine This method accepts a string argument, which is a computer's name, and returns an array with references to each private queue on that machine. You can use the appropriate member of the array to reference a specific queue in your code.

GetPublicQueues This method returns all the public queues on the network. An overloaded form of this method accepts as argument a string, which you can use to specify the criteria for selecting the desired queues. This argument is a property of the MessageQueueCriteria class, and you can combine multiple criteria with the And operator. The criteria available for filtering queues are properties of the MessageQueueCriteria enumeration and they're shown in Table 14.1.

GetPublicQueuesByCategory This method returns all the public queues that belong to the category specified with its argument. To use this method you must first assign a category to your queue's Category property. The category has the structure of a GUID but it need not be a real GUID, nor is it unique.

GetPublicQueuesByLabel This method returns all the public queues that have the label specified with its argument.

GetPublicQueuesByMachine This method returns all the public queues that reside on the machine specified with its argument.

The GetPublicQueues method accepts an argument that filters the desired queues. The filtering expression is a property of the MessageQueueCriteria class; the properties of this class are shown in Table 14.1. You can combine multiple filters with the And operator.

Message Properties

The Message class exposes the following properties, which apply to individual messages. Some of them are trivial, while others are not as simple to use. We'll use many of these properties in our sample applications, later in the chapter. In the following paragraphs we list the most important properties.

PRIORITY

Messages are sent to the destination queue with a default priority. Messages with the same priority are stored in the queue in chronological order. You can set a message's priority with the Priority property of the Message class, so that messages with higher priority are stored in front of messages with lower priority, regardless of the order in which they arrive to the queue. The Priority property's value can be a member of the MessagePriority enumeration: `Highest`, `Very High`, `High`, `AboveNormal`, `Normal`, `Low`, `VeryLow`, and `Lowest`. The default priority is `Normal`, which corresponds to the numeric value 3 you see on the MSMQ snap-in.

LABEL

Messages can have a label, which is displayed in the Label column of the MSMQ snap-in. If you don't specify a label, an empty string is displayed in this column. The Label property characterizes either the message or the application that sent it and need not be unique.

ID

Each queue gets a unique value as soon as it's created, which is a GUID string. When a message arrives at the queue it gets its own unique ID, which is the queue's GUID followed by an Identity value (a value that behaves just like the Identity column of a SQL Server table; its value keeps increasing with each incoming message, even after purging the queue). We don't usually retrieve messages by their IDs, because we simply don't know the IDs of the messages in a queue. There's one exception: related messages carry the ID of the parent message and we can use a message's ID to retrieve the related messages. We can obtain the ID of the original message either when we create it or when we read it from the queue.

Acknowledgment messages are related to a specific message; we can use the MessageQueue object's `ReceiveByCorrelationID` method to retrieve the acknowledgment messages for a specific message. This message's ID is passed to the `ReceiveByCorrelationID` method as argument and the method returns one or more related messages (if they exist). Later in this chapter, you'll see how this method is used to retrieve acknowledgments for a specific message.

We can also retrieve messages by their IDs when we just peek at the messages in the queue and want to process one of them. Once a message has been processed, you must always retrieve it from the queue. If not, the message will be processed again.

TIP The ID of a message has the form {GUID}\number, and it looks something like this:

{051C9A20-9A39-4696-AE8B-04A9C1C87BD5}\6721

This is what you see in the MSMQ snap-in and this is what you get when you retrieve the ID property of a Message object. To retrieve a message by its ID, however, you must strip the curly brackets and pass to the appropriate method an ID like this:

051C9A20-9A39-4696-AE8B-04A9C1C87BD5\6721

[Team Fly](#)

 Previous

Next 

TIME TO REACH QUEUE, TIME TO BE RECEIVED

The `Time ToReachQueue` property is a `TimeSpan` object that represents the maximum amount of time allowed for the message to reach the destination queue. The `TimeToBeReceived` property is also a `TimeSpan` object that represents the maximum amount of time allowed for the message to be retrieved from the destination queue. Normally, the setting of the `TimeToBeReceived` property should be longer than the `Time ToReachQueue` property. If not, the `Time ToBeReceived` property takes precedence.

If the interval specified by the `Time ToBeReceived` property expires before an application had a chance to remove the message from the destination queue, MSMQ discards the message, but not without an indication. If the message's `UseDeadLetterQueue` property is `True`, the message is sent to the dead-letter queue. If you specify an interval for retrieving the message from the queue, you should also set the `UseDeadLetterQueue` property to `True`. Leave this property set to `False` for messages that should be processed within a certain interval and ignored after that.

When sending critical messages, you should set up an acknowledgment mechanism. You can request MSMQ to send a negative acknowledgment message back to the sending application if the message is not retrieved in the specified time interval by setting the message's `Acknowledge Type` property. You can also request a positive acknowledgment for the successful arrival of the message at the destination queue, or its removal from the destination queue. You'll learn how to request acknowledgments for your messages in the following section.

RECOVERABLE

The `Recoverable` property determines whether the delivery of a message is guaranteed. By setting the property to `True` (its default value is `False`), you cause the message to be stored locally at every step along its route to the destination computer. By default, messages are stored in memory, which means that MSMQ was designed on the premise that messages should be retrieved from their queues as soon as possible. Even messages sent to a private queue of the local computer are stored in memory. As a result, even when you shut down the computer normally, the messages that haven't been processed yet will be lost. It's a good practice to set the `Recoverable` property to `True` before sending a message.

MESSAGEQUEUE.DEFAULTPROPERTIES TO SEND

By default, the `MessageQueue` object's `Send` method doesn't send all the properties of a message. You can specify the properties to be sent along with the message by setting the appropriate properties of the `Message` object. Alternatively, you can use the `MessageQueue` class's `DefaultPropertiesToSend` property to set the properties to be sent for all messages to the queue represented by the `MessageQueue` object. You do so by setting the appropriate member of the `MessageQueue` object's `DefaultProperties ToSend` property. If you want all messages sent to the `Orders` queue to be acknowledged, you can set the `Acknowledge Type` and `AdministrationQueue` members of the `DefaultPropertiesToSend` property, as shown in the

following code segment (the `Q` and `ackQ` variables reference valid queues):

```
Q.DefaultPropertiesToSend.AcknowledgeType = AcknowledgeTypes.FullReceive  
Q.DefaultPropertiesToSend.AdministrationQueue = ackQ  
Q.DefaultPropertiesToSend.Priority = MessagePriority.Highest
```

[Team Ely](#)

 Previous

Next 

Creating and Sending Messages

To send a message to a queue, you must first obtain a reference to the desired queue by creating a new `MessageQueue` object. The name of the queue can be passed as argument to the `MessageQueue` class's constructor, as shown here:

```
Dim orderQ As MessageQueue
orderQ = New MessageQueue(".\private$\Orders")
```

The next step is to tell MSMQ about the objects we're going to send to the queue. We do so by creating a formatter, which determines how objects will be serialized before they're sent to the queue. You must obviously use the same formatter to properly deserialize the objects at the destination queue. It's possible to send messages of different types to the same queue, so you start by creating an array of `Type` objects, one element for each type you want to send to the queue. The `Orders` queue will receive objects of the `Order` type only, so the array will contain a single element:

```
Dim targetType() As Type
targetType(0) = GetType(Order1)
```

where `Order1` is an instance of the `Order` class. Finally, we pass the `targetTypes` array to the constructor of the `XmlMessageFormatter` class and we assign the new `XmlMessageFormatter` object to the `Formatter` property of the `MessageQueue` object that represents our queue:

```
msgQ.Formatter = New XmlMessageFormatter(targetTypes)
```

If the `targetTypes` array contains multiple types, the formatter will pick the proper type for each message sent to the queue. If you send to the queue a message of a type other than the one(s) specified in the `XmlMessageFormatter` object, a runtime exception will be raised.

You can also serialize messages into a binary format before sending them to the queue. In this case, you'll have to create an instance of the `BinaryMessageFormatter` class and use it in the place of the `XmlMessageFormatter` class. If you choose to use a binary serializer, don't forget to add the `<Serializable>` attribute to the definition of the class(es) that represent the object you'll send to (or read from) the queue. All classes can be serialized in XML format by default, but this isn't true for binary serialization. Binary serialized messages are more compact, but you can't read their bodies in the MSMQ snap-in. In this chapter we'll use the `XmlMessageFormatter`.

To send `Message` objects to a queue, use the `Send` method and pass as arguments a reference to the object you want to send to the queue and its type:

```
msgQ.Send(obj, objType)
```

If you have a class that represents orders, the `Order` class, you can send objects of the `Order` type to the queue with a statement such as:

```
msgQ.Send(order1, "Order")
```

The following class represents simple orders; we'll use instances of this class to store orders. The Order class's Details property is an array of OrderDetail objects, and each OrderDetail object has fields for each order detail item:

```
Public Class Order
    Public ID As Integer
```

[Team Fly](#)

 Previous

Next 

```
        Public orderDate As Date
        Public CustomerID As String
        Public Details() As OrderDetails
    End Class
    Public Class Details
        ProductID As Integer
        Price As Integer
        Qty As Integer
    End Class
```

In our code we can create and initialize an Order object with statements such as the following (the order contains two detail lines):

```
Dim O As New Order
O.ID = 1000
O.orderDate = #9/15/2004#
O.CustomerID = 'ALFKI'
Dim details(1) As Order.OrderDetails
Details(0) = New OrderDetails
Details(0).Product = 27
Details(0).Price = 25.93
Details(0).Qty = 14
Details(1) = New OrderDetails
Details(1).Product = 6
Details(1).Price = 46.85
Details(1).Qty = 6
O.Details = Details
```

Then we can send this message to the Orders queue using the XmlMessageFormatter. Assuming that MSG is a Message variable and MSQ a properly initialized MessageQueue variable, the following statements will send the message to the queue referenced by the MSQ variable:

```
MSG.Body = O
MSQ.Send(MSG)
```

The message is serialized into XML format and written to the queue, and here's what the order looks like:

```
<?xml version="1.0"?>
<Order xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ID>0</ID>
  <orderDate>2003-05-14T00:00:00.0000000:00</orderDate>
  <CustomerID>ALFKI</CustomerID>
  <Details>
    <OrderDetails>
      <ProductID>27</ProductID>
      <Price>25.93</Price>
      <Qty>14</Qty>
```



```
</OrderDetails>
<OrderDetails>
  <ProductID>6</ProductID>
  <Price>46.85</Price>
  <Qty>6</Qty>
</OrderDetails>
</Details>
</Order>
```

You will see the code that reads the XML description of the message's body in the following section, where we discuss how to retrieve messages from a queue.

The messages written to a queue must be retrieved and processed by another application. To read messages from a specific queue, the application should know the type of objects stored in the queue. Again, you must create the appropriate `XmlMessageFormatter` object and assign it to the `Formatter` property of a `MessageQueue` object that references the specific queue:

```
Dim targetType(0) As Type
targetTypes(0) = GetType(Order1)
msgQ.Formatter = New XmlMessageFormatter(targetTypes)
```

Once the `MessageQueue` object has been set up, you can read messages off the queue either synchronously or asynchronously. The `Receive` method returns the first message in the queue; this is the oldest of the messages with the highest priority in the queue. The same message is also removed from the queue, and the next time you call the `Receive` method it reads the next message. If the queue is empty, the `Receive` method will wait until a message becomes available. The `Receive` method's purpose is to continuously monitor a queue and return a message as soon as one arrives at the queue. The following loop keeps reading messages off a queue:

```
While True
  newOrder = orderQ.Receive()
  ProcessOrder(newOrder)
End While
```

This loop is quite impractical, as you need a mechanism for terminating it when there are no longer messages to be read. If there are no messages in the queue, the `Receive` method waits until one arrives—and freezes the application's interface. One overloaded form of the `Receive` method accepts a second argument, which is a `TimeSpan` object. The `Receive` method waits for as long as specified with the `TimeSpan` argument for a message to arrive in the queue and reads it. If the queue isn't empty, the `Receive` method returns the first message in the queue immediately, of course. If the specified interval expires and no new message arrives at the queue, the `Receive` method throws a runtime exception. Listing 14.1 shows a simple technique for reading all the messages in a queue:

LISTING 14.1: READING MESSAGES SYNCHRONOUSLY

```
While True
  Try
    newOrder = orderQ.Receive(New TimeSpan(0,0,1))
```

```
    Catch ex As Exception
        Console.WriteLine("Processed all messages in queue")
        Exit Sub
    End Try
    ProcessOrder(newOrder)
End While
```

The `newOrder` variable is a `Message` variable. Before you can use the object retrieved from the queue, you must cast it to the appropriate type. There's an even better method of reading the messages of a specific queue: using an enumerator. The `MessageEnumerator` class encapsulates all the functionality you need to peek at or retrieve the queue's messages, and it's discussed next.

USING THE MESSAGEENUMERATOR CLASS

The best technique for reading messages off a queue is to create an `Enumerator` object that iterates through the messages in the queue. The `MessageEnumerator` class exposes the `MoveNext` method, which positions the enumerator to the next message (there's no `MovePrevious` method; the `MessageEnumerator` provides forward-only access to the messages of a queue). When you create a `MessageEnumerator` on a queue, it's placed right before the first message, so you must initially call the `MoveNext` method to land to the first message. The current message is returned by the `Current` property of the enumerator and is not removed from the queue. To remove a message from the queue, call the `RemoveCurrent` method. The `Reset` method, finally, resets the enumerator right before the first message in the queue.

The following statements form the core of a routine that iterates through all messages in a queue. `MSQ` is a properly initialized `MessageQueue` object:

```
Dim MSG As Message
Dim QEnum As Messaging.MessageEnumerator
QEnum = MSQ.GetEnumerator
While QEnum.MoveNext
    MSG = QEnum.RemoveCurrent
    ' PROCESS CURRENT MESSAGE,
    ' WHICH IS REPRESENTED BY THE MSG VARIABLE
End While
```

You can have multiple applications reading messages off the same queue with a `MessageEnumerator`. The `MoveNext` method retrieves the next available message from the queue, and multiple enumerators don't interfere with one another. If new messages are added to the queue, they will be picked by the enumerator as they become available.

WARNING *When you read a message from a queue you must also process it. If you retrieve a message and fail to process it, the message will be lost. On the other hand, if you process the message and fail to retrieve it from the queue, the same message will be processed again. MSMQ will acknowledge positively all messages retrieved from the queue and the application that requested the acknowledgment will assume that the message has been processed.*

[Team Fly](#)

 Previous

Next 

RETRIEVING MESSAGES ASYNCHRONOUSLY

The second method of reading messages off a queue is asynchronous: If no message is available we don't wait for one to arrive. Yet new messages are processed as soon as they arrive. To receive asynchronously, use the `BeginReceive` method. This method initializes a receive operation and fires the `ReceiveCompleted` event when the next message is read. The `MessageQueue` object that references the queue you want to read from must be declared at the form's level and with the `WithEvents` keyword:

```
Dim WithEvents MSQ as MessageQueue
```

To read messages asynchronously, you set up the `MSQ` object by setting its `Formatter` property and then call its `BeginReceive` method, as shown in Listing 14.2.

LISTING 14.2: RETRIEVING MESSAGES ASYNCHRONOUSLY

```
Dim MSQ As MessageQueue
MSQ = New MessageQueue(".\private$\orders")
Dim targetType() As Type
targetType(0) = GetType(Order1)
msgQ.Formatter = New XmlMessageFormatter(targetType)
msgQ.BeginReceive()
```

The `BeginReceive` method doesn't return a `Message` object; it simply initiates the reception of a message. When the message becomes available, the `ReceiveCompleted` event is fired. In this event's handler you must call the `EndReceive` method to acknowledge that you've read the message, then process it, and finally call the `BeginReceive` method again to initiate the process of reading the next message in the queue, as shown in Listing 14.3.

LISTING 14.3: PROCESSING THE RECEIVECOMPLETED EVENT

```
Private Sub msgQ_ReceiveCompleted(ByVal sender As Object, _
    ByVal e As System.Messaging.ReceiveCompletedEventArgs) _
    Handles msgQ.ReceiveCompleted
    Dim msg As Message = msgQ.EndReceive(e.AsyncResult)
    dim newOrder As Order = CType(msg.Body, Order)
    ProcessOrder(newOrder)
    msgQ.BeginReceive
End Sub
```

Both the `Receive` and `BeginReceive` methods accept an optional parameter, which is a `TimeSpan` object. This argument determines how long the method should wait for a message to arrive at the queue, if the queue is empty at the time the method is called. If you don't specify a timeout interval, both methods will wait for a message. This is not a problem for the asynchronous method, but the synchronous `Receive` method will block the application until a message arrives to the queue.

The last statement in the `ReceiveCompleted` event handler calls the `BeginReceive` method to continue monitoring for new messages. To terminate the asynchronous operation, you can call the `EndReceive` method but not call the `BeginReceive` method again.

When the specified interval expires and there's no message to be read in the queue, the `Receive` method throws a runtime exception. The `BeginReceive` method won't throw an exception; it simply won't fire the `ReceiveCompleted` event. We usually specify a timeout interval with the `Receive` method, but not with the `BeginReceive` method. Just bear in mind that the timeout interval of the `Receive` method doesn't mean anything on its own; the method throws an exception when the timeout interval expires, which means that you must embed the `Receive` method into a structured exception handler.

DELETING MESSAGES

Once a message has arrived at a queue, you can't edit it. This would be like editing an incoming message with your mail client application. The operation you can perform on a message besides retrieving it from the queue is to delete without processing it. To delete a message, call the `MessageQueue` class's `Delete` method. This method accepts a single argument, which is the path of the queue, and doesn't return any value. It simply deletes the top message in the queue. If the queue is empty, the method simply returns. The `Delete` method is an instance method and you can call it without first creating a `MessageQueue` object (that's why it expects the queue's path as argument). The following statements are equivalent:

```
MSQ.Delete (MSQ.Path)
System.Messaging.MessageQueue.Delete (path)
```

where `MSQ` is a properly initialized `MessageQueue` variable and `path` is a string variable that represents an existing queue's path.

PEEKING AT MESSAGES

In addition to retrieving messages from a queue, you can also peek at a message without removing it from the queue. To peek at a message, use the `Peek` method of the `MessageQueue` class. The `Peek` method returns a copy of the first message in the queue, but it doesn't remove the message from the queue. The `Peek` method doesn't accept any arguments and returns a `Message` object. Its behavior is similar to that of the `Receive` method: If the queue is empty, the `Peek` method will wait for a new message to arrive and freeze the application. You can pass to the method a `TimeSpan` object that represents the time interval it should wait for a message to arrive. If no message arrives in the specified interval, a runtime exception is thrown. Finally, you can peek at the top message asynchronously with the `BeginPeek` method, which is identical to the `BeginReceive` method. The difference is that when peeking at the queue, the message is not removed from the queue.

In addition to peeking at the first message in the queue, you can peek at any message in the queue with the `PeekByID` method, regardless of the message's order in the queue. This method expects a message ID as argument. We hardly ever know the message's ID, because this ID is generated by the system when a message is created. You can also peek at messages related to another message with the `PeekByCorrelationID` method. This method, like the `RetrieveByCorrelationID` method, accepts a message ID as argument and returns a collection of messages that are related to the message with the

events will cause a message to be sent to the acknowledgment queue. These events include the arrival of a message at its destination queue, the retrieval of a message from the destination queue, and so on.

Messages are not acknowledged instantly. A message sent to a remote queue may take quite a while to reach its destination. If the local machine is not connected to the network, or the destination queue is not available at any time, a message may take hours or days to be delivered. Waiting for an acknowledgment that may or may not arrive is not an option either. When we request an acknowledgment for a message, we also specify a time interval. If the operation completes within the specified interval, a positive acknowledgment is generated. If not, a negative acknowledgment is generated.

You can set a time-out for two operations: the delivery of a message to the destination queue and the retrieval of the message from the queue. The fact that a message regarding an order has reached its queue successfully isn't valuable information, because the message may not be read from the queue. If such a message remains unread in the queue for too long, the order should probably be canceled. To set the two time-out intervals, set the properties `TimeToReachQueue` and `TimeToBeReceived` of the appropriate `Message` object. Both properties' value is a `TimeSpan` object that represents the interval in which we want the operation to complete successfully. If the message is not delivered (or received) within the specified interval, the message is removed from its current queue and the appropriate negative acknowledgment is sent to the specified queue.

In addition, you must specify the type of action that will be acknowledged with the `AcknowledgeTypes` property. The `AcknowledgeTypes` property can be set to one of the members of the `AcknowledgeTypes` enumeration, which are listed in Table 14.2.

TABLE 14.2: THE MEMBERS OF THE `ACKNOWLEDGETYPES` ENUMERATION

| MEMBER NAME | DESCRIPTION |
|---------------------------------------|---|
| <code>FullReachQueue</code> | A positive acknowledgment is created if the message reaches the destination queue successfully, and a negative acknowledgment if the message fails to be delivered within the specified interval. |
| <code>FullReceive</code> | A positive acknowledgment is created if the message is retrieved from the destination queue within the specified interval, and a negative acknowledgment otherwise. |
| <code>NegativeReceive</code> | A negative acknowledgment is created if the message fails to be received from the queue. Note that peeking at a message does not remove it from the queue. |
| <code>None</code> | No acknowledgment message is created. |
| <code>NotAcknowledgeReachQueue</code> | A negative acknowledgment is created if the message does not reach the queue before the specified interval expires. |
| <code>NotAcknowledgeReceive</code> | A negative acknowledgment is created if the message is not received before the specified interval expires. |

PositiveArrival

A positive acknowledgment is created when the original message reaches the queue.

PositiveReceive

A positive acknowledgment is created when the message is received successfully.

[Team Fly](#)

 Previous

Next 

arrList.RemoveAt(1)

```
ListBox1.Items.Clear()  
ListBox1.Items.AddRange(arrList.ToArray)  
  
End Sub  
End Class
```

Notice that you don't have to use `For...Next` or other loop code to feed the data from an array to a `ListBox`. Instead, you can simply slap it in with the `ListBox`'s `AddRange` method. Or you could bind the data in an array directly to a `ListBox` (a technique illustrated later in this chapter).

You can, alternatively, simply specify an element's *contents* as another way of removing the element. Replace the line in boldface in the example above with the following line:

```
arrList.Remove('Pearl Harbor')
```

In this and other ways, the `ArrayList` sits somewhere between the limited, classic array and features more traditionally found in database management.

Because an `ArrayList` is *dynamic*—reallocating memory as needed when you add items to it—you need not worry about making such adjustments. However, you can set an `ArrayList`'s `Capacity` property explicitly if you wish, and you can freely resize an `ArrayList` at any time by changing this `Capacity` property. If you don't expect to add any more new elements to an `ArrayList`, you can free memory by using the `TrimToSize` method.

Mass Manipulation

An `ArrayList` can manipulate a range of its elements *en masse* by appending, inserting, reading, or removing the range all at once. This example reads a range (replace the code in the previous example's `Button1_Click` event with this):

```
Private Sub Button1_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles Button1.Click  
    Dim RangeOfArrList As ArrayList = arrList.GetRange(0, 2)  
    ListBox1.Items.Clear()  
    ListBox1.Items.AddRange(RangeOfArrList.ToArray)  
End Sub
```

Here the `GetRange` method specifies (start index, number of elements in range). Then that range is copied into a new `ArrayList` named *RangeOfArrList*.

Data Binding

You can now bind `DataGrids`, `ListBoxes`, and most other controls to an array, `HashTable`, or other collection. Prior to .NET you could only bind to a database or a recordset. Using the previous

You can combine multiple types of acknowledgments with the Or operator. For example, you can set the Acknowledgment Types property to:

```
PositiveArrival Or PositiveReceive
```

to indicate that a positive acknowledgment should be generated when the message arrives at the destination queue, as well as when the message is retrieved from the queue. Assuming that the `MessageQueue1` component represents a valid message queue, the statements of Listing 14.4 send a message to a local queue and request a positive acknowledgment for the event of the arrival of the message to the destination queue and a negative acknowledgment if the message isn't retrieved within 10 seconds of the moment it was sent. Just add a new button to the form of the test project you're using and enter the statements shown in Listing 14.4 in its Click event handler.

LISTING 14.4: SENDING A MESSAGE WITH AN ACKNOWLEDGMENT REQUEST

```
Private Sub Button3_Click(ByVal sender As System.Object, _  
                          ByVal e As System.EventArgs) _  
                          Handles Button3.Click  
  
    Dim msg As New Message  
    msg.Body = "Message to be acknowledged"  
    msg.Label = "ACK Message"  
    msg.AdministrationQueue = New MessageQueue(".\private$\ackQueue"  
    msg.AcknowledgeType = AcknowledgeTypes.FullReachQueue Or _  
                          AcknowledgeTypes.NegativeReceive  
  
    msg.TimeToBeReceived = New TimeSpan(0, 0, 10)  
    msg.Recoverable = True  
    MessageQueue1.Send(msg)  
End Sub
```

This button sends a simple message to the queue represented by the `MessageQueue1` component and requests an acknowledgment. Since the message is sent to a local queue it will arrive at the destination queue, but it will not be retrieved from the queue. The arrival of the message should be acknowledged with a positive acknowledgment message and the retrieval of the message with a negative acknowledgment message, 10 seconds later. You can change the setting of the Acknowledge Type and the `TimeToBeReceived` properties to experiment with the various acknowledgment types. Figure 14.3 shows the contents of the `ackQueue` local queue, which stores the acknowledgment messages, after a sending a few messages. As you can see, positive acknowledgment messages are indicated by a little green arrow at their lower left corner while negative acknowledgment messages are indicated by a red arrow at the same location. Under the Class heading of the snap-in, you can view a short description of the acknowledgment message ("The message reached the queue" for positive acknowledgments and "The time-to-be-received has elapsed" for negative acknowledgments). Negative acknowledgment messages have a body, which is the body of the message that either failed to arrive at the destination queue or wasn't read by an application at the destination queue. Positive acknowledgment messages don't have a body. Acknowledgment messages are not removed automatically from their queues.

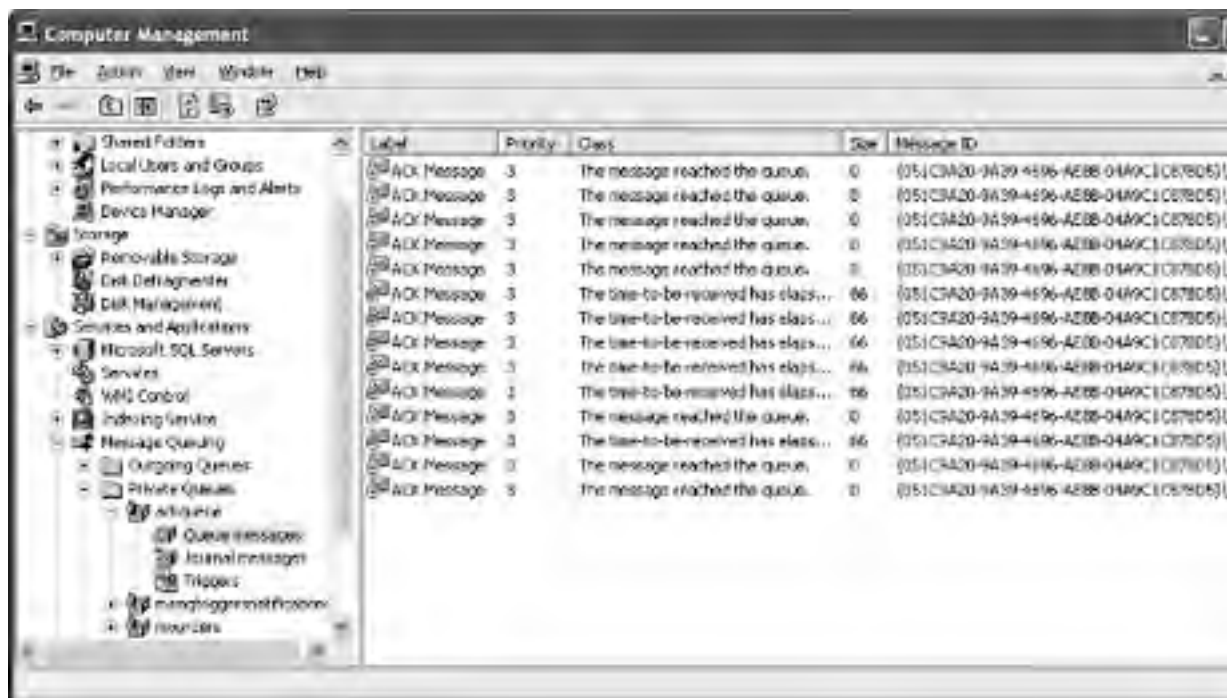


FIGURE 14.3 Viewing acknowledgment messages with the MSMQ snap-in

Notice that you don't have to set the Acknowledge Types property of each message you send. You can specify the settings of all messages sent to a specific queue with the `DefaultProperties ToSend` property of the `MessageQueue` object that represents the destination queue. This way you can use even the simple form of the `Send` method to send your messages and still take advantage of the acknowledgment features of MSMQ. The following code segment requests the same acknowledgment type as we did in the preceding example, only this time through the `DefaultProperties ToSend` property:

```
With msgQ.DefaultPropertiesToSend
    .AdministrationQueue = New MessageQueue(".\private$\ackQueue")
    .AcknowledgeType = AcknowledgeTypes.FullReachQueue _
        Or AcknowledgeTypes.FullReceive
End With
msgQ.Send( _
    "Please acknowledge the arrival and retrieval of this message")
```

Processing Acknowledgment Messages

For an acknowledgment message to be of any use, the application that requested it must receive it and take the appropriate action. Unlike regular messages, acknowledgment messages have no body. Instead, they expose a number of other properties, such as the `SentTime` property (the date and time the message was sent), the `MessageType` property (the message's type), and many more. These properties, however, are not received automatically by the `Receive` method. You must set the appropriate member of the `MessageReadPropertyFilter` property to `True` for each property you want to read. To read the values of the `SentTime`, `MessageType`, and `DestinationQueue` properties, you must use a few statements like the

following:

```
With Q.MessageReadPropertyFilter
    .SentTime = True
    .MessageType = True
    .DestinationQueue = True
```

[Team Fly](#)

 Previous

Next 

```
End With
Dim msg As Message
msg = Q.Receive
```

where Q is a properly initialized MessageQueue object that represents a specific queue.

The Read Top Acknowledgment Message button on the form of the SimpleQueue application reads the first message in the ackQueue queue, and its Click event handler is shown in Listing 14.5:

LISTING 14.5: PROCESSING AN ACKNOWLEDGMENT MESSAGE

```
Private Sub btnTopAckMessage_Click(ByVal sender As System.Object, _
                                   ByVal e As System.EventArgs) _
    Handles btnTopAckMessage.Click
    Dim Q As New MessageQueue(".\private$\ackQueue")
    Q.MessageReadPropertyFilter.SenderId = True
    Q.MessageReadPropertyFilter.Id = True
    Q.MessageReadPropertyFilter.SentTime = True
    Q.MessageReadPropertyFilter.MessageType = True
    Q.MessageReadPropertyFilter.DestinationQueue = True
    Dim msg As Message
    msg = Q.Receive
    TextBox1.Clear()
    TextBox1.AppendText(_
        "SENDER ID: " & _
        System.Text.Encoding.Unicode.GetString(msg.SenderId) & _
        vbCrLf)
    TextBox1.AppendText("MESSAGE ID: " & msg.Id.ToString & vbCrLf)
    TextBox1.AppendText("TIME SENT: " & msg.SentTime & vbCrLf)
    TextBox1.AppendText("MESSAGE TYPE: " & _
        msg.Acknowledgment.ToString & vbCrLf)
End Sub
```

Reading the top message off an acknowledgment queue is not a very useful operation. In most cases we want to read the acknowledgment(s) generated for a specific message. We do so by calling the ReceiveByCorrelationID method passing the ID of the original method as argument. This method will retrieve the acknowledgment messages generated for a specific message, identified by its ID. If no acknowledgment message has arrived for the specified message, the ReceiveByCorrelationID method will throw a runtime exception.

Use the SimpleQueue project to write a few messages to the ToolkitQueue queue. The Send Message & Request Acknowledgment button sends simple messages and stores their IDs to the CorrelationIDs ArrayList. This ArrayList is declared on the form's level, so that all event handlers can access it. The code behind the Send Message & Request Acknowledgment button is shown in Listing 14.6. It basically creates a new Message object, sets its AdministrationQueue, Acknowledge Type, and Time ToBeReceived properties, and sends the message. In addition, it saves the ID of this message to the CorrelationIDs ArrayList, so that it can later retrieve the acknowledgment messages associated with this message.

LISTING 14.6: REQUESTING ACKNOWLEDGMENT FOR A NEW MESSAGE

```
Private Sub btnMsgAcknowledge_Click( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles btnMsgAcknowledge.Click
    Dim msg As New Message
    msg.Body = "Message to be acknowledged"
    msg.Label = "ACK Message"
    msg.AdministrationQueue = New MessageQueue(".\private$\ackQueue")
    msg.AcknowledgeType = AcknowledgeTypes.FullReachQueue Or _
        AcknowledgeTypes.NegativeReceive
    msg.TimeToBeReceived = New TimeSpan(0, 0, 10)
    msg.Recoverable = True
    msg.AttachSenderId = True
    MessageQueue1.Send(msg)
    CorrelationIDs.Add(msg.Id)
    TextBox1.Clear()
    TextBox1.AppendText( _
        "The following message was written to the queue: " & vbCrLf)
    TextBox1.AppendText("ID " & msg.Id & vbCrLf & _
        "BODY " & msg.Body & vbCrLf)
    TextBox1.AppendText(">>> ACKNOWLEDGMENT REQUESTED <<<")
End Sub
```

The `AcknowledgeType` property in the sample application is set to request a negative acknowledgment for a delivery failure and a positive acknowledgment of a successful delivery to the destination queue. In other words, we request an acknowledgment message for two events: the successful arrival of the message to the destination queue (`FullReachQueue`) and the failure to retrieve the message from the queue (`NegativeReceive`). The time-out interval for the message's receipt is set to 10 seconds. You can experiment with other types of acknowledgments by changing the setting of the `AcknowledgeType` property.

The acknowledgment message(s) will be delivered to the queue specified by the `Administration-Queue` property. These messages, however, are not regular messages; they have no body. Open the MSMQ snap-in and create the `ackQueue` private queue. Then start the Simple Messages application and click the button `Send Message & Request Acknowledgment` a few times. Each time, a new message will be sent and a positive acknowledgment message will arrive at the `ackQueue`. The acknowledgment message's size is zero bytes. However, it has an ID and its `Class` property is the string "Message reached queue." This message will arrive almost instantly, because the `MessageQueue1` queue is on the local computer. Ten seconds later another message will arrive, this time a negative acknowledgment message ("Time to be received timed out"), indicating that the message wasn't retrieved from the destination queue within the specified interval. Figure 14.3 shows the contents of the `ackQueue` queue after sending a few messages with acknowledgment requests.

Now click the `Read Top Acknowledgment Messages` to read the first message in the `ackQueue` queue. The `Retrieve` message won't retrieve much information from the queue, because acknowledgment

messages have no body (if you select a message in the ackQueue queue and look at its properties, you'll see that the Body tab of the property pages is empty). To specify the properties of the message to be read, you must set the MessageReadPropertyFilter property of the MessageQueue object that represents the queue with the acknowledgment messages. Listing 14.7 shows the code behind the button that retrieves the top acknowledgment message in the queue.

LISTING 14.7: RETRIEVING AND PROCESSING AN ACKNOWLEDGMENT MESSAGE

```
Private Sub btnTopAckMessage_Click( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles btnTopAckMessage.Click
    Dim Q As New MessageQueue(".\private$\ackQueue")
    Q.MessageReadPropertyFilter.SenderId = True
    Q.MessageReadPropertyFilter.Id = True
    Q.MessageReadPropertyFilter.SentTime = True
    Q.MessageReadPropertyFilter.MessageType = True
    Q.MessageReadPropertyFilter.DestinationQueue = True
    Dim msg As Message
    Try
        msg = Q.Receive(New TimeSpan(0, 0, 1))
    Catch ex As Exception
        TextBox1.AppendText("Couldn't read any acknowledgment messages")
        Exit Sub
    End Try
    TextBox1.Clear()
    TextBox1.AppendText("SENDER ID: " & _
        System.Text.Encoding.Unicode.GetString(msg.SenderId) & _
        vbCrLf)
    TextBox1.AppendText("MESSAGE ID: " & msg.Id.ToString & vbCrLf)
    TextBox1.AppendText("TIME SENT: " & msg.SentTime & vbCrLf)
    TextBox1.AppendText("MESSAGE TYPE: " & _
        msg.Acknowledgment.ToString & vbCrLf)
End Sub
```

After reading the acknowledgment message, the code prints the values of certain properties of the message on a TextBox control.

Retrieving individual acknowledgment messages isn't a common practice. It's more practical to keep track of the IDs of all messages sent through a queue and after a while retrieve the acknowledgment messages for each ID. This is a task you can perform at the end of the day, or even use timers to signal to your application the expiration of each message. Let's assume that each message sent to a specific queue must be retrieved from its destination queue within 48 hours. You can program a Timer to fire an event after 48 hours and a few minutes. This event should also report the message ID for which the event is fired; in the event's handler, you can retrieve the acknowledgment message for the specific ID. If no negative acknowledgment message is found in the acknowledgment queue, you

know that the message has been retrieved from the destination queue and you need not take any special action. We're assuming that most messages will be delivered successfully, so we request negative acknowledgments for the messages that were not delivered successfully.

Another technique for processing acknowledgment messages is to request negative acknowledgments only and write a loop that runs in the background and retrieves messages from the acknowledgment queue continuously. For every negative acknowledgment, you must take the appropriate action (repeat the message, log an error entry, send the same message to another queue, and so on).

The Process Acknowledgment Messages of the SimpleQueue application retrieves from the ackQueue queue the messages associated with the ID of each of the sent messages. The code, shown in Listing 14.8, iterates through the IDs of the messages it has sent to the MessageQueue1 queue. At each iteration it retrieves the acknowledgment message associated with a specific message ID (all the IDs correspond to messages sent by the application with the Send Message & Request Acknowledgment button).

The acknowledgment message is classified according to the value of its Acknowledgment type and displayed on the TextBox control at the bottom of the form. Notice that the ReceiveByCorrelationID retrieves a single acknowledgment message, while the queue may contain up to two acknowledgment messages for each regular message. You must either edit the code to retrieve all the messages specified with a particular ID, or edit the code that generates the messages and request a single acknowledgment. We usually request a negative acknowledgment for messages that weren't retrieved from the destination queue within the specified time interval.

LISTING 14.8: PROCESSING THE ACKNOWLEDGMENT MESSAGE

```
Private Sub btnAckRead_Click(ByVal sender As System.Object, _
                             ByVal e As System.EventArgs) _
    Handles btnAckRead.Click
    Dim Q As New MessageQueue(".\private$\ackQueue")
    Q.MessageReadPropertyFilter.SenderId = True
    Q.MessageReadPropertyFilter.Id = True
    Q.MessageReadPropertyFilter.SentTime = True
    Q.MessageReadPropertyFilter.MessageType = True
    Q.MessageReadPropertyFilter.DestinationQueue = True
    Dim msg As Message
    Dim id As String
    TextBox1.AppendText(Now.TimeOfDay.ToString)
    For Each id In CorrelationIDs
        Try
            msg = Q.ReceiveByCorrelationId(id)
            Select Case msg.Acknowledgment
                Case Acknowledgment.ReachQueue
                    TextBox1.AppendText(vbCrLf & "MESSAGE " & id & _
                                         " reached destination queue")
                Case Acknowledgment.Receive
                    TextBox1.AppendText(vbCrLf & "MESSAGE " & id & _
                                         " received")
                Case Acknowledgment.ReachQueueTimeout
```



```
        TextBox1.AppendText(vbCrLf & _
            'MESSAGE ' & id & _
            " timed out (did not reach destination queue)")
    Case Acknowledgment.ReceiveTimeout
        TextBox1.AppendText(vbCrLf & "MESSAGE " & id & _
            " timed out (not received)")
    End Select
Catch ex As Exception
    TextBox1.Clear()
    TextBox1.AppendText("No acknowledgment for message " & _
        id & " found" & vbCrLf)
    TextBox1.AppendText(ex.Message)
End Try
Next
End Sub
```

Fault Tolerance and Load Balancing

MSMQ provides two more features, which aren't quite obvious and haven't been sufficiently stressed in our discussion: fault tolerance and load balancing. Messages are guaranteed to be delivered to the destination queue, even if there's no connection at the time they're created. If a message can't be delivered to the remote machine at the time of its creation (the moment the Send method is executed), it's written to an outgoing queue on the local machine. Messages in the outgoing queues will be sent to their respective queues as soon as this become possible—that is, as soon as the local machine establishes a connection to the remote machine. You don't have to do anything about the messages in the outgoing queues, because MSMQ will transmit them automatically. Of course, you must not forget to set the Message object's Recoverable property to True, so that the message is actually written to disk, rather than saved in memory. This ensures that the messages will remain in the outgoing queues even after you turn off the local computer.

Messages may even be routed to other nodes between the local machine and the remote computer. This is the case with public queues, which may reside on a different domain of the network. MSMQ routes the messages to their destination and the local machine may not have direct access to the destination queue. In short, messages will be delivered to their destination unless it's physically impossible (if the remote machine doesn't exist, for example, or if the remote queue has been renamed).

Even if the delivery of a message can't take place within a specified time interval (which is specified by the sending application), MSMQ detects the condition and generates the appropriate acknowledgment messages, provided that the sending application has requested the generation of acknowledgment methods. MSMQ is a reliable robust mechanism for exchanging messages between two computers, and it exhibits the fault tolerance necessary in building loosely coupled applications. In other words, MSMQ won't drop any messages, no matter what. Whether a message can't be delivered to another local queue on the same network because the sending application doesn't have permission to write to this queue, or a message can't be delivered to a remote computer because the two machines can't establish a connection to one another, you can write applications to handle

the messages that failed to be delivered and/or processed. All the functionality you need is exposed by MSMQ through a simple object model, which you already know how to program against.

The second feature of MSMQ is load balancing. You can have multiple applications reading messages off a queue. Each time an application requests a message from the queue, the message is passed to the application and removed from the queue. MSMQ doesn't actually balance the messages among multiple applications, but you can run the same instance of the same application on multiple machines and each application will retrieve and process a number of messages without interfering with the other applications. In effect, this is a form of load balancing. You can even use the `MessageEnumerator` class to create a dynamic view of the queue. As messages are added to the queue, or removed from the queue, the enumerator is adjusted accordingly. This is possible because the enumerator always gets the next message from the queue. The enumerator doesn't know which is the next message until it's requested to read it from the queue. Only then does it pick the message from the queue and present it to the application.

But why should we have multiple applications processing the same queue's messages? The applications are reading messages of the same type and will most likely update the same database, or perform identical actions. There may be situations in which a single application can't process all the messages within a reasonable time interval. Let's say you have a commercial site on the Web. The site, which may actually reside on an ISP's machines, logs all orders into a queue. It could be a queue on the same machine as the web server, or on one of the servers on your company's network. Once an order arrives at your company, it must be processed. This means that it must be entered into an order-tracking system. It's very likely that you will have many people processing the orders. Depending on the address of the customer, you may process different orders at different locations. In any case, the orders need not be moved from the queue into your order-tracking system by a single machine. Every workstation that runs the order-tracking application can read messages off the queue and log them into a local database. An order read at a specific warehouse will most likely be processed locally. If not, it will probably be forwarded to another warehouse through a different queue. The idea is that a queue's messages need not be processed from a single workstation. Any of the available workstations can process any order—as long as two or more workstations don't read the same order from the queue. If you make sure that messages are removed from the queue as they're read, then no two applications can read the same message.

Let's look at a simple application that reads messages off a queue in a load-balanced manner. We'll write a single application and then run many instances of it to simulate a scenario in which multiple workstations read messages from the same queue. You can run multiple instances of this application on the same machine, or copy the executable on multiple workstations on your network and run them. Under test conditions, all workstations will process approximately the same number of messages. In a production environment, the least loaded workstations will process more messages than the more loaded workstations.

Figure 14.4 shows several instances of the `MSMQLoadBalancing` project. The `Add Messages To Queue` button adds 50 new messages to the `LoadBalancedQueue` queue of the local

machine. You can click this button on any of the running instances of the application to add messages to the queue. The Retrieve Messages From Queue button reads messages off the same queue, one at a time. We've inserted a three-second delay between reading messages to simulate some computational load. The messages are simple strings and the application just reads them from the queue. A real message will take a few moments to a few seconds to be processed.

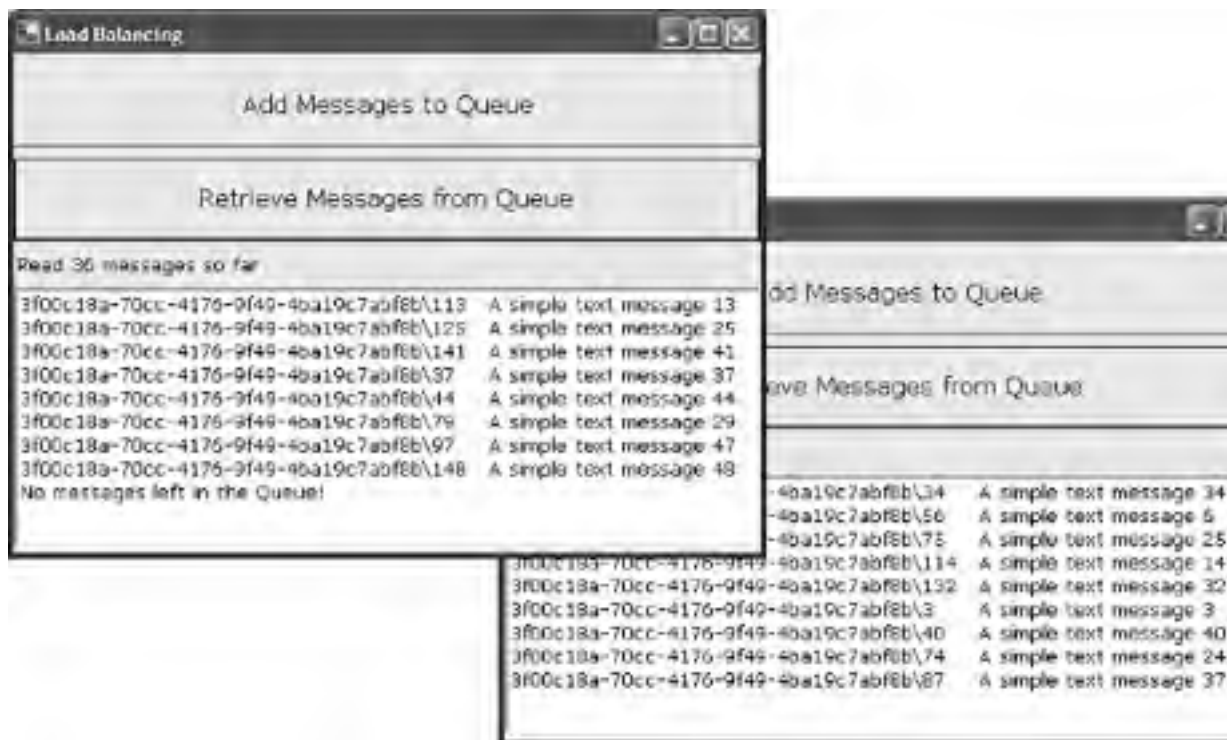


FIGURE 14.4 Consuming a queue's messages from within multiple clients

Let's look at the application's code. When the form is first loaded, the program creates a reference to the `LoadBalancedQueue` queue. If the queue doesn't exist, the program creates it with the statements of Listing 14.9:

LISTING 14.9: CREATING AND SETTING UP THE BALANCEDQUEUE

```
Private Sub Form1_Load( _  
    ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) _  
    Handles MyBase.Load  
    If MessageQueue.Exists(".\private$\BalancedQueue") Then  
        MQ = New MessageQueue(".\private$\BalancedQueue")  
    Else  
        MQ = MessageQueue.Create(".\private$\BalancedQueue")  
    End If  
    Dim types(0) As Type  
    types(0) = GetType(System.String)  
    MQ.Formatter = New XmlMessageFormatter(types)  
End Sub
```

The `MQ` variable is a form-level `MessageQueue` variable, which must be accessed by the procedures that read and write to the queue. We use an `XmlFormatter` so that we can later read the message's `Body` property. The messages written to the queue are simple strings. Listing 14.10 shows the code behind the Add Messages To Queue button:

LISTING 14.10: CREATING RANDOM MESSAGES

```
Private Sub btnAddMessages_Click( _  
    ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) _  
    Handles btnAddMessages.Click  
  
    Dim msg As Message  
    Dim i As Integer  
    For i = 1 To 50  
        msg = New Message('A simple text message ' & i.ToString)  
        msg.Label = "BalancedQueue Message"  
        MQ.Send(msg)  
    Next  
    MsgBox("50 new messages added to BalancedQueue")  
End Sub
```

The code so far is fairly trivial. Let's look at the code that reads the messages off the queue. The code creates an enumerator to access the messages in the queue, the `QEnum` variable. Then it iterates through the elements of the enumerator with a `While` loop and removes the top message from the queue. Each message's ID and body are printed on the `TextBox` control at the bottom of the form. Between messages, the code calls the main thread's `Sleep` method to simulate a three-second delay, as shown in Listing 14.11:

LISTING 14.11: ENUMERATING THE MESSAGES IN A QUEUE

```
Private Sub btnReadMessages_Click( _  
    ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) _  
    Handles btnReadMessages.Click  
  
    Dim MSG As Message  
    Dim QEnum As Messaging.MessageEnumerator  
    QEnum = MQ.GetEnumerator  
    Dim nMessages As Integer  
    While QEnum.MoveNext  
        MSG = QEnum.RemoveCurrent  
        System.Threading.Thread.CurrentThread.Sleep(3000)  
        nMessages += 1  
        Label2.Text = "Read " & nMessages.ToString & " messages so far"  
        TextBox1.AppendText(MSG.Id & vbTab & MSG.Body & vbCrLf)  
        Application.DoEvents()  
    End While  
    TextBox1.AppendText("No messages left in the Queue!" & vbCrLf)  
End Sub
```

To see how the load is balanced among multiple instances of an application, compile the project and minimize (or close) the Visual Studio window. Then locate the executable file in the project's Bin folder and start several instances of it. Click the Add Messages To Queue button on one of the forms on the desktop to add 50 new messages to the queue. A message box will pop up to confirm the operation, and then you can click the Retrieve Messages From Queue button on all the forms. All applications will be reading messages off the queue using an enumerator, and each enumerator will return the top message from the queue to its application. The enumerators do not interfere with one another and allow multiple applications to process messages from the same queue. A single instance of the application that processes the messages may not be able to keep up with the incoming flow of messages, but you can run multiple instances of the same application on multiple workstations to process the messages of a queue in a timely manner.

A single instance of the application would read 50 messages in 150 seconds, because of the three-second delay between messages. Three instances of the same application will process the same 50 messages in one-third of the time, because the Sleep method doesn't keep the processor busy. It has the same effect as running each instance of the application on a separate machine. By the way, we could have implemented the same application with threads, but this would have added a substantial complexity to the code, which would obscure the point we're demonstrating. Besides, there would be no performance benefit, because all the threads would be running on the same computer. By allowing multiple workstations to access the same queue, we can actually reduce the total time needed to process all the messages.

The call to the DoEvents method in the code gives the processor a chance to update the TextBox control and allows you to switch to another instance of the application and start reading messages before the current application processes all the messages in the queue. You can use the Add Messages To Queue button on any of the forms on the desktop to add messages to the queue. No matter which instance of the application adds them to the queue, the messages will be processed by all applications in a load-balanced manner.

You can stop any instance of the MSMQLoadBalancing application. As long as there are running instances of the application, they will keep processing the messages in the queue. As you can see, a simple message-processing application based on the Messaging class is both load-balanced and fault-tolerant. The beauty of implementing messaging applications is that these two features are available for free. No special code is required and there's no performance penalty. The applications that read the messages need not be aware of one another and need not communicate with one another. You can think of MSMQ as a very simple form of multithreading. Let's say you need to perform a number of different tasks. If you can implement a class that describes each task, you can then create instances of this class, put them in a queue, and have multiple applications read the messages and take the appropriate action.

The applications that retrieve messages from a queue are called server applications, while the applications that put messages in a queue are the client applications. You may find it odd, since the server application is usually the unique one that processes requests from multiple clients. In the context of MSMQ, the client applications leave messages to a queue, to be processed by one or more server applications.

[Team Fly](#)

 Previous

Next 

Transactional Messages

In addition to moving messages around in a robust manner, MSMQ can send and receive multiple messages in a transactional mode. If one of the messages involved in the transaction fails, then all the messages will be rolled back: the sent messages will not be allowed to appear in their destination queues and the read messages will be placed back in their corresponding queues.

To send and receive transactional messages, you must first create a new queue and, when you're prompted to enter the queue's name in the Queue Name dialog box, check the box Transactional (see Figure 14.5). You can create a transactional queue with the Create method of a MessageQueue object by specifying an additional argument after the queue's name. This argument is a Boolean value that determines whether the new queue will be marked as transactional. The simple form of the Create method creates non-transactional queues. To find out whether an existing queue is transactional or not, examine the value of the Transactional property of the MessageQueue object that represents the specific queue.



FIGURE 14.5 Creating a transactional queue

Transactional queues can only send transactional messages and an attempt to send a non-transactional message through a transactional queue will cause an exception. Likewise, any attempt to send a transactional message through a non-transactional queue will also cause an exception. It's possible to send a non-transactional message through a transactional queue, as long as you use the form of the Send method that specifies the context of a transaction.

You will also have to insert a few additional statements in your code. First, you must create a `MessageQueueTransaction` object. This object must be passed as argument to all `Send` and `Receive` methods that will participate in the transaction. There are two more methods that can participate in transactions: the `ReceiveByID` and the `ReceiveByCorrelationID` methods. To specify that the action performed by the corresponding method is part of a transaction, you simply pass an additional argument, which is an instance of the `MessageQueueTransaction` class.

The `MessageQueueTransaction` class exposes three methods for controlling a transaction: the `Begin` method, which marks the beginning of a transaction, the `Commit` method, which commits a transaction, and the `Abort` method, which aborts a transaction. When a transaction is aborted, any messages sent or received are rolled back. When messages are sent as part of a transaction, the messages aren't visible at the destination queue, nor do they generate any acknowledgment messages, until

the transaction is committed. Listing 14.12 demonstrates the basic steps in sending messages in the context of a transaction. We're using the simple form of the Send method, but you can replace it with a few statements that create a new Message object and send it.

LISTING 14.12: SENDING TWO MESSAGES IN A TRANSACTION

```
Dim MTX As New MessageQueueTransaction()
Dim MQ As New MessageQueue(".\private$\TrxOrders")
MTX.Begin()
Try
    MQ.Send("The first message in the transaction", msgTx)
    MQ.Send("The second message in the transaction", msgTx)
    MTX.Commit()
Catch
    MTX.Abort
Finally
    MQ.Close
End Try
```

Implementing transactions with MSMQ isn't much different than implementing database transactions. In addition to the three basic methods for controlling the transaction, the MessageQueue-Transaction class exposes the Status property, which returns the current status of the transaction. This property can return one of the members of the MessageQueueTransactionStatus enumeration, which are the following: *Initialized* (the transaction object has been created, but no message has been sent, or read, in the context of this transaction), *Pending* (the transaction is still in progress), *Committed* (the transaction has completed successfully), and *Aborted* (the transaction has been aborted).

To demonstrate the use of transactional queues, we'll use the order tracking system as an example. The computer that receives the orders creates a message for each order and sends it to a queue. Another application, running on a different, possibly remote, computer reads the messages off the queue and processes them. The application that reads the messages off the queue may log an entry or send a confirmation message to the customer. Without transactions, it's possible to read the message off the queue but fail to log the appropriate entry. Another fairly common situation is to send the same message to two different queues. You could send the order to a warehouse to process it, as well as to another computer at the headquarters. MSMQ allows you to wrap the submission or retrieval of multiple messages in the context of a transaction with a few additional statements.

You can also use transactions as you read messages from a queue. You can start a transaction before retrieving a message and commit it after its successful processing. If the processing fails for any reason, you can abort the transaction and return the message to the queue. You must add statements to your code to handle this situation, because it's likely that the same error will occur the next time the application attempts to process the same message. For example, you can insert a few statements that will move the message to a different queue, create a different message and send it to the machine where the message originated, and so on.

processes them. The orders aren't uploaded in real time, but only when the client computer has access to the server's queue. Until then, orders are stored in a local queue, the Outgoing queue, whose messages will be automatically uploaded to the remote queue. You can have multiple instances of the client application, all sending messages to the same queue.

In Chapter 18 we're going to build an application for entering orders into the Northwind database. The application will allow the user to enter the details of an order on a ListView control and then commit the entire order to the database. In this section we'll present a variation of the application that doesn't commit the orders to the database. Instead, it serializes each order into a new message and sends it to the NWOrders queue of the local machine. We'll also write an application that retrieves the orders from the queue and processes them (it simply commits the orders to the database). Figure 14.6 shows the DisconnectedOrders application that creates the orders and Figure 14.7 shows the OrdersServer application that processes the orders. The OrdersServer application's interface allows an operator to review each order and decide whether to submit it to a local database for further processing, or reject it.



FIGURE 14.6 Preparing an order with the DisconnectedOrders application



| | | | | | |
|--------------|-------------------|-------|----------|---|--------------|
| 26 | Gumbär Gumbärchen | 31.23 | 1 | 0 | 31.23 |
| TOTAL | | | 6 | | 89.73 |

◀ Commit Order Delete Order ▶

FIGURE 14.7 Processing the order messages generated by the OrdersServer application

We're not going to get into the details of the `DisconnectedOrders` application here (the application's code is discussed in detail in Chapter 18). In this section we'll look at the code that creates an `Order` object out of the data entered by the user on the form of Figure 14.6 and sends it to the `NWOrders` local queue.

Before we explore the code of the application, we'd like to discuss a few practical issues. The `DisconnectedOrders` application, as its name implies, is meant to be used offline, on a salesperson's notebook computer. To take an order from a customer, the salesperson must have an up-to-date pricelist. The application should be able to work with a local copy of the pricelist and should download the current pricelist from the server whenever the two computers are connected. If the price of a product changes while the salesperson is on the go, the product will be sold at its old price. The application running on the notebooks of the salespersons will create a message for each new order and send it to a specific queue, on a remote machine. Messages will be stored to an outgoing queue at the local computer and delivered to the destination queue when the two computers connect.

The application's `Data` menu has two items, the `Download Tables` and `Populate Tables` commands. The `Download Tables` command downloads the pricelist from the server into a `DataSet` and persists the `DataSet` to an XML file. The `Populate Tables` command populates the same `DataSet` with the data stored in the XML file. The customers are stored in another table of the same `DataSet`. Users should be able to enter new customers into the database. We're not showing here how to enter new customers to the client `DataSet`. Instead, we'll work with the existing customers. You can easily modify the application to accept new customers. You must create a new custom class for storing new customer data and submit them to a different (or the same) queue. It's possible to send messages of different types to the same queue, as long as you differentiate one of their properties. For example, you can set the `Label` property of the messages that represent orders to `"NWOrder"` and the `Label` property of the messages that represent customers to `"NWCcustomers."` Or you can set the `Category` property of the two types of messages. The idea is that the application that processes the messages will be able to quickly determine each message's type and cast it accordingly. You should also assign a higher priority to messages that correspond to customers, so that they'll be processed before their orders.

We'll use the Northwind sample database in our application. The basic columns of the `Products` table (product IDs, names, and prices) and the `Customers` table (customer IDs, names, and so on) will be downloaded to each client, where they will be persisted to an XML file. The application can populate a `DataSet` from the XML file at any time, regardless of whether it's connected to the server at the time. Obviously, every time the salesperson connects to the company's server to upload orders, he should also update the price list on his portable computer.

Preparing Orders

The class we'll use to store orders is the `NewOrder` class, whose definition is shown in Listing 14.13. The header of the order has just two fields, the `CustomerID` and `EmployeeID` fields. The order's details are stored in an array of `OrderedProduct` objects. Each element of this array corresponds to one of the order's detail lines.

[Team Ely](#)

 Previous

Next 

LISTING 14.13: THE NEWORDER CLASS'S DEFINITION

```
Public Class OrderedProduct
    Public ProductID As String
    Public ProductPrice As Decimal
    Public ProductQTY As Integer
    Public ProductDiscount As Decimal
End Class

Public Class NewOrder
    <XmlAttributeAttribute()> Public CustomerID As String
    <XmlAttributeAttribute()> Public EmployeeID As Integer
    Public Details() As OrderedProduct
End Class
```

When the application starts, we set up the `MSQ` object from within the form's Load event handler. This variable represents the queue to which the order messages will be sent. It's a local queue in our example, but you can use a queue on a remote machine just as easily. To serialize the `NewOrder` objects in XML format we'll use the `XMLMessageFormatter`. Listing 14.14 shows the code for setting up the `NWOrders` queue.

LISTING 14.14: SETTING UP A QUEUE FOR THE ORDERS

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Lo
    MSQ = New MessageQueue
    MSQ.Path = ".\private$\nworders"
    MSQ.DefaultPropertiesToSend.AttachSenderId = True
    MSQ.DefaultPropertiesToSend.UseJournalQueue = True
    MSQ.DefaultPropertiesToSend.Label = "NW Order"
    Dim MTypes(0) As Type
    MTypes(0) = GetType(OrdersClass.NewOrder)
    MSQ.Formatter = New XmlMessageFormatter(MTypes)
End Sub
```

To enter a new invoice, select a customer's name in the first tab and then switch to the Details tab of the form. Enter a product's ID in the ID TextBox control and press Enter. Alternatively, you can enter the first few characters of the product's name in the Product Name box and press Enter. If more than one product matches the description, a ListBox with the matching product names will appear. Select the desired product and then enter the quantity and the product's discount (or press Enter to accept the default values). When you're done, click the Save Invoice button to finalize the order. Instead of submitting the new order to the database, the `DisconnectedOrders` application will create a new instance of the `NewOrder` class and store the invoice to the newly created object. This object is then serialized and sent to the `MSQ` queue. These actions take place from within the Save Order button's Click event handler, which is shown in Listing 14.15.

LISTING 14.15: SUBMITTING A NEW ORDER TO THE NORTHWIND DATABASE

```
Private Sub btnAccept_Click(ByVal sender As System.Object, _
                            ByVal e As System.EventArgs) _
    Handles btnAccept.Click
    ' Combine rows that refer to the same product.
    ' See the ReduceRows() subroutine for more information.
    ReduceRows()
    ' Create an array list of OrderedProduct items
    Dim orderedItems(lvOrderGrid.Items.Count - 1) As _
        OrdersClass.OrderedProduct
    Dim P As OrdersClass.OrderedProduct
    Dim i As Integer
    ' Iterate through the items of the ListView control and
    ' add a new OrderedProduct object to the ArrayList
    ' for each ListView item (i.e., each ordered product)
    For i = 0 To lvOrderGrid.Items.Count - 1
        P = New OrdersClass.OrderedProduct
        P.ProductID = CInt(lvOrderGrid.Items(i).Text)
        P.ProductPrice = CDec(lvOrderGrid.Items(i).SubItems(2).Text)
        P.ProductQTY = CInt(lvOrderGrid.Items(i).SubItems(3).Text)
        P.ProductDiscount = _
            CDec(lvOrderGrid.Items(i).SubItems(4).Text / 100)
        orderedItems(i) = P
    Next
    Dim order As New OrdersClass.NewOrder
    order.CustomerID = lstCustomers.SelectedValue
    order.EmployeeID = cmbEmployees.SelectedValue
    order.Details = orderedItems
    Dim MSG As New Message()
    MSG.Body = order
    MSG.Label = "'Remote Order'"
    MSG.Recoverable = True
    MSQ.Send(MSG)
    MsgBox("Ordered saved successfully." & vbCrLf & _
        "Press OK to prepare a new order")
End Sub
```

Processing Orders

Let's switch our attention to the application that processes the orders. This is the InvoiceServer application, whose interface is shown in Figure 14.7. The Retrieve Orders button retrieves copies of all orders in the `NWOrders` queue and stores them in the `Orders` `ArrayList`. This `ArrayList` contains a collection of `NewOrder` objects. In addition, we store each message's ID in the `IDs` `ArrayList` collection (we'll need this ID later to commit and delete orders).

The code behind this button calls the `GetAllMessages` method, which returns copies of the messages but doesn't retrieve them from the queue. We have not used an enumerator, because it's a forward-only structure. Our approach allows the user to move back and forth through the messages and commit orders to the database, or delete orders, at will. Every time you click the Retrieve Orders button, a different set of messages will be copied to the `ArrayList`: the messages you processed (or deleted) will be missing, while new messages that have arrived recently will be included in the collection.

As it's read, each message is converted into an instance of the `NewOrder` class and stored in the `Orders ArrayList`. The message's ID is stored in the `IDs ArrayList` and you'll see shortly how this ID is used to delete processed messages from the queue. After reading all the messages with a `For ...Next` loop, the program shows the first message with the `ShowOrder()` subroutine and enables the navigational buttons at the bottom of the form. The code behind the Retrieve Orders button is shown in Listing 14.16.

LISTING 14.16: RETRIEVING COPIES OF THE MESSAGES IN THE QUEUE

```
Private Sub btnGetOrders_Click(ByVal sender As System.Object, _
                               ByVal e As System.EventArgs) _
    Handles btnGetOrders.Click
    MSQ = New MessageQueue
    MSQ.Path = ".\private$\NWOrders"
    MSQ.DefaultPropertiesToSend.AttachSenderId = True
    MSQ.DefaultPropertiesToSend.UseJournalQueue = True
    MSQ.DefaultPropertiesToSend.Label = "NW Order"
    Dim MTypes(0) As Type
    MTypes(0) = GetType(OrdersClass.NewOrder)
    MSQ.Formatter = New XmlMessageFormatter(MTypes)
    Try
        Messages = MSQ.GetAllMessages
    Catch ex As Exception
    End Try
    If Messages Is Nothing Then
        MsgBox("Could not retrieve messages from the specified queue")
        Exit Sub
    End If
    If Messages.Length = 0 Then
        MsgBox("There were no messages in the specified queue")
        Exit Sub
    End If
    Dim order As OrdersClass.NewOrder
    Dim currMessage As Message
    Dim iMsg As Integer
    Orders.Clear()
    For iMsg = 0 To Messages.GetUpperBound(0)
        Console.WriteLine(Messages(iMsg).Id)
        currMessage = Messages(iMsg)
        order = New OrdersClass.NewOrder
```

```
        order = CType(currMessage.Body, OrdersClass.NewOrder)
        Orders.Add(order)
        IDs.Add(Messages(iMsg).Id)
    Next
    currentOrder = 0
    ShowOrder(CType(Orders(currentOrder), OrdersClass.NewOrder))
    btnNext.Enabled = True
    btnPrevious.Enabled = True
    btnCommit.Enabled = True
    btnDelete.Enabled = True
End Sub
```

The code behind the Next and Previous buttons, shown in Listing 14.17, is quite simple. Both buttons adjust the `currentOrder` variable, which is an integer and points to the current order in the collection. The current order is the one currently displayed.

LISTING 14.17: THE NAVIGATIONAL BUTTONS

```
Private Sub btnNext_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
    Handles btnNext.Click
    If currentOrder = Orders.Count - 1 Then
        MsgBox("This is the last order")
        Exit Sub
    End If
    currentOrder += 1
    ShowOrder(CType(Orders(currentOrder), OrdersClass.NewOrder))
End Sub

Private Sub btnPrevious_Click(ByVal sender As System.Object, _
                              ByVal e As System.EventArgs) _
    Handles btnPrevious.Click
    If currentOrder = 0 Then
        MsgBox("This is the first order")
        Exit Sub
    End If
    currentOrder -= 1
    ShowOrder(CType(Orders(currentOrder), OrdersClass.NewOrder))
End Sub
```

When the Commit Order button is clicked, the code shown in Listing 14.18 casts the current message into an instance of the `NewOrder` type and processes its fields. The fields are passed as arguments to the `AddHeader` and `AddDetailLine` stored procedures. The `AddHeader` stored procedure inserts the order's header, and the `AddDetailLine` stored procedure inserts a detail line of the

order to the Order Details table. The AddDetailLine stored procedure is executed as many times as there are detail lines in the order. The insertion of the appropriate rows in the database takes place from within a transaction, so that if any of the operations fails, the entire transaction is aborted. If the order is committed successfully, the message is removed from the queue with the ReceiveByID method of the MessageQueue class.

LISTING 14.18: COMMITTING AN ORDER TO THE DATABASE

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

    Dim order As OrdersClass.NewOrder
    order = CType(Orders(currentOrder), OrdersClass.NewOrder)
    Dim CMD As New SqlClient.SqlCommand
    CMD.CommandText = 'AddHeader'
    CMD.CommandType = CommandType.StoredProcedure
    CMD.Parameters.Add(New SqlClient.SqlParameter( _
                        "@customerID", order.CustomerID))
    CMD.Parameters.Add(New SqlClient.SqlParameter( _
                        "@employeeID", order.EmployeeID))

    Dim CN As New SqlClient.SqlConnection
    CN.ConnectionString = "data source= "(local); & _
                        "initial catalog=Northwind; " & _
                        "user id=YOUR ID ;password= YOUR_PASSWORD"

    CMD.Connection = CN
    Dim TRN As SqlClient.SqlTransaction
    CN.Open()
    TRN = CN.BeginTransaction
    CMD.Transaction = TRN
    Try

        Dim OrderID As Integer
        OrderID = CInt(CMD.ExecuteScalar())
        CMD = New SqlClient.SqlCommand
        CMD.Connection = CN
        CMD.Transaction = TRN
        CMD.CommandText = "AddDetailLine"
        CMD.CommandType = CommandType.StoredProcedure
        CMD.Parameters.Add(New SqlClient.SqlParameter( _
                            "@OrderID", Data.SqlDbType.Int))
        CMD.Parameters.Add(New SqlClient.SqlParameter( _
                            "@ProductID", Data.SqlDbType.Int))
        CMD.Parameters.Add(New SqlClient.SqlParameter( _
                            "@Quantity", Data.SqlDbType.Int))
        CMD.Parameters.Add(New SqlClient.SqlParameter( _
                            "@Price", Data.SqlDbType.Money))
        CMD.Parameters.Add(New SqlClient.SqlParameter( _
                            "@Discount", Data.SqlDbType.Real))
```

```
Dim Item As OrdersClass.OrderedProduct
For Each Item In order.Details
    CMD.Parameters("@OrderID").Value = OrderID
    CMD.Parameters("@ProductID").Value = Item.ProductID
    CMD.Parameters("@Quantity").Value = Item.ProductQTY
    CMD.Parameters("@Price").Value = Item.ProductPrice
    CMD.Parameters("@Discount").Value = Item.ProductDiscount
    CMD.ExecuteNonQuery()
Next
Catch exc As Exception
    TRN.Rollback()
    CN.Close()
Exit Sub
End Try
TRN.Commit()
MSQ.ReceiveById(Messages(currentOrder).Id)
Orders.RemoveAt(currentOrder)
CN.Close()
ShowNextOrder()
End Sub
```

The transaction involves database operations only. In Chapter 16 you'll see how to implement transactions that involve both database and MSMQ operations. This is a fairly advanced topic that requires the installation of serviced components, and we'll postpone its discussion until Chapter 16.

The deletion of a message is simpler. We retrieve the ID of the message to be deleted from the IDs ArrayList and use it to retrieve the corresponding message from the queue. The Orders collection stores the orders we read from the queue with the Retrieve Orders button, but no ID that relates an order to its matching message. That's why we had to store the IDs of the messages separately. There are other techniques to deal with this problem. For example, you could add a new member to the Order class and use it to store the ID of the corresponding message. Listing 14.19 shows the statements that remove an order from the MSMQ message queue. Notice that the code removes the order's ID from the IDs ArrayList as well.

LISTING 14.19: DELETING AN ORDER AND THE CORRESPONDING MESSAGE

```
Private Sub btnDelete_Click(ByVal sender As System.Object, _
                            ByVal e As System.EventArgs) _
    Handles btnDelete.Click
    MSQ.ReceiveById(CStr(IDs(currentOrder)))
    Orders.RemoveAt(currentOrder)
    IDs.RemoveAt(currentOrder)
    ShowNextOrder()
End Sub
```

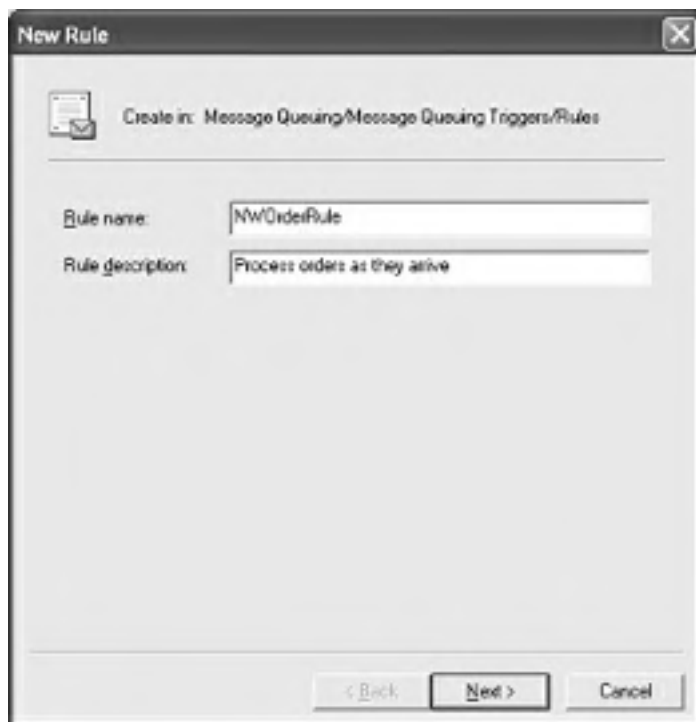



FIGURE 14.8 Creating a new rule

Click the Next button to see the next window of the New Rule wizard, where you can add conditions to the rule. Conditions are selected from a drop-down list and are evaluated every time a message arrives. If all conditions are true, the trigger will be fired. The conditions are basically properties of the message, such as whether the message's label contains a specific string, whether the message's body contains a specific string, the ID of the computer that sent the message, and so on. After selecting a condition, you must enter the value that will be used for the comparison. The rule shown in Figure 14.9 says that the trigger will be fired for messages whose Label contains the string "NWOrder." In the drop-down list with the conditions, select "Message label contains" and in the box below enter the string "NWOrder" (without the quotes). Then click the Add button to add the condition to the rule, as shown in Figure 14.9. A rule may contain multiple pairs of conditions/comparison values.

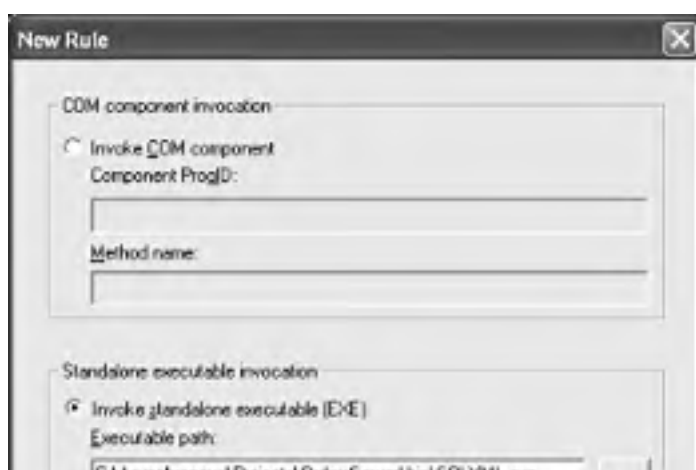




FIGURE 14.9 Establishing condition for a rule

Click Next again to see the last window of the New Rule dialog box, where you can define the action that will take place when the conditions are met. You can react to the trigger by executing a method of a COM component, or invoke a standalone executable, as shown in Figure 14.10. Check the appropriate option on the dialog box of Figure 14.10 and specify the programmatic ID of the COM component, or the full path to the executable that will service the request. You can specify any programmatic ID, or the path to any EXE file on your system. You'll have to return to the properties of the rule and revise it after you've written the application that will service the trigger.



FIGURE 14.10 Specifying an action for a rule

Whether you select a COM component or a standalone executable, MSMQ can pass as arguments one or more items related to the message that fired the trigger. There's a fixed number of parameters you can pass to the program that will service the trigger, and you can select any of them. To view the available parameters that you can pass as arguments, click the Parameters button; you'll see the Invocation Parameters window, where you can select the parameters to be passed to the program that will handle the trigger. In the Parameter drop-down list you can select one of a number of parameters and click the Add button to add it to the list of Invocation Parameters. You can add as many of the available parameters as you need. For example, you can pass to the program the message's body, but this will only work for simple text messages. We normally pass the message's ID to our application and use the RetrieveByID method to read the desired message from the queue.

Defining Triggers

Now that we have specified the necessary rules and the corresponding actions, we can create a new trigger. Right-click the Triggers items and select New ➤ Trigger from the context menu to see the dialog box in Figure 14.11. Set the trigger's name to "Order Trigger." In the Monitored Queue section, you can specify the queue that will be monitored. Check the User Queue option and enter the path of the queue where the orders are written (.\\private\$\nworders). In the Message Processing Type section of the window, you can specify how MSMQ will process the message. This property can be set to one of the following values: Peeking, Retrieval, or Transactional Retrieval. Check the

Retrieval option to be sure that messages are also retrieved when processed. Peeking may be a more reasonable choice for a practical application, because we want to be sure that a message has been processed before we remove it from the queue. To disable a trigger temporarily, clear the Enabled button in the dialog box shown in Figure 14.11. Note that the trigger-related windows for Windows 2003 server will be a little different than their XP versions, shown in the figures of this chapter.



FIGURE 14.11 Setting up a new trigger

Click the Next button to see the next window of the dialog box, where you can assign one or more of the existing rules to the current trigger. The rules you've created under the Rules branch of Message Queuing Triggers will appear in the Existing Rules box. You can select any one of the rules with the mouse and click the Attach button to add it to the trigger. After adding all the rules to the trigger, click the Finish button to create the trigger.

As soon as the new trigger is created, it will be activated and all the messages in the NWOrders queue will be processed. In the following section, we'll write a simple console application that retrieves the order that fired the trigger and displays the order's fields on the console window.

The ProcessOrders Console Application

Let's build an example to demonstrate how to set up and use Message Queuing Triggers. Earlier in this chapter we developed the `DisconnectedOrders` application, which takes orders and writes them to the `NWOrder` local queue. The `OrderServer` application retrieves the messages from the queue and processes them. If this application isn't running, then the messages will keep accumulating in the queue. We can use the Message Queuing Triggers service to process messages as soon as they arrive, without deploying an application that will run continuously in the background.

The application we'll build in this section is a console application, which will simply display the message in a console window. Once you know how to retrieve the message from the queue and cast it to the appropriate type, you can process it in any way you like. Start a new console application, the `ProcessOrders` application, and add the `OrdersClass` class to it (copy the class's definition from the

ProcessOrders application and paste it onto a new class module by the same name in the new project). We need the definition of the class so that we can extract messages that represent orders and create objects out of them, just as we did with the OrdersServer application. The trigger passes the following arguments to the application: the message's ID, the queue's path, and two date values (when the message was sent and when it arrived at the destination queue). We only really need the message's ID, but we've included a few more arguments to demonstrate how to parse multiple arguments. The trigger, which is the OrderTrigger, was set up to peek at messages, so we must remove the messages from within our code. Once the ID of the message is known, we can use the RetrieveByID method to remove from the queue the message that fired the trigger.

Before looking at the project's code, we'll discuss a couple of issues we ran into as we developed the code for the application. The arguments are passed to the application as command-line arguments. Each argument is embedded in a pair of quotes; consecutive arguments are separated by spaces. Here's a typical command-line argument that contains the ID of the message, the queue's path, and the date and time the message was sent and arrived at the queue:

```
"{051C9A20-9A39-4696-AE8B-04A9C1C87BD5}\6721" "POWERTOOLKIT\private$\NWOrders"  
"4/9/2003 11:24:51 AM" "4/9/2003 11:24:51 AM"
```

This string must be parsed. As you can see, there's no unique character that separates the various arguments. If you used the space as separator, the dates will split into three separate arguments. We decided to remove the first quote and then replace the string that consists of a double quote followed by a space and another quote with the Char(255). Then we can use this character to split the string into its constituent parts. In general, you should always examine the value of the string passed to the component and make sure that your parsing code will work. The following statements retrieve the command-line arguments as a string and then split this string to extract the individual arguments:

```
Dim commands As String = Microsoft.VisualBasic.Command()  
commands = commands.Replace(""" """, Chr(255))  
commands = commands.Substring(1)  
Dim separators As String = Chr(255)  
Dim args() As String = commands.Split(separators.ToCharArray)
```

Notice the format of the message's ID. It's exactly as it appears in the MSMQ snap-in, but if you attempt to use it as-is to retrieve the message from within your code, a runtime exception will be thrown to the effect that no message with the specified ID exists in the queue. You must remove the curly brackets that delimit the computer's unique identifier and then pass the remaining string as argument to the RetrieveByID method. Here's how the code retrieves the message that fired the trigger:

```
msgID = msgID.Replace("{", "")  
msgID = msgID.Replace("}", "")  
MSG = MSQ.ReceiveById(msgID)
```

The complete listing of the console application's Main subroutine is shown in Listing 14.20. The code retrieves the message that fired the trigger from the queue, uses it to construct an Orders-Class.NewOrder object, and displays the order's fields on a console window. You can process the order in any way you see fit; for example, you can duplicate much of the code of the OrdersServer project to commit the order to the database.

[Team Fly](#)

 Previous

Next 

LISTING 14.20: PROCESSING A TRIGGER

```
Sub Main()
    Dim separators As String = Chr(255)
    Dim commands As String = Microsoft.VisualBasic.Command()
    commands = commands.Replace(""" """, Chr(255))
    commands = commands.Substring(1)
    Dim args() As String = commands.Split(separators.ToCharArray)
    Dim msgID As String = args(0)
    Dim msgPath As String = args(1)
    Dim msgSentDate, msgArrivedDate As Date
    msgSentDate = args(2)
    msgArrivedDate = args(3)
    Console.WriteLine("MESSAGE ID: " & msgID & vbCrLf & _
        "MESSAGE PATH: " & msgPath & vbCrLf & _
        "MESSAGE SENT ON: " & msgSentDate & vbCrLf & _
        "MESSAGE ARRIVED ON: " & msgArrivedDate)

    Dim MSG As New Message
    Dim MSQ As New MessageQueue(".\private$\NWOrders")
    Dim typeNameNames(0) As Type
    typeNameNames(0) = GetType(OrdersClass.NewOrder)
    MSQ.Formatter = New XmlMessageFormatter(typeNameNames)
    msgID = msgID.Replace("{", " ")
    msgID = msgID.Replace("}", " ")
    MSG = MSQ.ReceiveById(msgID)

    Dim O As New OrdersClass.NewOrder
    O = CType(MSG.Body, OrdersClass.NewOrder)
    Console.WriteLine("CUSTOMER ID: " & O.CustomerID)
    Console.WriteLine("EMPLOYEE ID: " & O.EmployeeID.ToString)
    Dim dtl As OrdersClass.OrderedProduct
    Console.WriteLine("ORDER DETAILS")
    Console.WriteLine("ID" & vbTab & "QTY" & vbTab & _
        "PRICE" & vbTab & "DISC.")
    For Each dtl In O.Details
        Console.WriteLine(dtl.ProductID & vbTab & _
            dtl.ProductQTY & vbTab & _
            dtl.ProductPrice & vbTab & _
            dtl.ProductDiscount)
    Next
    Console.WriteLine()
    Console.WriteLine("Press any key to close this window")
    Console.Read()
End Sub
```

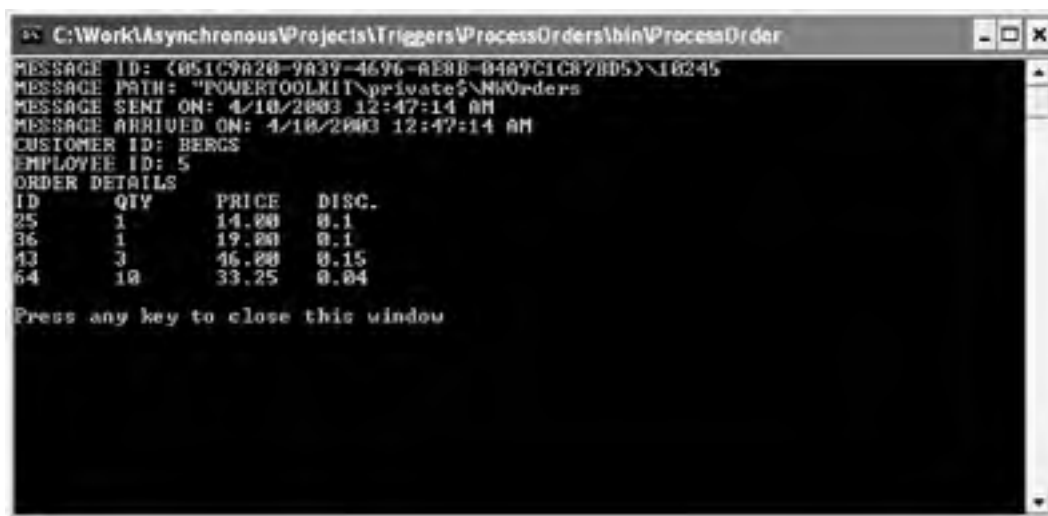


FIGURE 14.13 Processing a message with a trigger's action



FIGURE 14.14 The order corresponding to the data of Figure 14.13

Messages are sent from computer to computer using the MSMQ component of the Windows operating system. MSMQ takes care of many low-level details for us. We can, for instance, request that messages are acknowledged at their destination. If a message doesn't arrive on time at the destination queue, or a message isn't read from the queue within a specified time interval, MSMQ can notify the sending computer.

Using queues, you can write asynchronous applications that are both fault-tolerant and load-balanced. You can have multiple computers process messages from the same queue. If one of them fails, the remaining ones will continue processing the queue's messages. If a system accepts an unusual number of requests in a short time interval, it can write messages to a queue and process them later. The classic example is that of a system for processing time-

cards. A single machine is overwhelmed by requests early in the morning and in the evening. Instead of processing each card entered into the system in real time, we can create messages with each cards' data and send them to a queue. These messages can be processed at a slower pace than they were entered.

Finally, you can manage messages automatically, with the newly released Message Queuing Triggering component. This component allows you to set up rules for incoming messages and invoke a program automatically when a new message meets certain criteria. MSMQ is one of the most useful tools in building enterprise applications. We've demonstrated how to access MSMQ from within your VB applications with a practical example of forwarding and processing orders.

This page intentionally left blank.

Connection and Command classes are abstract classes; you can't use them directly in your code. The Connection class is the parent class for the SqlConnection and OleDbConnection classes, which provide connectivity features for SQL Server and OLE DB-compliant databases, respectively. The Command class is the parent class for the SqlCommand and OleDbCommand classes for SQL Server and OLE DB databases. Finally, there are two flavors of the DataAdapter class, the SqlDataAdapter and OleDbDataAdapter classes.

The DataSet is the primary data storage mechanism for the client. You can also retrieve data through a DataReader object, which returns the selected rows. It's your responsibility to extract the data from the DataReader into a data structure such as an ArrayList, or populate a control such as a ListBox or a ListView control. The DataSet is central in ADO.NET and most data-driven applications are based on this class. The data is moved to the client and stored into a DataSet, which is a miniature database. It contains as many tables as we specified in our query, and each table contains the columns and rows specified in the query. These tables are usually related (there are joins between pairs of tables), and the same relations can be established between the tables of the DataSet as the ones that exist in the database. The DataSet can even enforce relational and non-null constraints, but not arbitrary constraints such the ones you can specify at the database level.

The Visual Database Tools

We'll start our exploration of ADO.NET by going through the visual database tools of the Visual Studio IDE. It's a very brief overview meant for readers who are not familiar with these tools. Our goal is to show you how to use the ADO.NET classes in writing data-driven Windows applications. The visual tools are just wizards that allow you to set up the various ADO.NET objects through point-and-click operations. Once you know how to access a database with the visual tools, you'll find it easier to understand how to program the classes behind them.

We'll present the visual database tools by building a simple program to display and edit the products of the Northwind sample database. Start a new project and name it NWProducts. We'll start by setting up the objects that will enable us to view and edit the rows of the Products table in the Northwind database.

ESTABLISHING A CONNECTION

Our first step is to create a connection to the database. Open the Server Explorer tab of the Toolbox and you will see an item called Data Connections at the top. Under Data Connections there are connections to the databases you work with; it will be empty if you haven't established a connection to a database yet. Right-click Data Connections and select Add Connection. Another command in the context menu is the Create New SQL Server Database (this option is available only in the Professional and Enterprise Architect versions of Visual Studio). You can create a new SQL Server database right in Visual Studio's IDE, but the tools are almost identical to the tools of SQL Server. Databases are usually designed with SQL

Server's Enterprise Manager, but you can create small databases for convenience in the Visual Studio environment.

The Data Link Properties dialog box will pop up; here you can specify the database to which you want to connect and the properties of the connection. On the Provider tab you can select the database provider. In this book we'll focus on SQL Server databases, but setting up a connection to an Access database is very similar. On the Provider tab you must select the option "Microsoft OLE DB Provider for SQL Server," which is the default option. On the Connection tab, select the name of the

[Team Fly](#)

 Previous

Next 

database server on the top ComboBox and the authentication method in the second section on the tab. We're using Windows authentication so that you won't have to edit the properties of the connections in the sample projects. On the last ComboBox on the dialog box, select the name of the database (Northwind). The Connection tab of the Data Link Properties dialog box should look like the one shown in Figure 15.1. On the Advanced tab, you can set the connection's timeout (how long the connection will wait for the server to respond before it times out and closes), as well as access rights for the application when the server is on a network. Switch back to the Connection tab and click the Test Connection button to verify that the connection can actually see the database. Then close the dialog box. The item *computer.Northwind* will be added under Data Connections in your Server Explorer window (where *computer* is the name of your computer).

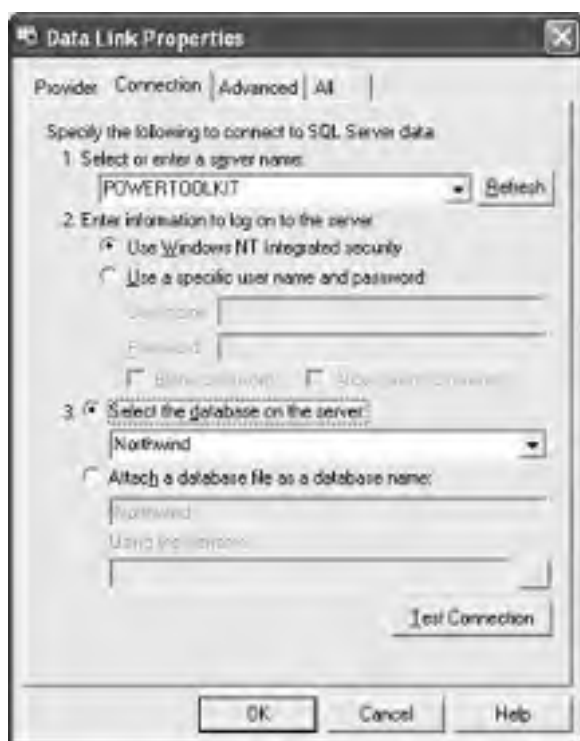


FIGURE 15.1 Setting up a connection to the North-wind database

CONFIGURING A DATAADAPTER

If you expand this connection object you will see the names of the database objects you can access, just like opening a database's branch in Enterprise Manager. Expand the Tables item and you will see the names of all tables in the database. Select the Products table and drop it on the form. Two new items will be added to the controls tray at the bottom of the designer: the `SqlConnection1` object, which is a Connection object for the Northwind database, and the `SqlDataAdapter1` object, which is a DataAdapter object for the Products table of the database. Rename the `SqlDataAdapter1` object to `DAProducts`. Select the DataAdapter and

right below the Properties window of the new DataAdapter, you will see the following links (you will find commands by the same names in the Data menu, which is visible only while a DataAdapter object on the form is selected):

Configure Data Adapter Starts a wizard to configure the DataAdapter.

Generate DataSet Generates a DataSet based on the properties of the DataAdapter.

Preview Data Retrieves the data selected by the DataAdapter for preview purposes.

Select **Configure Data Adapter** to start the **Data Adapter Configuration Wizard**. This wizard will guide you through the steps of setting up the SQL commands to be executed against the database. Click **Next** to go past the wizard's welcome screen. On the next screen you must select a connection (select the connection to the Northwind database you just designed) and click the **Next** button again. Notice that you can use the **New Connection** button to create a new connection from within the wizard. The steps are identical to the ones described earlier in this section. On the next screen you're prompted to specify the type of commands that the wizard will create: **SQL Statements**, **New Stored Procedures**, or **Existing Stored Procedures**.

Accept the default selection, **Use SQL Statements**, and click **Next** to continue. On the next screen you're prompted to enter the SQL statement that selects the desired rows from the **Products** table. Since we started the wizard by dropping the **Products** table on the designer, the wizard has generated a **SELECT** statement that selects all the columns of all rows in the table. This statement will move the entire **Products** table to the client, which is not what we usually want. In a production database, the **Products** table may be enormous in size. You can either specify a **SELECT** statement to retrieve selected columns of selected rows, or click the **Query Builder** button to build the query with visual tools, as shown in Figure 15.2. We'll assume you're familiar with SQL and we'll not discuss how to build SQL statements in this book. However, we will discuss in detail the statements generated by the wizard.

At the bottom of this screen is the **Advanced Options** button. If you click this button you will see a dialog box where you can specify a few more options that give you more control over how the statements will be generated. We'll look at these options in detail in the section "Concurrency Issues," later in this chapter.

Click the **Next** button again to end the wizard. If all went well, you'll see a list of the commands generated by the wizard. The wizard will generate the appropriate **SELECT**, **DELETE**, **UPDATE**, and **INSERT** statements for the **Products** table. If the query combined several tables, the wizard may not be able to generate update statements, and it will display the appropriate messages on the last form. Click **Finish** to end the wizard.





FIGURE 15.2 Specifying the rows and columns you want to retrieve from the Products table

[Team Fly](#)

 Previous

Next 

CREATING A DATASET

So far we created a Connection object to connect to the Northwind database and a DataAdapter that knows how to access the Products table. Although we specified only the SELECT statement, the DataAdapter has generated statements to update the table. Now we need a mechanism to download the selected data to the client, process them locally, and submit the changes back to the database. This mechanism is the DataSet object. Each DataAdapter should be associated with a specific table in the DataSet. Moreover, the DataAdapter exposes a few methods to execute the statements generated by the Wizard. The Fill method of the DataAdapter fills one of the tables of a DataSet with the selected data and the Update method submits the changes made to a specific table of the DataSet back to the database. It's possible to associate a DataAdapter with multiple related tables, but it won't be able to submit any changes to the database (in other words, the wizard won't generate the INSERT, UPDATE and DELETE statements).

Before you can download the table's rows to the client, you must create a DataSet for storing them. Select the Generate DataSet command from the Data menu (or click the link by the same name under the Properties window) and you'll see the Generate DataSet dialog box. Here you can create a new DataSet and specify which of the tables will be added to the DataSet. Check the name of the Products table on the dialog box of Figure 15.3, specify the name of the DataSet, and click OK.



FIGURE 15.3 Creating a new DataSet

A few moments later a new component will be added to your project, the `Products1` DataSet. In the Solution Explorer window you'll see the `Products.xsd` item, which is the schema of the DataSet. This is an XML file that describes the structure of the DataSet. The `Products1`

component added to your application is an instance of the Products class. Now place a Button and a DataGrid control on the form—the arrangement of the controls on the form is shown in Figure 15.6. Set the button's caption to "Populate DataSet" and enter the statements of Listing 15.1 in its Click event handler.

[Team Ely](#)

 Previous

Next 

LISTING 15.1: POPULATING THE PRODUCTS TABLE THROUGH A DATAADAPTER

```
Private Sub btnPopulate_Click(ByVal sender As System.Object, _  
                               ByVal e As System.EventArgs) _  
                               Handles btnPopulate.Click  
    Products1.Clear()  
    DAProducts.Fill(Products1, "'Products")  
    MsgBox("DataSet filled successfully")  
End Sub
```

First we clear the DataSet of its data and then we fill the Products table of the DataSet. The DataSet is cleared, because if you click the same button more than once, the DataAdapter will attempt to add the same rows to the existing DataSet. Each product has a unique ID and as soon as you attempt to add another row with the same ID, an exception will be thrown. If you run the program and click the Populate DataSet button, you'll see a message box informing you that the DataSet was filled successfully. But you can't see the data anywhere.

VIEWING THE DATASET

The few statements you've entered so far copy data from the database into the DataSet at the client. The quickest method to view the data in the DataSet is to bind the DataSet to a DataGrid control. Place an instance of the DataGrid control on the form and then set its DataSource property to `Products1` (you'll actually select this setting from a drop-down list). Then run the program again and click the Populate DataSet button. The data will appear on the DataGrid control and you can even edit them. You can't submit the changes to the server yet, but we'll get there shortly.

The DataGrid control is bound to the `Products1` DataSet. This means that the DataSet becomes the data source for the DataGrid control. The data in the DataSet are displayed on the DataGrid control and any changes you make to the DataGrid control's data are reflected in the DataSet. The DataSet is local to the client and the changes aren't automatically written to the database. When the DataSet is populated you can click the plus symbol next to the first (and only) row of the DataGrid. You will see the names of the tables in the DataSet. In our case, the DataSet contains a single table, the Products table. Click its name and you'll see the columns and rows of the Products table on the control. The DataGrid is a convenient tool for viewing a table's data without any substantial programming effort. However, it's not as convenient as you might expect. The CategoryID and SupplierID columns contain the IDs of the product's category and supplier, respectively. As you can understand, editing a table's rows on a DataGrid is not the most user-friendly approach. In our view, one of the common mistakes programmers do when it comes to database applications is that they try to build user interfaces based on the DataGrid control. Once you understand how the DataGrid control works and what it can do for your application, you'll be able to determine whether it's the appropriate tool for your application's needs or not. Most practical data-driven applications deploy an interface based on regular Windows controls; you'll see several examples of such interfaces in this book.

WORKING WITH MULTIPLE TABLES

Let's add some complexity to the application and make it a little more useful. This time we'll add the Categories and Suppliers tables to our DataSet. Stop the application and return to design mode.

[Team Fly](#)

 Previous

Next 

Then drop the Categories and Suppliers tables on the design surface. We don't want all the columns of the two tables, just enough information to identify categories and suppliers. Rename the two DataAdapter objects that will be created automatically when the two tables are dropped on the design surface to `DACategories` and `DASuppliers`. In the corresponding wizards, specify the following SELECT statements:

DACategories DataAdapter

```
SELECT CategoryID, CategoryName
FROM Categories
```

DASuppliers DataAdapter

```
SELECT SupplierID, CompanyName, ContactName,
ContactTitle, Phone, Fax
FROM Suppliers
```

These statements will configure the corresponding DataAdapters to retrieve enough columns from each table to identify the corresponding row. Then click the Generate DataSet button, select all three tables in the list (the Products, Suppliers, and Categories tables), and add them to the Products DataSet. You must also edit the statements that populate the DataSet by adding statements to retrieve the rows of the two new tables, as shown in Listing 15.2.

LISTING 15.2: POPULATING A DATASET WITH THREE DATATABLES

```
Private Sub btnPopulate_Click(ByVal sender As System.Object, _
                              ByVal e As System.EventArgs) _
    Handles btnPopulate.Click
    Products1.Clear()
    DAProducts.Fill(Products1, "Products")
    DACategories.Fill(Products1, "Categories")
    DASuppliers.Fill(Products1, "Suppliers")
    MsgBox("DataSet filled successfully")
End Sub
```

If you click the plus symbol at the first line on the grid, you will see the names of all three tables. Select the name of any table and you'll see its rows on the DataGrid control. To go back and select another table, click the control's Back button (the white arrow pointing to the left on the control's navigational section, near the top). The three tables are related, but their relations aren't reflected on the control; they must be established manually. Stop the program to return to design mode, and double-click the Products.xsd item in the Solution Explorer to see the definition (or schema) of the DataSet. The DataSet contains three tables, as shown in Figure 15.4. The arrows between the tables represent relations, but you won't see them initially; you must add them from within the IDE with simple point-and-click operations, which are described immediately.

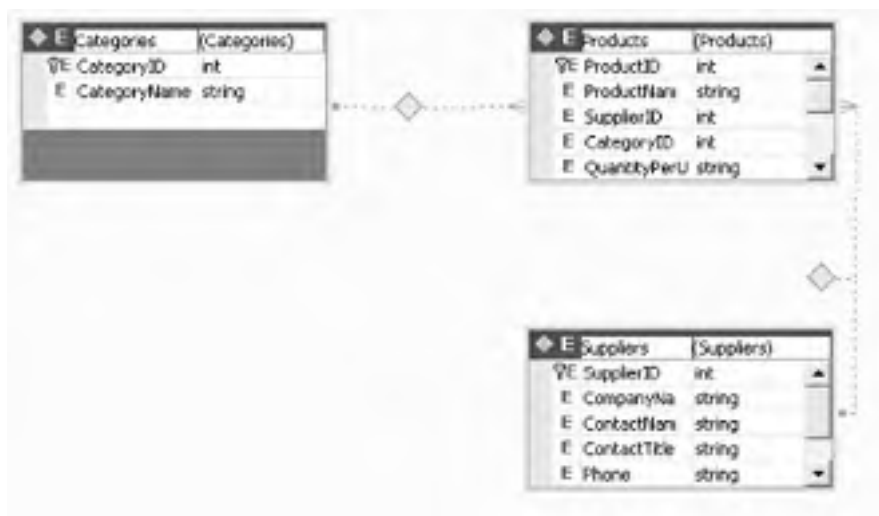


FIGURE 15.4 Viewing the schema of a DataSet generated with the visual database tools

Open the Toolbox, select the Relation item, and drop it on any of the tables on the design surface. The Toolbox contains items that apply to the design of XML schemas, not the usual controls. In this chapter we'll only discuss how to add relations between existing tables. The remaining items allow you to create a schema from scratch. As soon as the Relation item is dropped on one of the tables, you will see the Edit Relation dialog box, where you can define relations between pairs of tables. In the Parent element box you must specify the parent table of a relation and in the Child element you must specify the child element of the relation. Then, in the Fields box you must specify the fields that make up the relation. The first relation is between the Categories and Products table and is based on the CategoryID of both tables, as shown in Figure 15.5.

Then establish a relation between the Suppliers and Products tables based on the SupplierID field of both tables. Run the project and click the Populate DataSet button. A runtime exception will occur:

```
Failed to enable constraints. One or more rows contain values violating non-null, unique, or foreign-key constraints.
```

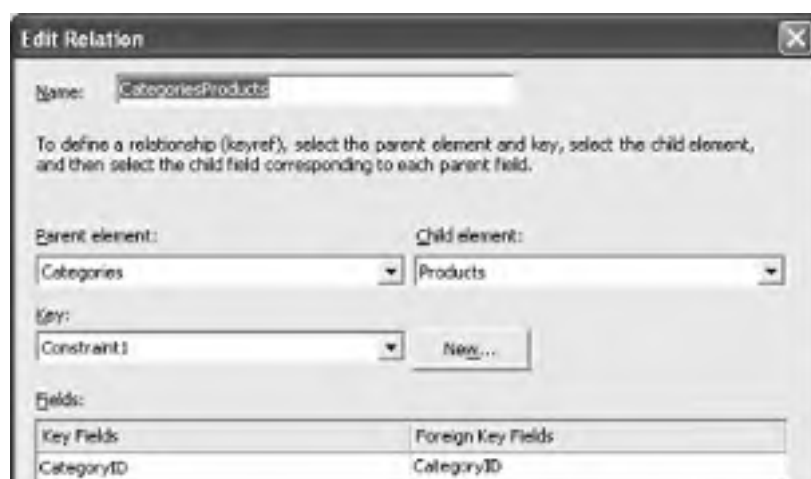




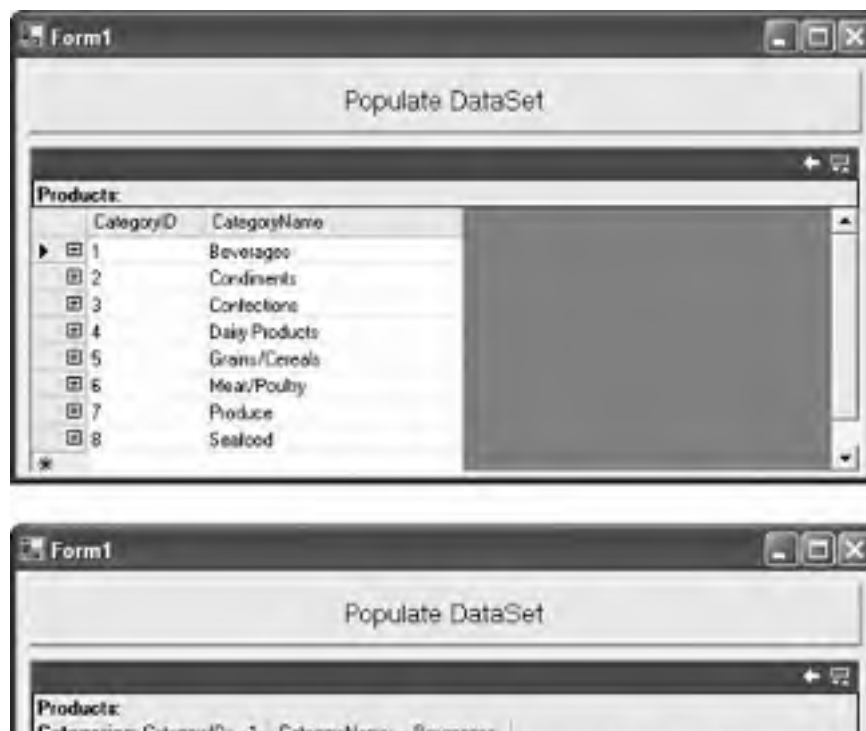
FIGURE 15.5 Establishing a relation between the Categories and Products tables

What this means is that you're attempting to load a row of the Products table that refers to a category, or supplier, that doesn't exist in the DataSet. To fix this problem, just change the order in which the tables are populated. You must populate the two parent tables first (the Categories and Suppliers tables) and then the child table. Here are the statements that fill the tables in the correct order:

```
DACategories.Fill(Products1, "Categories")  
DASuppliers.Fill(Products1, "Suppliers")  
DAProducts.Fill(Products1, "Products")
```

If you run the application now, the DataSet will be filled as expected. Select the Categories table on the DataGrid control, as shown in Figure 15.6 (a). In front of each row there's a plus symbol, which indicates that the row has child (related) rows in the DataSet. Click this symbol and you'll see the names of the relations that apply to this row. The Categories table is related to the Products table with the CategoriesProducts relation. If the same parent table was related to more child tables, you'd see more relation names here. Click the name of the CategoriesProducts relation and you'll see the rows of the Products table that are related to the selected category.

The DataGrid control allows us to visualize the relations between tables, but the suppliers and categories of each product are still displayed as integer values. There's no way to display a ComboBox with the names of the suppliers and categories on the DataGrid control, which is one of the most serious limitations of this control as a data entry tool. OK, it's possible to detect the coordinates of the selected row and display a custom ComboBox control there, but this requires a lot of code. The current version of the DataGrid control can't display lookup columns. A user-friendly interface can't be based on a grid control either. Most users would like to edit data on a form with typical Windows controls.



| ProductID | ProductName | SupplierID | CategoryID | QuantityPerUnit | UnitPrice | Units |
|-----------|-----------------|------------|------------|-------------------|-----------|-------|
| 1 | Chai | 1 | 1 | 10 boxes x 20 b | 18.0000 | 39 |
| 2 | Chang | 1 | 1 | 24 - 12 oz bottle | 19.0000 | 17 |
| 24 | Gustaf's Fantás | 10 | 1 | 12 - 355 ml can | 4.5000 | 20 |
| 34 | Sasquatch Ale | 16 | 1 | 24 - 12 oz bottle | 14.0000 | 111 |
| 35 | Steele's Stout | 16 | 1 | 24 - 12 oz bottle | 18.0000 | 20 |
| 38 | Côte de Blaye | 18 | 1 | 12 - 75 cl bottle | 260.5000 | 17 |
| 42 | Inch Coffee | 20 | 1 | 16 - 800 ml box | 46.0000 | 17 |

FIGURE 15.6 Viewing related rows on the DataGrid control

```
Dim at As Integer
Dim [end] As Integer
Dim count As Integer
[end] = str.Length
start = [end] / 2
Console.WriteLine()
Console.WriteLine("All occurrences (sic) of 'he' from position {0} to {1}.", _
start, [end] - 1)
Console.WriteLine("{1}{0}{2}{0}{3}{0}", Environment.NewLine, br1, br2, str)
Console.WriteLine("The string 'he' occurs at position(s): ")
count = 0
at = 0
While start <= [end] AndAlso at > - 1
    ' start++count must be a position within -str-.
    count = [end] - start
    at = str.IndexOf("he", start, count)
    If at = - 1 Then
        Exit While
    End If
    Console.WriteLine("{0} ", at)
    start = at + 1
End While
Console.WriteLine()
End Sub 'Main
End Class ' Sample
```

This example was obviously written by a C programmer who had little experience with Visual Basic programming. There are several giveaways. First is the use of a class to illustrate a simple, small programming technique. Creating a class for this purpose is overkill, though some strict OOP enthusiasts insist that *everything* in programming must be an object.

Also, Visual Basic programmers almost never use `Sub Main`. Instead, we use the `Form1_Load` event as the startup object. For one thing, this event is the *default* startup object in Visual Basic. Programmers using languages that are not form-based, such as C, employ `Sub Main` as their entry point when a program begins execution. There is no default `Form1` in C, so C programmers are accustomed to starting their programs in `Sub Main` (which is where they put initialization code, and also where they test short code samples).

To make the many "VB" code samples in Help work, a VB.NET programmer must right-click the project name in Solution Explorer, choose Properties, then manually adjust the startup object in a Property Pages dialog box.

Another giveaway that this isn't true VB is the line `Imports System`. VB.NET programmers know that by default the `System` namespace is always available and need not be imported.

Also, VB.NET programmers do not use the brace `{ }` symbols to fill in fields when displaying results, nor do we use the word `End` as a variable name, requiring that it be enclosed in brackets `[]`.

For example:

```
Open "'C:\Test.Txt" As 5
```

In VB.NET you use a C-like "stream" (an incoming or outgoing data flow) technique for reading or writing to a file. Streams are more flexible than the traditional VB approach, and streams can be used not only with disk files but with other data stores, including data incoming from the Internet—and indeed, from other streams.

Reading a File

Here's how to read a disk file:

```
Imports System.IO
Private Sub Button1_Click_1(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    Dim a As String
    Dim sr As New StreamReader("e:\test.txt")
    a = sr.ReadLine
    a += sr.ReadLine
    sr.Close()
End Sub
```

However, for a more flexible approach, type Listing 2.7 into the Button's Click event.

LISTING 2.7: READING DISK FILES VIA STREAMS

```
Private Sub Button1_Click_1(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click

    Dim strFileName As String = TextBox1.Text

    If (strFileName.Length < 1) Then
        msgbox( "Please enter a filename in the TextBox")
        Exit Sub
    End If

    Dim objFilename As FileStream = New FileStream(strFileName, F
    FileAccess.Read, FileShare.Read)

    Dim objFileRead As StreamReader = New StreamReader(objFilename)

    While (objFileRead.Peek() > -1)
        textbox1.Text += objFileRead.ReadLine()
    End While
End Sub
```

In short, the DataGrid control is made up of DataTable objects that correspond to the tables specified in the queries you specified in the configuration of the DataAdapter. Each table in the DataSet is represented by a DataTable object and each DataTable contains as many rows as you have specified in the corresponding query. You can also create relations between the DataSet's DataTables, as we did earlier. The DataSet enforces the relations as well (unless you specify that relations shouldn't be enforced). If you attempt to edit a product by specifying a CategoryID value that doesn't exist in the Categories table, you'll see a message box explaining that the CategoryID value you specified is invalid:

```
ForeignKeyConstraint CategoriesProducts requires the child key values  
(22) to exist in the parent table. Do you want to correct the value?
```

You can either correct the value or click No to restore the original value. The proper course of action, of course, is to add a new item to the Categories DataTable and then use it with the Products table.

UPDATING THE DATABASE

Let's add some code to submit the changes made at the client back to the server. It's likely that some rows may fail to update the database. For example, the UnitPrice field in the Products table can't be negative. This is a constraint enforced by the database, but not by the DataSet. The DataSet can enforce only referential and non-null constraints, but not arbitrary constraints that exist in the database. It's possible to specify a negative price on the DataGrid control, but the corresponding row will be rejected by the database when you attempt to insert the row. To submit the changes to the database, add one more button on the form, set its caption to "Update Database," and enter the statements shown in Listing 15.3 into its Click event handler.

LISTING 15.3: POPULATING A DATASET WITH MULTIPLE RELATED TABLES

```
Private Sub btnUpdate_Click(ByVal sender As System.Object, _  
                             ByVal e As System.EventArgs) _  
                             Handles btnUpdate.Click  
    DACategories.Update(Products1)  
    DASuppliers.Update(Products1)  
    DAProducts.Update(Products1)  
End Sub
```

If the data on the DataGrid contains no errors, the changes will be submitted to the database. The Update method of the DataAdapter object submits all the changes in the tables of the specified DataSet to the database, using the commands generated by the DataSet configuration wizard. The DACategories DataAdapter, for example, knows how to submit the changes made to the Categories table, the DASuppliers DataAdapter knows how to submit the changes made to the Suppliers table, and the DAProducts DataAdapter knows how to update the Products table in the database. When a DataAdapter runs into an error, it terminates the update process and doesn't attempt to update additional rows. To change this behavior and force the DataAdapters to update as many rows as possible, select each DataAdapter on the form and set its ContinueUpdateOnError property to True.

Let's see how the DataGrid handles update errors. Edit a row or two in the Products table and set their UnitPrice column to a negative value. While you're editing the Products table, delete a product name. Actually, select it with the mouse and cut it (you'll need to restore its value later). Then click the Update Database button to submit the changes to the database. Some rows will fail to update the underlying tables. A red icon with an exclamation mark will appear in front of each row that failed to update the database. This is an instance of the ErrorProvider control and you can hover the pointer over it to see a description of the error that prevented the DataAdapter from submitting the edited row to the database, as shown in Figure 15.7.

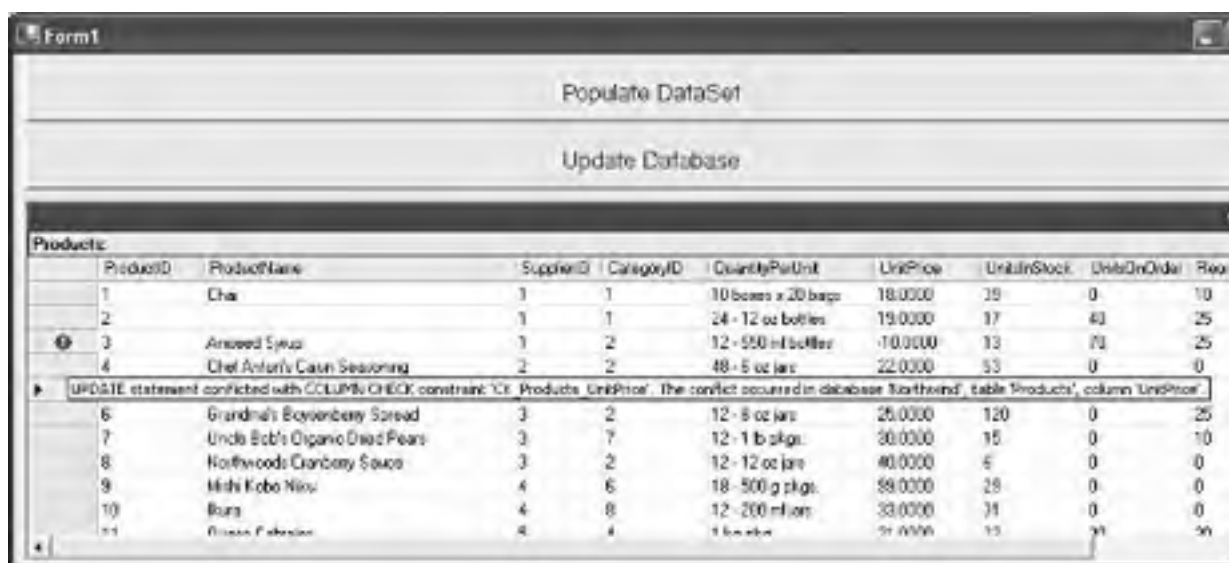


FIGURE 15.7 Rows in error are marked automatically on the DataGrid.

Notice that the row with the empty product name is not in error, even though the Products table has a constraint to that effect. The ProductName column is not nullable. By deleting the name of the product we've set it to an empty string, which is not a Null value. To set a field to Null you must press Ctrl+0 (zero). The empty string is a valid value for a string column, so the database accepted it. We usually insert code to handle Null values in our applications. For the case of the ProductName field, a product without a name is clearly an invalid condition. We might as well convert all the empty product names to Null values before submitting them to the database, or display a warning to the user. You can further edit the rows on the DataGrid control until all edited rows have been submitted to the database successfully. To verify that all changes were submitted to the database, click the Populate DataSet button to read the tables from the database again and see that the new values have taken effect.

Formatting the DataGrid Control

The default appearance of the DataGrid control is rather dull, but you can adjust it through the numerous properties it exposes. Select the DataGrid control on the design surface and locate its TableStyles property in the Property Browser. This property is a collection, made up of

TableStyle objects, and there's one TableStyle object for each of the tables in the DataSet. Each TableStyle provides properties for controlling the appearance of the corresponding table (the header's font, the color of the grid lines, the default width of each column, whether the rows can be sorted or not, and so on). Each TableStyle has a GridColumnStyle property, which is also a collection, and there's a GridColumnStyle object for each column of the table. Here you can define the column's caption (the default caption being the column's name), its width, and the format for its values. You can experiment with the DataGrid control's TableStyles collection to design a better-looking grid.

[Team Fly](#)

 Previous

Next 

The DataGrid control is a very convenient control for displaying related tables, but it's not the primary tool for building data-driven applications. If you search the Internet you'll find hundreds of articles on customizing the DataGrid control. In our view, the DataGrid control is a programmer's tool, but not the ultimate tool for building user-friendly applications. A major limitation of the DataGrid control is that you can't access its cells with a row/column convention. Instead, you must access the underlying DataSet and change fields in the DataSet, and the changes will be automatically reflected on the control's contents. Another limitation is that you can't use lookup columns with it. For example, if you bind the Products tables to a DataGrid control, the CategoryID item will be an integer; there's no simple method of displaying a ComboBox with the category names and bind the ComboBox control to the category ID of the current row. If you've worked a little with this control, you may also have discovered that it doesn't capture keystrokes and mouse clicks. There are workarounds for all the shortcomings mentioned here, but once you start writing code to deal with the limitations of a given control, the benefits of a RAD system evaporate. In Chapter 18 you'll learn how to use the regular Windows controls to build much more functional interfaces. For this, you'll have to understand how to program the ADO.NET objects.

In this chapter we use the DataGrid a lot, because it's very convenient and it allows us to quickly explore DataSets without having to write elaborate interfaces. In Chapter 18 you'll see several examples of practical user interfaces.

The wizard has created a class that establishes a connection to the database through a Connection object and executes commands against the database through a Command object. In the following section, we'll explore how the wizard manipulates these two objects and how you can program them from within your application.

The Connection Class

To establish a connection to a database you must first set the Connection object's ConnectionString property and then call its Open method. The connection must be closed as soon as the operation completes (reading or writing data to the database). To close a connection, just call its Close method. Never open a connection to the database in the Form's Load event handler and keep it open for the application's lifetime. This approach will work with small projects, but such an application will never scale out. Maintaining a large number of open connections is a very expensive operation, especially with Web applications. .NET maintains a pool of connections and reuses them to accommodate all the clients with the least number of connections. That's why you must close your connections and return them to the pool as soon as possible.

The ConnectionString property provides all the information necessary to establish a connection to the data source: the name of the server machine on which the DBMS resides, the name of the database, and authorization information, among other things. SQL Server will not honor a connection request unless the user who requested the connection has the appropriate privileges. There are two techniques to authorize a user: pass a user ID and password to SQL Server or ask Windows to authenticate the workstation that initiated the request. The second is the safest authentication technique, because you don't have to store passwords anywhere. If a

user is authenticated successfully by Windows itself, then SQL Server will honor the connection.

[Team Fly](#)

 Previous

Next 

A common scenario is to create an account on SQL Server for all the users of an application and pass this account's user ID and password along with the connection request. The following statement shows a `ConnectionString` that authenticates the user, `DBUser`:

```
Dim CN As SqlClient.SqlConnection
CN.ConnectionString = _
    'initial catalog=Northwind;persist" & _
    "security info=False; & _
    "data source = YOUR Server; & _
    "user id=DBUser;password=YOUR_PASSWORD; & _
"workstation id=YOUR WORKSTATION;packet size=4096"
```

Many developers use the `sa` account without a password to connect to SQL Server, especially when they test their applications on the same workstation on which the DBMS is installed. You should at least supply a password to the `sa` account, if you're still using it.

To let Windows authenticate a user, omit the User ID and Password options of the Connection string and add the option:

```
Integrated Security = SSPI
```

The following statements establish a connection to the Northwind database using the built-in Windows authentication:

```
Dim CN As SqlClient.SqlConnection
CN.ConnectionString = _
    "Integrated Security=SSPI;Initial Catalog=Northwind;" & _
    "Data Source=localhost" & _
    "workstation id=YOUR WORKSTATION;packet size=4096"
```

The sample applications in this book use Windows authentication. To change the authentication mode, set a user ID and password that will work on your machine.

The sample connection string refers to the database on the same computer, and the setting of the Data Source attribute is `localhost` (you can also use the period to reference the workstation on which the application is running). To connect to an instance of SQL Server running on another machine on the same network, set the Data Source attribute to the name of this machine. The advantage of using Windows authentication is that you don't have to store sensitive information (passwords) in your code.

Instead of writing the connection strings yourself (and testing them), you can let the Visual Studio IDE create them for you. Just open the Server Explorer and drop the database to which you want to connect onto a form. Depending on the database you selected, an `SqlConnection` or an `OleDbConnection` object will be added to the project. Select this object on the design surface and look up its `ConnectionString` setting in the Properties window. Connection objects are created automatically by the IDE and added to your form every time you drop an item from a specific database (a table, view, or stored procedure) from the Server Explorer onto the design surface. Dropping additional items from the same database creates new `DataAdapter` objects, which will reuse the same connection.

[Team Fly](#)

 Previous

Next 

The DataAdapter Class

The DataAdapter is an object that knows how to access a database and perform specific tasks against it. The tasks we perform against a database with DataAdapters are the following:

- ◆ Retrieval of selected rows
- ◆ Insertion of new rows
- ◆ Deletion of existing rows
- ◆ Update of existing rows

Each of these four tasks corresponds to a SQL command: the SELECT, INSERT, DELETE, and UPDATE commands, respectively. To retrieve data through a DataAdapter, you must call its Fill method, which accepts as arguments an instance of a DataSet and the name of the DataTable, and fills the specified DataTable in the DataSet with the results of a query. You can't access the results of the query directly; the DataAdapter can only be used to populate DataSets. You have already seen how to fill a DataTable with the help of the DataAdapter. Notice that the DataAdapter doesn't empty the DataTable before filling it; instead, it attempts to append the new data to the existing one. If this isn't what you want, you must clear the DataTable, or the entire DataSet, with the Clear method. In most cases, the rows we retrieve from the database contain a primary key and, if you attempt to insert duplicate keys in the same DataTable, a runtime exception will be thrown. However, it's possible to add two distinct sets of rows from the same table by calling the Fill method twice.

The other basic method of the DataAdapter is the Update method, which submits the changes made to the DataSet back to the database. Although you specified only the SELECT statement to retrieve the desired data from the database, the wizard generated SQL statements for updating existing rows, inserting new ones and deleting existing ones. These statements are executed by the DataAdapter for all the rows in the DataTable that must update the underlying table. The simplest form of the Update method accepts no arguments and submits all the changes made to its DataTable to the database. There are overloaded methods that allow you to specify a subset of the rows you want to submit to the underlying table in the database.

THE DATAADAPTER'S COMMANDS

The DataAdapter provides commands for performing the basic actions against the database; you can access them through the SelectCommand, InsertCommand, DeleteCommand, and UpdateCommand properties. These properties are Command objects, and you can configure them from within your code. The wizard that takes you through the steps of configuring a DataAdapter sets up these objects behind your back. Let's see what the wizard has done for us behind the scenes in the NWProducts project.

Open the NWProducts project in design mode, select the `DASuppliers` DataAdapter on the design surface, and locate the SelectCommand property in the Property Browser and expand it. One of its attributes is the CommandText; its setting is the following SQL statement:

```
SELECT SupplierID, CompanyName, ContactName,  
       ContactTitle, Phone, Fax  
FROM   Suppliers
```

[Team Fly](#)

 Previous

Next 

The DataAdapter selects by default the entire table for us. While this is quite acceptable with a small database such as Northwind, in a production database you can't select entire tables, because they may contain thousands of rows and you can't afford to move them all to the client every time the user wants to view a phone number or an e-mail address. You can change the SELECT statement as you configure the DataAdapter by adding a WHERE clause to limit the number of selected rows.

Now locate the InsertCommand property and look up its CommandText attribute. This is the SQL statement that will insert new rows into the Customers table. The syntax for the INSERT command generated by the wizard is:

```
INSERT INTO Suppliers (SupplierID, CompanyName, ContactName,
    ContactTitle, Phone, Fax)
    VALUES (@SupplierID, @CompanyName, @ContactName,
        @ContactTitle, @Phone, @Fax);
SELECT      SupplierID, CompanyName, ContactName,
            Phone, Fax
FROM        Suppliers
WHERE       (SupplierID = @SupplierID)
```

There are two SQL statements here. The first one inserts a new row into the Customers table and the second one selects the newly inserted row and returns it to the application. The INSERT statement contains a bunch of parameters, which represent the values of the fields of the new row. These parameters will be assigned their values automatically by the DataAdapter when you call the DataAdapter's Update method, which submits the changes made to the selected row(s) at the client to the server. It knows which rows should be inserted (the new rows added to the DataSet) and which rows should be deleted and updated. To insert new rows, the DataAdapter executes the command of the InsertCommand object, passing the values of the new row as arguments.

The DeleteCommand object's command is a little more complicated:

```
DELETE FROM Suppliers
WHERE (SupplierID = @Original_SupplierID) AND
      (CompanyName = @Original_CompanyName) AND
      (ContactName = @Original_ContactName OR
        @Original_ContactName IS NULL AND ContactName IS NULL) AND
      (ContactTitle = @Original_ContactTitle OR
        @Original_ContactTitle IS NULL AND ContactTitle IS NULL) AND
      (Fax = @Original_Fax OR
        @Original_Fax IS NULL AND Fax IS NULL) AND
      (Phone = @Original_Phone OR
        @Original_Phone IS NULL AND Phone IS NULL) AND
```

This statement deletes a row whose columns match the columns of the row that was read from the database. This is how the DataAdapter handles *concurrency*. If another user has edited the same row since the application read it from the database, the row won't be deleted. To implement concurrency, the DELETE statement generated by the DataAdapter compares each column of the original version of the row (the version of the row that was read from the database) to the corresponding columns of the same row in the database. If even a single column's value is different, the DELETE

statement will fail. The default behavior of the DataAdapter doesn't allow you to delete a row if it has been edited in the meantime. The original values of the row's columns are stored in the variables starting with the @Original prefix. As you will see in the following section, the DataSet maintains the original values of the rows read through the DataAdapter from the database.

The UPDATE statement makes use of the original values of the row's columns and updates a row only if it hasn't been edited since it was read. Here's the UPDATE statement generated by the DataAdapter configuration wizard for the Suppliers table:

```
UPDATE Suppliers
SET SupplierID = @SupplierID, CompanyName = @CompanyName,
    ContactName = @ContactName, ContactTitle = @ContactTitle,
    Phone = @Phone, Fax = @Fax
WHERE (SupplierID = @Original_SupplierID) AND
    (CompanyName = @Original_CompanyName) AND
    (ContactName = @Original_ContactName OR
        @Original_ContactName IS NULL
        AND ContactName IS NULL) AND
    (ContactTitle = @Original_ContactTitle OR
        @Original_ContactTitle IS NULL
        AND ContactTitle IS NULL) AND
    (Fax = @Original_Fax OR
        @Original_Fax IS NULL AND Fax IS NULL) AND
    (Phone = @Original_Phone OR
        @Original_Phone IS NULL AND Phone IS NULL) AND
SELECT SupplierID, CompanyName, ContactName, Phone, Fax
FROM Suppliers
WHERE (SupplierID = @SupplierID)
```

Although substantially longer than the DELETE statement, the UPDATE command is based on same principle. If the row has been edited since it was read, the update operation will fail.

Notice how the DataAdapter handles Null values. Comparisons with Null values are invalid in T-SQL. In other words, two columns that are Null are not equal, because Null means that the column has no value and therefore no comparison is made. The T-SQL code generated by the DataAdapter compares the two columns:

```
Fax = @Original_Fax
```

as well as both columns to Null:

```
@Original_Fax IS NULL AND Fax IS NULL
```

If either comparison returns True, then the two fields are the same. If the current and original versions of a column have the same value, or they're both Null, the statement proceeds. If both comparisons fail, then the two values are different and the statement will fail.

The two preceding statements are fairly complicated, but here's how the DataAdapter works. First, it generates a SELECT statement to retrieve all the columns of all the rows in the table (that is, the entire table). You can provide your own SQL statement to limit the number of rows and/or columns retrieved from the database. Then it generates the corresponding INSERT, DELETE, and

UPDATE statements. The format of the action queries is always the same. You can modify the default behavior of the wizard that generates the statements for the DataAdapter. That is the topic of the next section.

HANDLING CONCURRENCY ISSUES WITH THE DATAADAPTER

You can change the default behavior of the DataAdapter, either with visual tools or from within your code. To control the generation of the SQL statements, select the DataAdapter on the designer and click the Configure Data Adapter link below the Properties window. Click the Next button a few times until you see the Generate The SQL Statements window, which is shown in Figure 15.2. Click the Advanced Options button on this window to see the Advanced SQL Generation Options window, where you can customize how the DataAdapter generates the SQL statements. The options on this window are shown in Figure 15.8.



FIGURE 15.8 Controlling the generation of the DataAdapter's commands

The first option, Generate Insert, Update And Delete Statements, determines whether the DataAdapter will generate action queries. If you're selecting data to populate a control and the application won't allow users to edit the data, clear this option and the DataAdapter will generate only the SELECT command. If you turn off this option, the other two options on the window will be disabled.

The second option, Use Optimistic Concurrency, determines the WHERE clause of the statements. By default, the DataAdapter uses optimistic concurrency. Optimistic concurrency is based on the assumption that it's highly unlikely that two users will be working on the same row at the same time. When this happens, one of the two users won't be allowed to change, or delete, a row that has been already modified by another user. Optimistic concurrency works well with most applications, because the fundamental assumption is mostly true. There are exceptions from time to time, and when this happens the update operation fails. The user must

read the row again from the database, edit it, and submit the changes to the database.

Turning off optimistic concurrency will make the DELETE and UPDATE statements generated by the wizard simpler, because not all columns are taken into consideration in the WHERE clause of the corresponding statements. As you can see, the statement uses only the table's primary key to select the appropriate row. Moreover, it updates it no matter what the values of the other columns are. This scenario is known as last-write-wins (it's also known as *destructive* concurrency): the last user

to edit a row overwrites the changes made by all other users. This scenario also works well in many applications. When two users edit a row of the Customers table, it's very likely that they both edit the same column (change the customer's phone number, for example). However, it can lead to situations that may frustrate users. If a user edits a product's name and another one edits the same product's price, only one of the changes will take effect. The user who edited the product's name will read back the same row a few moments later and find out that the product's name didn't change but the price is different. In a table with thousands of products it's quite unlikely that two users will edit the same row at the same time. With some types of applications (such as banking, or seat reservation applications), this scenario shouldn't be used.

Another factor that should be taken into consideration is how often the client updates the database. If your application downloads an entire table to the client, allows users to edit the data locally, and submits the changes to the database a few hours later, there's a good chance that some of its rows have been edited by other users in the meantime. A different application that retrieves a small set of the table to the client and submits the changes to the database as soon as they occur is much less likely to run into concurrency problems.

Turning off optimistic concurrency doesn't enable pessimistic concurrency, which is the second technique to deal with concurrency issues. With pessimistic concurrency, we avoid the concurrency problem by making sure that no two users can access the same row. To implement pessimistic concurrency we lock the row(s) as soon as we read them, so that no other users can access them. This is a very crude method of handling concurrency issues and rarely used. If a user opens a row and goes to a meeting without closing the application, the row may be locked for hours. No other application can read it, reports may skip this row (and therefore the totals will be incorrect), and awkward situations can result. It's even possible that an error in the application doesn't unlock rows properly and the number of locked rows increases with time.

The DataAdapter doesn't support pessimistic concurrency. However, it is possible to implement optimistic concurrency with transactions. You can start a transaction, open the row, and close the transaction after the row is written back to the database. Keeping many transactions open for long periods of time is a sure way to deteriorate SQL Server's performance and write an application that can't be scaled. As far as ADO.NET is concerned, pessimistic concurrency is out of the question. Transactions are discussed later in this chapter, but the idea is to lock one or more rows to perform an operation that involves these rows. If any of the operations fail, then all the operations are rolled back and no changes are made to the database. The transaction should be completed in the shortest possible time span, so that the rows will be freed soon and made available to other users.

The last option on the Advanced SQL Generation Options window is the generation of the SELECT statement that retrieves the edited row from the database after an insert or update operation. When this option is enabled, the INSERT and UPDATE statements retrieve the inserted or edited row from the database. The new version of the row is downloaded to the client, where it replaces the original row. We usually don't turn off this option with optimistic concurrency.

The last property of the DataAdapter you should set is the ContinueUpdateOnError property. This is a very important property and it determines what the DataAdapter does when it runs into an update error. DataAdapters can be used to submit multiple updates (insertions, updates, and deletions) to the database. If one of the operations fails, the DataAdapter will not attempt to submit any more changes to the database, because the default setting for this property is False. If you want to perform as many of the updates as possible and then handle all the errors, set the ContinueUpdateOnError property to True.

```
End While

objFileRead.Close()
objFilename.Close()
```

```
End Sub
```

Note the End While command, which replaces the VB6 Wend command.

How Do You Know You're at the End?

When reading from a file, you have to know when you've reached the end. In the previous example, you used the following code:

```
While (objFileRead.Peek() > -1)
```

Alternatively, you can use this:

```
While (objFileRead.PeekChar() <> -1)
```

Or you can use yet another technique for reading (or writing, using the FilePut command). In this case, you test for the end of file with the venerable EOF property, End Of File. Also, you use the FileGet command to read from a file; in this case, you are reading individual characters. Start a new project, and put a TextBox on the form. Now type Listing 2.8.

LISTING 2.8: USING FILEGET TO READ A FILE

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

    Dim objN As New ReadBytes()

    TextBox1.Text = objN.ReadAFile

End Sub

Public Class ReadBytes

    Private strRead As String

    Public Function ReadAFile()
        strRead = ""

        Dim chrHolder As Char
        Dim filenumber As Int16

        filenumber = FreeFile() ' whatever filenumber isn' t already
```


USING COMMANDS WITH PARAMETERS

Most practical commands accept parameters and commonly return values to the VB application through parameters. Let's consider a simple SELECT statement that retrieves the customers from a specific country:

```
SELECT * FROM Customers WHERE Country = @country
```

@country is a variable name and you @country is a variable name and you must set its value before executing the preceding statement. To submit this command to the database, you must first create an SqlConnection object as already explained, the CN object. Then create an SqlCommand object, the CMD object, that uses this connection and set its CommandText and CommandType properties:

```
Dim CMD As New SqlCommand  
CMD.Connection = CN  
CMD.CommandText = 'SELECT * FROM Customers WHERE Country = @country'
```

The command's parameters are members of the Parameters property, which is made up of Parameter objects. The Parameter object has a number of properties you can use to specify the parameter. At the very least you must set the parameter's type, using either the DbType property or the SqlDbType property. The settings for these two properties are the basic data types (or the data types supported by SQL Server) and you'll see them on a list as you type. The DbType property, for example, can be set to String, which is very convenient for VB developers. The SqlDbType, on the other hand, doesn't have a String setting. You must specify one of the data types supported by SQL Server (such as varchar or text). If the parameter's type is String, you must also set the maximum size of the string with the Size property. If the parameter is a numeric value, you can set the Scale and Precision properties. The second property of the Parameter object you must set is the Direction property, which determines whether this is an input parameter (it's passed by the application to the statement or stored procedure), an output parameter (it's passed by the statement or stored procedure to the application), or both. Finally, you can set the parameter's value through the Value property.

To specify a value for the @country variable in the preceding example, you must set up a Parameter object with the following statements:

```
Dim P As New SqlClient.SqlParameter  
P.ParameterName = "@country"  
P.DbType = DbType.String  
P.Size = 25  
P.Direction = ParameterDirection.Input  
P.Value = "Germany"  
CMD.Parameters.Add(P)
```

The last statement attaches the parameter to the Command object's Parameters collection. You can create additional parameters, set their properties, and append them to the Parameters collection. Then you can call any of the Command object's Execute methods. The following VB code will execute an UPDATE statement against the Products table to increase the price of all products from a specific supplier by \$2. This is the code behind the Execute SQL Statement button of the DataSet Basics project, as shown in Listing 15.4.

LISTING 15.4: EXECUTING A SQL STATEMENT AGAINST THE DATABASE

```
Private Sub btnExecuteSQL_Click(ByVal sender As System.Object, _  
                                ByVal e As System.EventArgs) _  
                                Handles btnExecuteSQL.Click  
  
    Dim supplierID As Integer = InputBox("'Please enter a supplier ID  
Me.Refresh()  
If supplierID <= 0 Then Exit Sub  
Dim CMD As New SqlClient.SqlCommand  
CMD.Connection = SqlConnection1  
CMD.CommandText = "UPDATE Products" &  
                   "SET UnitPrice = UnitPrice + 2" &  
                   "WHERE Products.SupplierID = @supplierID"  
CMD.CommandType = CommandType.Text  
Dim P0 As New SqlClient.SqlParameter  
P0.ParameterName = "@supplierID"  
P0.DbType = DbType.Int32  
P0.Direction = ParameterDirection.Input  
P0.Value = supplierID  
CMD.Parameters.Add(P0)  
Dim rows As Integer  
SqlConnection1.Open()  
rows = CMD.ExecuteNonQuery  
SqlConnection1.Close()  
MsgBox("The query affected " & rows.ToString & " rows")  
End Sub
```

To execute a stored procedure, use similar statements. Let's code the previous UPDATE statement as a stored procedure and attach it to the database. The ChangeSupplierPrices stored procedure is shown next:

```
CREATE PROCEDURE ChangeSupplierPrices  
@supplierID int  
AS  
UPDATE Products  
SET UnitPrice = UnitPrice + 2  
WHERE Products.SupplierID = @supplierID
```

The statements in Listing 15.5 will execute the ChangeSupplierPrices stored procedure against the database. They're almost identical to the statements of Listing 15.4 that executed the equivalent SQL statement against the database, with the exception of the statement that specifies the type of the command.

LISTING 15.5: EXECUTING A STORED PROCEDURE AGAINST THE DATABASE

```
Private Sub btnExecuteSP_Click(ByVal sender As System.Object, _  
                               ByVal e As System.EventArgs) _  
    Handles btnExecuteSP.Click  
    Dim supplierID As Integer = InputBox("'Please enter a supplier ID  
Me.Refresh()  
If supplierID <= 0 Then Exit Sub  
Dim CMD As New SqlClient.SqlCommand  
CMD.Connection = SqlConnection1  
CMD.CommandText = "ChangeSupplierPrices"  
CMD.CommandType = CommandType.StoredProcedure  
Dim P0 As New SqlClient.SqlParameter  
P0.ParameterName = "@supplierID"  
P0.DbType = DbType.Int32  
P0.Direction = ParameterDirection.Input  
P0.Value = supplierID  
CMD.Parameters.Add(P0)  
SqlConnection1.Open()  
Dim rows As Integer  
rows = CMD.ExecuteNonQuery()  
SqlConnection1.Close()  
MsgBox("The stored procedure affected " & rows.ToString & " rows"  
End Sub
```

You can also execute commands and stored procedures to select rows from the database, but you can't use the result of a query executed through a Command object to populate a DataSet. There are basically two methods of storing data at the client. One of them is to read the data returned by a command through a DataReader object and store them to a data structure, such as an ArrayList or an array. You can also use the data to populate a Windows control. For example, you can map each row returned by the command to a ListViewItem and populate a ListView control. The second method is to populate a DataSet with the selected rows through a DataAdapter. The DataSet is a data store that resides at the client and has the structure of a database. Data are stored in tables and the same relations that exist in the database can be applied to the DataSet. As you will see, the DataSet is an excellent mechanism for storing data at the client, as well as for passing data between layers. In the following section we'll discuss the DataReader class and how to read the result of a query through the DataReader class. Then we'll explore the architecture of the DataSet class.

The third method for executing a command is the Command object's ExecuteScalar method, which executes the command and returns a single value. This is the value of the first column of the first row in the resultset. What this basically means is that the ExecuteScalar method returns a single value, which you must return from within your SQL statement or stored procedure with a SELECT statement. The following statements retrieve the average number of products per supplier, as shown in Listing 15.6.

LISTING 15.6: THE PRODECTPERSUPPLIER STORED PROCEDURE

```
DECLARE @productCount int
SELECT @productCount = COUNT(Products.ProductID)
FROM Products
WHERE SupplierID IS NOT NULL

DECLARE @supplierCount int
SELECT @supplierCount = COUNT(Suppliers.SupplierID)
FROM Suppliers
WHERE SupplierID IN (SELECT DISTINCT SupplierID FROM Products)
SELECT cast(@productCount AS float) / Cast(@supplierCount AS float)
```

We're taking into consideration only the products that have a valid SupplierID value and the suppliers that refer to one or more products. You can assign this lengthy SQL statement to the CommandText property of a Command object, execute the command with the ExecuteScalar method, and retrieve the value selected by the last line of the stored procedure. The ExecuteScalar method should be called with the statements shown in Listing 15.7.

LISTING 15.7: EXECUTING A STORED PROCEDURE WITH THE EXECUTESCALAR METHOD

```
Private Sub btnExecuteScalar_Click(ByVal sender As System.Object, _
                                   ByVal e As System.EventArgs) _
    Handles btnExecuteScalar.Click
    Dim CMD As New SqlClient.SqlCommand
    CMD.Connection = SqlConnection1
    CMD.CommandText = "ProductsPerSupplier"
    CMD.CommandType = CommandType.StoredProcedure
    SqlConnection1.Open()
    Dim avgProductNumber As Single
    avgProductNumber = CMD.ExecuteScalar
    SqlConnection1.Close()
    MsgBox("The average number of products per supplier is " & _
           avgProductNumber.ToString)
End Sub
```

USING THE DATAREADER

The DataReader is an object that provides fast, forward-only, read-only access to the data returned by a command. In other words, you can iterate through the selected rows in a forward-only fashion and you can only read data. To update the underlying table(s) in the database, you must set up other commands to execute the appropriate action queries, or stored procedures, against the database. Assuming that the CMD object represents a properly configured SqlCommand object, the following

statements execute the command with the `ExecuteReader` method and iterate through the selected rows. Each row is added to a `ListView` control at the client, as shown in Listing 15.8.

LISTING 15.8: POPULATING A LISTVIEW CONTROL WITH THE DATAREADER

```
Dim SQLRSR As New SqlConnection.SqlDataReader
SQLRDR = CMD.ExecuteReader
Dim LI As ListViewItem
While SQLRDR.Read
    LI = New ListViewItem()
    LI.Text = SQLRDR.GetString(0)
    LI.SubItems.Add(SQLRDR.GetString(1))
    LI.SubItems.Add(SQLRDR.GetString(2))
    ListView1.Items.Add(LI)
End While
```

The `ExecuteReader` method of the `Command` object returns a `DataReader` object (a `SqlDataReader` object for the `SqlCommand` object and an `OleDbDataReader` object for the `OleDbCommand` object). The `DataReader` is positioned in front of the very first row and you must use the `Read` method to move to the next row in the resultset. The `While` loop keeps reading one row at a time with the `Read` method, which will return `False` after reading the last row. In the loop's body we can use one of a variety of methods to read a column's value. Since all three columns in the requested rows are strings, we use the `GetString` method. The `DataReader` method exposes methods for reading all types that SQL Server may return (`GetChar`, `GetChars`, `GetDateTime`, `GetFloat`, `GetGuid`, and so on). You can look up the methods that read data in the drop-down list with the object's methods as you type. Notice that the various columns are identified by an integer, which is their order in the selection list of the `SELECT` statement.

Our sample code adds the columns of the selected rows to a `ListView` control. You could have created a custom class for storing the same data—say, the `CustomerInfo` class. At each iteration you should create a new instance of this class, populate its fields, and add it to a collection. If you only wanted to perform a few calculations with the data, you could insert the appropriate statements in the loop's body and not store the data values anywhere. It's usually simpler to write a SQL statement or stored procedure to perform the calculations at the server and return the result(s) to the client application.

One cool feature of the `DataReader` is that it can process multiple cursors returned by batch queries. If you want to retrieve products and their suppliers, you can execute a batch query such as the following:

```
SELECT * FROM Products
WHERE CategoryID = 5;
SELECT * FROM Suppliers
    INNER JOIN Products
    ON Products.SupplierID = Suppliers.SupplierID;
```



```
252  
253     Public Overloads Function AddCategoriesRow(ByVal CategoryName As String,   
254     ByVal Description As String) As CategoriesRow  
255         Dim rowCategoriesRow As CategoriesRow = CType(Me.NewRow,   
256         CategoriesRow)  
257         rowCategoriesRow.ItemArray = New Object() {Nothing, CategoryName,   
258         Description}  
259         Me.Rows.Add(rowCategoriesRow)  
260         Return rowCategoriesRow  
261     End Function  
262  
263     Public Function FindByCategoryID(ByVal CategoryID As Integer) As   
264     CategoriesRow  
265         Return CType(Me.Rows.Find(New Object() {CategoryID}), CategoriesRow)  
266     End Function  
267  
268     Public Function GetEnumerator() As System.Collections.IEnumerator   
269     Implements System.Collections.IEnumerable.GetEnumerator  
270         Return Me.Rows.GetEnumerator()  
271     End Function  
272  
273     Public Overrides Function Clone() As DataTable  
274         Dim cln As CategoriesDataTable = CType(MyBase.Clone,   
275         CategoriesDataTable)  
276         cln.InstrVars
```

FIGURE 15.9 The DSProducts.vb class provides the functionality of a typed DataSet.

The following statements demonstrate the difference between typed and untyped DataSets. To access the ProductName column of the first row in the Products table in an untyped DataSet, you'd use an expression such as the following:

```
Products1.Products.Rows(0).Item("ProductName")
```

If the Products1 DataSet is typed, you can create an object of the Products.ProductsRow type with the following statement:

```
Dim productRow As Products.ProductsRow = Products1.Products.Rows(0)
```

Then use the productRow variable to access the columns of the corresponding row:

```
productRow.ProductName  
productRow.UnitPrice
```

Accessing the DataSet's Tables

The DataSet is made up of tables, which are represented by the DataTable class. Each DataTable in the DataSet may correspond to a table in the database, or a view. When you execute a query that retrieves fields from multiple tables, all selected columns will end up in a single DataTable of the DataSet. You can select any DataTable in the DataSet by its index, or its name:

```
DS.Tables(0)  
DS.Tables("Customers")
```

With a typed DataSet, table names are exposed as properties of the DataSet and you can access them with an expression such as the following:

```
DS.Customers
```

[Team Fly](#)

 Previous

Next 

Each table contains a number of columns, which you can access through the Columns collection. The Columns collection is made up of DataColumn objects, one DataColumn object for each column in the corresponding table. The Columns collection is the schema of the DataTable object, and the DataColumn class exposes properties that describe a column. ColumnName is the column's name, DataType is the column's type, MaxLength is the maximum size of text columns, and so on. The AutoIncrement property is True for Identity columns and the AllowDBNull property determines whether the column allows Null values. In short, all the properties you can set visually as you design a table are also available to your code through the Columns collection of the DataTable object. You can use the DataColumn class's properties to find out the structure of the table, or create a new table. To add a table to a DataSet, you can create a new DataTable object. Then create a DataColumn object for each column, set its properties, and add the DataColumn objects to the DataTable's Columns collection. Finally, add the DataTable to the DataSet. The process is described in detail in the online documentation and we won't repeat it here.

Working with Rows

As far as data are concerned, each DataTable is made up of DataRow objects. All DataRow objects of a DataTable have the same structure and can be accessed through an index, which is the row's order in the table. To access the rows of the Customers table, use an expression such as the following:

```
DS.Customers.Rows(iRow)
```

where `iRow` is an integer value from zero (the first row in the table) up to `DS.Customers.Rows.Count - 1` (the last row in the table). To access the individual fields of a DataRow object, use the Item property. This property returns the value of a column in the current row either by its index or by its name:

```
DS.Customers.Rows(0).Item(0)
```

or

```
DS.Customers.Rows(0).Item("CustomerID")
```

To access a row's columns by name, create a DataRow object of the type that corresponds to the rows of a specific table. The following statements create a typed DataRow object to reference a specific product and then retrieve the row's value as properties of the DataRow object:

```
Dim prod As DSProducts.ProductsRow  
prod = DsProducts1.Products.Rows(0)  
MsgBox("The price of " & prod.ProductName & _  
       " is " & prod.UnitPrice.ToString)
```

To iterate through the rows of a DataSet, you can set up a `For...Next` loop such as the following:

```
Dim iRow As Integer
For iRow = 0 To DSProducts1.Products.Rows.Count -1
    ' process row: DSProducts.Products.Rows(iRow)
Next
```

[Team Fly](#)

 Previous

Next 

If you're using a typed DataSet, you can create a DataRow object that represents the rows of a specific table and then set up a For Each...Next loop such as the following:

```
Dim prod As DSProducts.ProductRow
For Each prodRow In DSProducts1.Products.Rows
    ' process prodRow row:
    ' prodRow.ProductName, prodRow.UnitPrice, and so on
Next
```

To edit a specific row, simply assign new values to its columns. To change the value of the ContactName column of a specific row in a DataTable that holds the customers of the Northwind database, use a statement such as the following:

```
DS.Customers(3).Item("ContactName") = "new contact name"
```

The new values are usually entered by a user on the appropriate interface, and in your code you'll most likely assign a control's property to a row's column with statements such as the following:

```
If txtName.Text.Trim <>"" Then
    DS.Customers(3).Item("ContactName") = txtName.Text
Else
    DS.Customers(3).Item("ContactName") = DBNull.Value
End If
```

The code segment assumes that when the user doesn't supply a value for a column, this column is set to Null (if the column is Nullable, of course). If the control contains a value, this value is assigned to the ContactName column of the fourth row in the Customers DataTable of the DS DataSet.

Handling Null Values

A very important (and quite often tricky) issue in coding data-driven applications is the handling of Null values. Null values are special, in the sense that you can't assign them to control properties, or use them in other expressions. Every expression that involves Null values will throw a runtime exception. The DataRow object provides the IsNull method, which returns True if the column specified by its argument is a Null value:

```
If customerRow.IsNull("ContactName") Then
    ' handle Null value
Else
    ' process value
End If
```

In a typed DataSet, DataRow objects provide a separate method to determine whether a specific column has a Null value. If the customerRow DataRow belongs to a typed DataSet, you can use the IsContactNameNull method instead:

```
If customerRow.IsContactNameNull Then  
    ' handle Null value for the ContactName  
Else  
    ' process value: customerRow.ContactName  
End If
```

[Team Fly](#)

 Previous

Next 

If you need to map Null columns to specific values, you can do so with the IsNull function of T-SQL, as you retrieve the data from the database. In many applications, we want to display an empty string or a zero value in the place of a Null field. We can avoid all the comparisons in our code by retrieving the corresponding field with the IsNull function in our T-SQL statement. Where the column name would appear in the SELECT statement, use an expression such as the following:

```
IsNull(customerBalance, 0.00)
```

If the customerBalance column is Null for a specific row, SQL Server will return the numeric value zero. This value can be used in reports, or in other calculations in your code. Notice that the customer's balance shouldn't be Null. A customer always has a balance, even if it's zero. When a product's price is Null it means that we don't know the price of the product (and therefore can't sell it). In this case, a Null value can't be substituted with a zero value. You must always handle Null columns in your code carefully; how you'll handle them depends on the nature of the data they represent.

Adding and Deleting Rows

To add a new row to a DataTable, you must first create a DataRow object, set its column values, then call the Add method of the Rows collection of the DataTable to which the new row belongs, passing the new row as argument. If the DS DataSet contains the Customers DataTable, the following statements will add a new row for the Customers table.

```
Dim newRow As New DataRow = dataTable.NewRow  
newRow.Item("CompanyName") = "new company name"  
newRow.Item("CustomerName") = "new customer name"  
newRow.Item("ContactName") = "new contact name"  
DS.Customers.Rows.Add(newRow)
```

Notice that we need not set the CustomerID column. This is an Identity column and it's assigned a new value automatically by the DataSet. Of course, when the row is submitted to the database, the ID assigned to the new customer by the DataSet may already be taken. SQL Server will assign a new unique value to this column when it inserts it into the table. We'll have more to say about inserting and updating columns with Identity values. If you're designing a new database, use GUIDs (globally unique identifiers) instead of identity values. A GUID can be created at the client and it is unique. The same GUID that will be generated by the client will also be inserted in the table, when the row is committed. To create GUIDs, call the NewGuid method of the Guid class:

```
newRow.Item("CustomerID") = Guid.NewGuid
```

Finally, to delete a row you can either remove it from the Rows collection with the Remove or the RemoveAt methods of the Rows collection, or call the Delete method of the DataRow object that represents the row. The Remove method accepts as argument a DataRow object and removes it from the collection:

```
Dim customerRow As DS.CustomerRow  
customerRow = DS.Customers.Rows(2)  
DS.Customers.Remove(customerRow)
```

[Team Fly](#)

 Previous

Next 

```
FileOpen(filename, 'C:\test.txt', OpenMode.Binary)

Do While Not EOF(filename)
    FileGet(filename, chrHolder)
    strRead = strRead & chrHolder
Loop
FileClose(1)

Return strRead

End Function
```

```
End Class
```

Listing 2.9 shows one more way to read from a file. This one does not require a loop.

LISTING 2.9: USING THE READTOEND METHOD TO READ A FILE

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

    Dim objFilename As FileStream = New FileStream("C:\test.txt"
    FileMode.Open, FileAccess.Read, FileShare.Read)
    Dim strRdr As New StreamReader(objFilename)

    TextBox1.Text = strRdr.ReadToEnd()
    TextBox1.SelectionLength = 0 'turn off the selection

End Sub
```

This one relies on the ReadToEnd method of the StreamReader object. The one kink is that the text that is placed into the TextBox is selected (white text on black background). So, to deselect it, you set the SelectionLength property to zero.

Writing to a File

The code that writes to a file is similar to the previous file-reading examples. The simplest approach is as follows:

```
Dim sw As New StreamWriter("test.txt")
sw.WriteLine("My example line.")
sw.WriteLine("A second line.")
sw.Close
```

For a more flexible, advanced example, type Listing 2.10.

DELETING VERSUS REMOVING ROWS

Deleted rows are not always removed from the DataSet, because the DataSet maintains its state. If the row you've deleted exists in the underlying table (in other words, if it's a row that was read into the DataSet when you filled it), the row will be marked as deleted, but it will not be removed from the DataSet. If it's a row that was added to the DataSet after it was read from the database, the deleted row is actually removed from the Rows collection. You can physically remove deleted rows from the DataSet by calling the DataSet's `AcceptChanges` method. However, once you've accepted the changes in the DataSet, you can no longer submit any updates to the database. If you call the DataSet's `RejectChanges` method, then the deleted rows will be restored in the DataSet.

The `RemoveAt` method accepts as argument the index of the row you want to delete in the Rows collection. Finally, the `Delete` method is a method of the `DataRow` class and you must apply it to a `DataRow` object that represents the row to be deleted:

```
customerRow.Delete
```

Locating Rows

To search for specific rows, you can use one of the `FindBy` methods or the `Select` method of the appropriate `DataTable` object. The `FindBy` method is specific to the database and searches the `DataTable` based on the ID of a row. When you add the Products table to a DataSet, the `FindByProductID` method will be generated for you automatically. To use it, just pass the ID of the desired product as argument to this method:

```
Dim selRow As Products.ProductsRow = _  
    Products1.Products.FindByProductID (34)
```

The type that represents the rows of a specific table is a member of the Products class (the definition of the DataSet) and not of the instance of the DataSet. The `FindByProductID` method is a member of the specific DataSet. Notice that the method returns a `DataRow` object, and not the index of the row in the table.

The `Select` method allows you to search a table using SQL-like search criteria and returns an array of `DataRow` objects. The criteria involve column names, values, and comparison operators, such as:

```
ContactName LIKE 'Anton%'
```

or

```
OrderDate < '5/13/2004'
```

You can combine multiple criteria with the AND and OR keywords.

```
Dim selRows() As DataRow  
SelRows = DS.Products.Select('SupplierID=" & supplierID & _  
    " AND Price > 19.99')
```


In this example we've stored the rows returned by the Select method into an array of DataRow objects. Since the DataSet is typed, we could have declared the array as `DS.Products.ProductsRow` type.

[Team Fly](#)

 Previous

Next 

The Rows collection of the DataTable object is just a collection and is not connected in any way to the underlying table in the database. The new row need not be appended at the end of the collection. You can use the InsertAt method of the Rows collection to insert the new row at any place in the collection:

```
DS.Customers.Rows.InsertAt(newRow, idx)
```

The `idx` variable is the index of the new row in the collection.

EDITING WITH CONSTRAINTS

While you're editing a row, the DataSet keeps enforcing the constraints. Sometimes we may wish to edit a row in the DataSet without worrying about referential integrity, or other constraints. When we're done editing the row, the data should be in a consistent state, or else the DataSet won't accept the changes. However, we may wish to violate referential integrity momentarily. By default, the DataSet won't allow you to violate any of the constraints even while editing a row. To maintain referential integrity at all times, we must make sure that the fields are edited in a specific order, which may not be what we want to do while editing the DataSet. To turn off the constraints while editing a DataRow, you can use the BeginEdit and EndEdit methods.

```
Products(3).BeginEdit  
Products(3).CategoryID = 99  
Products(3).EndEdit
```

You can also validate the data from within your code after the user is done editing the row. If there are errors, you can call the CancelEdit method in the place of the EndEdit method to cancel the edit operation. Practical user interfaces, which use regular Windows controls for data entry, usually provide an OK button and a Cancel button. You can call the BeginEdit operation when users click a button to indicate their intention to edit the current row. If they click the OK button, call the EndEdit method; if they click the Cancel button, call the CancelEdit method to reject changes made to the row.

Navigating through a DataSet

The DataTables making up a DataSet may be related, and they usually are. There are methods that allow you to navigate from table to table following the relations between their rows. For example, you can start with a row in the Customers DataTable, retrieve its child rows in the Orders DataTable (the orders placed by the selected customer), and then drill down to the details of each of the selected orders.

The relations of a DataSet are DataRelation objects and are stored in the Relations property of the DataSet. Each relation is identified by a name, the two tables it relates, and the fields of the tables on which the relation is based. It's possible, and really quite simple, to create relations in your code. Let's consider a DataSet that contains the Categories and Products tables. To establish a relation between the two tables, create two instances of the DataTable class to reference the two tables:

```
Dim tblCategories As DataTable = DS.Categories  
Dim tblProducts As DataTable = DS.Products
```

[Team Fly](#)

 Previous

Next 

Then create two `DataColumn` objects to reference the columns on which the relation is based. They're the `CategoryID` columns of both tables:

```
Dim colCatCategoryID As DataColumn = _
    tblCategories.Columns("CategoryID")
Dim colProdCategoryID As DataColumn = _
    tblProducts.Columns("CategoryID")
```

And, finally, create a new `DataRelation` object and add it to the `DataSet`:

```
Dim DR As DataRelation
DR = New DataRelation("Categories2Products", _
    colCatCategoryID, colProdCategoryID)
```

Notice that we need only specify the columns involved in the relation, and not the tables to be related. The information about the tables is derived from the `DataColumn` objects. The first argument of the `DataRelation` constructor is the relation's name. If the relation involves multiple columns, then the second and third arguments of the constructor become arrays of `DataColumn` objects.

To navigate through related tables, the `DataRow` object provides the `GetChildRows` method, which returns the current row's child rows as an array of `DataRow` objects, and the `GetParentRow/GetParentRows` methods, which return the current row's parent row(s). The `GetParentRow` returns a single `DataRow` object and the `GetParentRows` returns an array of `DataRow` objects. Since a `DataTable` may be related to multiple `DataTables`, you must also specify the name of the relation. Consider a `DataSet` with the `Products`, `Categories`, and `Suppliers` tables. Each row of the `Products` table can have two parent rows, depending on which relation you want to follow. To retrieve the product's category, use a statement such as the following:

```
DS.Products(iRow).GetParentRow("CategoriesProducts")
```

The product's supplier is given by the following expression:

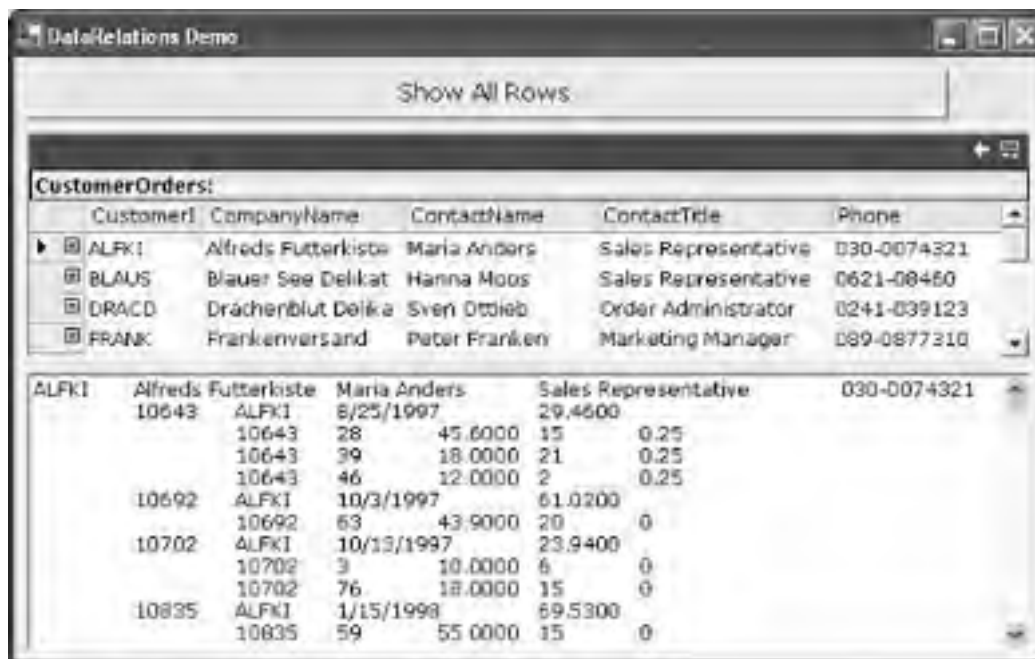
```
DS.Products(iRow).GetParentRow("SuppliersProducts")
```

We're assuming that the `CategoriesProducts` and `SuppliersProducts` relations have already been created, either in the IDE or programmatically.

THE DATA RELATIONS PROJECT

Let's consider a `DataSet` that contains the `Customers`, `Orders`, and `Order Details` tables of the Northwind database. Let's also assume that the `CustomerOrders` and `OrdersOrderDetails` relations have been created. You can open the `DataRelations` project and find the `CustomerOrders` `DataSet` with the `Customers`, `Orders`, and `Order Details` tables of the Northwind database and the appropriate relations. The data is downloaded from the server when the form is loaded, and stored into the `CustomerOrders` `DataSet`, which is bound to the

DataGrid control. The Show All Rows button displays on the lower TextBox control the customers, their names, and the orders' details by iterating through the rows of all tables in the DataSet. Figure 15.10 shows the application's output after clicking the Show All Rows button.



| CustomerID | CompanyName | ContactName | ContactTitle | Phone |
|------------|---------------------|---------------|----------------------|-------------|
| ALFKI | Alfreds Futterkiste | Maria Anders | Sales Representative | 030-0074321 |
| BLAUS | Blauer See Delikat | Hanna Moos | Sales Representative | 0621-08460 |
| DRACD | Dr achenblut Delika | Sven Ottlieb | Order Administrator | 0241-039123 |
| FRANK | Frankenversand | Peter Franken | Marketing Manager | 089-0877310 |

FIGURE 15.10 The DataRelations project demonstrates how to use the relations between tables to iterate through a DataSet.

The code behind the Show All Rows button, which is shown in Listing 15.9, goes through the rows of the Customers table and at each row prints the fields of the row. Then it retrieves the current customer's orders with the GetChildRows method, passing as argument the name of the relation between the Customers and Orders tables. The related rows of the Orders table are returned as an array of CustomerOrder.OrdersRow objects and the code goes through these rows with a For Each...Next loop. For each order, it prints the current order's fields and then retrieves the related detail lines from the OrderDetails table. To do so it calls the GetChildRows method, passing as argument the name of the second relation in the DataSet, and processes the details in a similar manner (prints their fields on the TextBox control).

LISTING 15.9: NAVIGATING THROUGH THE ROWS OF RELATED TABLES

```
Private Sub btnShow_Click(ByVal sender As System.Object, _  
                          ByVal e As System.EventArgs) _  
    Handles btnShow.Click  
    Dim custRow As CustomerOrders.CustomersRow  
    For Each custRow In CustomerOrders1.Customers  
        Dim custCol As DataColumn  
        For Each custCol In custRow.Table.Columns  
            TextBox1.AppendText( _  
                custRow.Item(custCol.ColumnName) & vbTab)  
        Next  
        TextBox1.AppendText(vbCrLf)  
        Dim OrderRows() As CustomerOrders.OrdersRow  
        Dim orderRow As CustomerOrders.OrdersRow  
        OrderRows = custRow.GetChildRows("CustomersOrders")  
        For Each orderRow In OrderRows  
            Dim orderCol As DataColumn
```



```
For Each orderCol In orderRow.Table.Columns
    TextBox1.AppendText( _
        orderRow.Item(orderCol.ColumnName) & vbTab)
Next
TextBox1.AppendText(vbCrLf)
Dim detailRows() As CustomerOrders.Order_DetailsRow
Dim detailRow As CustomerOrders.Order_DetailsRow
detailRows = orderRow.GetChildRows('OrdersOrderDetails")
For Each detailrow In detailRows
    Dim detailCol As DataColumn
    TextBox1.AppendText(vbTab & vbTab)
    For Each detailCol In detailRow.Table.Columns
        TextBox1.AppendText( _
            detailRow.Item(detailCol.ColumnName) & vbTab)
    Next
    TextBox1.AppendText(vbCrLf)
Next
Next
Next
End Sub
```

In Chapter 18 you'll see a better application for navigating through the tables of the Northwind database; it demonstrates how to move not only from parent to child rows, but from child to parent rows as well.

ROW STATES AND VERSIONS

Each row in the DataSet has a State property, which indicates the row's state and its value, is a member of the DataRowState enumeration. The members of the DataRowState enumeration are shown next.

Added The row has been added to the DataTable and the AcceptChanges method has not been called.

Deleted The row was deleted from the DataTable and the AcceptChanges method has not been called.

Detached The row has been created with its constructor, but has not yet been added to a DataTable.

Modified The row has been edited and the AcceptChanges method has not been called.

Unchanged The row has not been edited or deleted since it was read from the database or the AcceptChanges was last called (in other words, the row's fields are identical to the values read from the database). You can use the GetChanges method to find the rows that must be added to the underlying table in the database, the rows to be updated, and the rows to be removed from the underlying table.

If you want to update a single row of a `DataTable`, call an overloaded form of the `DataAdapter`'s `Update` method, which accepts as argument a `DataTable` and submits its rows to the database. The edited rows are submitted through the `UpdateCommand` object of the appropriate `DataAdapter`, the new rows are submitted through the `InsertCommand` object, and the deleted rows are submitted through the `DeleteCommand` object. Instead of submitting the entire table, however, you can create a subset of a `DataTable` that contains only the rows that have been edited, inserted, or deleted. The `GetChanges` method of the `DataTable` object retrieves a subset of rows, depending on the argument you pass to it. This argument is a member of the `DataRowState` enumeration:

```
Dim DT As New DataTable
DT = Products1.Products.GetChanges(DataRowState.Deleted)
```

The preceding statement retrieves the rows of the `Customers` table that were deleted and stores them into a new `DataTable`. The new `DataTable` has the same structure as the one from which the rows were copied, and you can access its rows and their columns as you would access any `DataTable` of a `DataSet`. You can even pass this `DataTable` as argument to the appropriate `DataAdapter`'s `Update` method. This form of the `Update` method allows you to submit selected changes to the database. For more information on updating the underlying data source with `DataAdapters`, see the section "Update Operations," later in this chapter.

In addition to a state, rows have a version. What makes the `DataSet` such a powerful tool for disconnected applications is that it maintains not only data but also the changes in its data. The `Row` property of the `DataTable` object, which represents a `DataRow`, is usually called with the index of the desired row:

```
DS.Tables(0).Rows(i)
```

where *i* is the index of the row we want to access. The `Rows` property can be called with an additional argument, which determines the version of the row you want to read. This argument is a member of the `DataRowVersion` enumeration, whose values are the following:

Current Returns the row's current values (the fields as they were edited in the `DataSet`).

Default Returns the default values for the row. For added, edited, and current rows, the default version is the same as the current. For deleted rows, the default version is the same as the original version. If the row doesn't belong to a `DataTable`, the default version is the same as the proposed version.

Original Returns the row's original values (the values read from the database).

Proposed Returns the row's proposed value (the values assigned to a row that doesn't yet belong to a `DataTable`).

If you attempt to submit an edited row to the database and the operation fails, you can give the user the option to edit the current version of the row, or restore the values of the row to their original values. To retrieve the original version of a row, use an expression such as the following:

```
DS.Tables(0).Row(i, DataRowVersion.Original)
```

While you can't manipulate the version of a row directly, you can use the `AcceptChanges` and `RejectChanges` methods to either accept the changes or reject them. These two methods are exposed by the `DataSet`, `DataTable`, and `DataRow` classes. The difference is the scope. Applying `RejectChanges` to the `DataSet` restores all changes made to the `DataSet` (not a very practical operation). Applying

`RejectChanges` to a `DataTable` object restores the changes made to the specific table's rows; applying the same method to the `DataRow` object restores the changes made to a single row.

The `AcceptChanges` method sets the original value of the affected row(s) to the current value (it will also set each row's state to `Unchanged`). Deleted rows are physically removed. The `RejectChanges` method removes the proposed version of the affected row(s). You can call the `RejectChanges` method when the user wants to get rid of all changes in the `DataSet`. Notice that once you call the `AcceptChanges` method, you can no longer update the underlying tables in the database, because the `DataSet` no longer knows which rows were edited, inserted, or deleted. Call the `AcceptChanges` method only for `DataSets` you plan to persist on disk and not submit to the database.

Using DataViews

In addition to processing tables directly in the `DataSet`, you can view and edit tables through views. A view is another way of looking at a `DataTable` and is represented with a `DataRowView` object. The view may contain a subset of the original table, or the table itself with a different arrangement (sorted by a specific column, for example). The `DataRowView` object allows you to perform certain operations that are not possible with the `DataTable` object. A view can be sorted, for example. The order of the rows in the `DataTable` can't change, and it's the order in which the rows were returned to the client by the server. You can edit the view as you would edit the original `DataTable`. All the changes are reflected immediately to the `DataSet`.

Each `DataTable` object provides a default view, which is identical to the `DataTable` and is accessible through the `DefaultView` property of the `DataTable` object. To create a new view, call the `DataRowView` object's constructor passing the name of the `DataTable` as argument:

```
Dim sortedView As New DataRowView(TblCustomers)
```

The `sortedView` `DataRowView` object can be sorted by setting its `Sort` property to the name(s) of one or more columns and the `ASC` or `DESC` qualifiers. To sort the `sortedView` view by company name within each country, set the `Sort` property to the names of the two fields as shown next:

```
sortedView.Sort = "'Country DESC, CompanyName DESC"
```

The `DESC` qualifier requests that rows will be sorted in descending order. To access the rows of a `DataTable` in sorted order without creating a new `DataRowView` object, set the `Sort` property of the table's default view:

```
TblCustomers.DefaultView.Sort = "CompanyName"
```

Notice that the `TblCustomers` `DataTable`'s rows aren't sorted. You can view them in sorted order by reading the `Rows` collection of the `DataTable`'s default view (`TblCustomers.DefaultView`).

In addition to sorting the view, you can apply a filter to select specific rows. To filter the rows of a view, set the RowFilter property of the corresponding DataView object to an expression like the ones you'd use in the WHERE clause of a selection query. The following expression filters out the customers from all countries but Germany:

```
TBLCustomers.DefaultView.RowFilter = "Country = 'Germany'"
```

Strings within the filtering expression are delimited with single quotes and dates are delimited with the pound symbol. To combine multiple expressions in a filter, use the Boolean operators AND and OR. Both the asterisk (*) and the percent sign (%) can be used as wildcard characters.

Another filtering mechanism is the `RowStateFilter`, which filters rows based on their version and state. The `RowStateFilter` property must be set to one of the values of the `DataViewRowsStates` enumeration, whose members are shown next:

CurrentRows Selects the current versions of the rows. This is the default value and it returns all the rows but the deleted ones.

Added Returns the rows added since the `DataTable`'s `AcceptChanges` method was called.

Deleted Returns the rows deleted since the `DataTable`'s `AcceptChanges` method was called.

ModifiedCurrent Returns the current version of all rows that were modified since the `Data-Table`'s `AcceptChanges` method was called.

ModifiedOriginal Returns the original version of all rows that were modified since `DataTable`'s `AcceptChanges` method was called.

None Returns no rows, just creates a `DataView` with the same structure as the `DataTable`.

OriginalRows Returns the original version of all rows, including the ones that were deleted since the `DataTable`'s `AcceptChanges` method was called.

Unchanged Returns the rows that haven't been changed since the `DataTable`'s `AcceptChanges` method was called.

Finally, the `DataView` class provides two methods for searching the column on which the view is sorted. The difference between the search methods of the `DataView` and the search methods of the `DataTable` is that the `DataView` class's search methods return the index of the matching rows in the view, and not the actual row. If you'd rather work with the index, you should use the `DataTable`'s default view. Suppose you're displaying the rows of a `DataTable` on a `ListView` control. For certain operations it may be more convenient to retrieve the index of a selected row, rather than the row itself, so that you can locate the same row on the `ListView` control instantly.

The searching methods of the `DataView` class are the `Find` and `FindRows` methods. They both accept as argument a value (or an array of values) and locate the row with the same value in the column(s) on which the view is sorted. The `Find` method returns the index of the first match, while the `FindRows` method returns an array with the matching rows. The following statements create a new view on the `TBLCustomers` `DataTable`, sort it according to the `ContactTitle` column, and then locate the index of the first contact whose title is "Sales Representative."

```
Dim DVCustomers As New DataView(DS.TBLCustomers)
DVCustomers.Sort = "ContactTitle"
Dim idx As Integer
idx = DVCustomers.Find("Sales Representative")
```

To retrieve all the contacts that are sales representatives in an array of `DataRow` objects, use the following statements:

```
Dim DVCustomers As New DataView(DS.TBLCustomers)
DVCustomers.Sort = "ContactTitle"
Dim rows() As DataRow
rows = DVCustomers.Find("Sales Representative")
```

[Team Fly](#)

 Previous

Next 

how the DataGrid marks the rows that failed to update the underlying tables. It's a very convenient approach for developers, but you can't sell this type of interface. It's not what end users want to see. Real applications deploy Windows forms where users can edit the DataSet's tables. You can still use DataSets and DataAdapters, but if the Update operation fails you must handle it somehow from within your code. The DataTable object provides the members you need to retrieve the rows that failed to update the database, as well as the messages returned by the database server. You'll see how these members are used in this section.

To demonstrate the techniques of this section we'll return to the NWProducts project and add some code to the Update Database button, which submits the changes to the database. You'll find more practical examples of handling update errors in Chapter 18. Here we'll show you how to retrieve the rows that failed to update the database, as well as the reason why, from within our code. Once you know what went wrong, you'll be able to build the appropriate interface to help users correct their mistakes and try again. Some errors can't be corrected. If the user has edited a row that has been removed in the meantime by another user, there's not much you can do, short of removing the row from its DataTable and accepting this change. With some additional effort, you can create a new row with the same values as the deleted one and submit it to the database as a new row (with a different ID, of course).

The Update Database button calls the Update method of the three DataAdapters:

```
Dim CategoryRows, SupplierRows, ProductRows As Integer
CategoryRows = DACategories.Update(Products1)
SupplierRows = DASuppliers.Update(Products1)
ProductRows = DAProducts.Update(Products1)
```

The values returned are the number of rows affected in each table. The Update methods may not have updated all the rows in the underlying tables. If a product was removed in the meantime from the Products table in the database, the DataAdapter's UpdateCommand will not be able to submit the changes made to the specific product. A product with a negative value may very well exist in the DataSet (which doesn't enforce arbitrary constraints), but the database will reject this row, because it violates one of the constraints of the Products table. As we mentioned, you should validate the data as best as you can at the client before submitting them to the database. The ContinueUpdateOn- Error property of the three DataAdapter objects is set to True, so that they will submit all the changes to the database, even if some of the rows fail to update the underlying tables.

If the database returned any errors during the update process, the HasErrors property of the DataSet object will be set to True. You can retrieve the rows in error from each table with the Get- Errors method of the DataTable class. This method returns an array of DataRow objects and you can process them in any way you see fit. The code shown next iterates through the rows of the Categories table that are in error and prints the description of the error on the Output window:

```
If Products1.HasErrors Then
    If Products1.Categories.GetErrors.Length = 0 Then
        Console.WriteLine('No errors in the Categories DataTable")
    Else
        Dim RowsInError() As Products.CategoriesRow
        RowsInError = Products1.Categories.GetErrors
        Dim row As Products.CategoriesRow
```


LISTING 2.10: WRITING TO A FILE VIA STREAMING

```
Private Sub Button1_Click_1(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click

    Dim strText As String = TextBox1.Text

    If (strText.Length < 1) Then
        MsgBox( "Please type something into the TextBox so we ca
        Exit Sub
    Else
        Dim strFileName As String = "C:\MyFile.txt"
        Dim objOpenFile As FileStream = New FileStream(strFileNan
        FileMode.Append, FileAccess.Write, FileShare.Read)
        Dim objStreamWriter As StreamWriter = New StreamWriter(ok

        objStreamWriter.WriteLine(strText)

        objStreamWriter.Close()
        objOpenFile.Close()
    End If

End Sub
```

Because you used the `FileMode.Append` property, each time you run this program new text will be added to any existing text in the file. If you want to overwrite the file, use `FileMode.Create` instead. Alternatively, you can save to a file by borrowing functionality from the `SaveFileDialog` class (or the `SaveFileDialog` control), as shown in Listing 2.11.

LISTING 2.11: WRITING TO A FILE USING THE SAVEFILEDIALOG OBJECT

```
Dim sfd As New SaveFileDialog()
    Dim dlgResponse As Integer
    Dim strFname As String

    sfd.DefaultExt = "txt" ' specifies a default extension
    sfd.InitialDirectory = "C:"

    dlgResponse = sfd.ShowDialog

    If dlgResponse = 1 Then
        strFname = sfd.FileName
        msgbox(strFname)
    End If
```

```
        Console.WriteLine(''Errors in the Categories table")
    For Each row In RowsInError
        Console.WriteLine(vbTab & row.CategoryID & vbTab & _
            row.RowError)
    Next
End If
End If
```

The `DataRow` object exposes the `RowError` property, which is a description of the error that prevented the update for the specific row. It's possible that the same row has more than a single error. To retrieve all columns in error, call the `DataRow` object's `GetColumnsInError`, which returns an array of `DataColumn` objects—the columns in error.

So, it's possible to retrieve information about the update errors, even if you're using the `DataAdapter`'s `Update` method to submit the changes to the database. Of course, the `DataAdapter` is not the only method of submitting changes to the database.

One of the overloaded forms of the `Update` method allows you to specify the rows to be submitted to the database. The `DataAdapter`'s `Update` method can accept as argument a `DataTable`, or an array of `DataRow` objects, and it will update only the specified rows.

Handling Identity Columns

An issue that deserves special attention in coding data-driven applications is the handling of Identity columns. Identity columns are used as primary keys, and each row is guaranteed to have a unique Identity value, because this value is assigned by the database the moment the row is inserted into its table. The client application can't generate unique values. When new rows are added to a `DataSet`, they're assigned Identity values, but these values are unique in the context of the local `DataSet`. When a row is submitted to the database, the Identity columns will be assigned their final values by the database. The temporary Identity value assigned by the `DataSet` is also used as foreign key value by the related rows, and we must make sure that every time an Identity value is changed, the change will propagate to the related tables.

Handling Identity values is a very important topic, and here's why. Consider an application for entering orders or invoices. Each order has a header and a number of detail lines, which are related to a header row with the `OrderID` column. This column is the primary key in the `Orders` table and the foreign key in the `Order Details` table. If the primary key of a header is changed, the foreign keys of the related rows must change also. Figure 15.11 shows the `Transactions` application that creates new orders and submits them to the database. The new orders are displayed on a `DataGrid` control and you can view their headers and their details. You can submit them to the database one at a time, or in batch mode, by clicking the `Submit Orders` button.

This application has two basic requirements:

- The order header and the order detail must be related at all times. If the ID of an order header is changed, the IDs of its related rows in the `Order Details` table must also change.

- Orders must be entered into the database in a transactional context. This means that if a detail line can't be inserted into the Order Details table, the entire order must be aborted.

[Team Fly](#)

 Previous

Next 

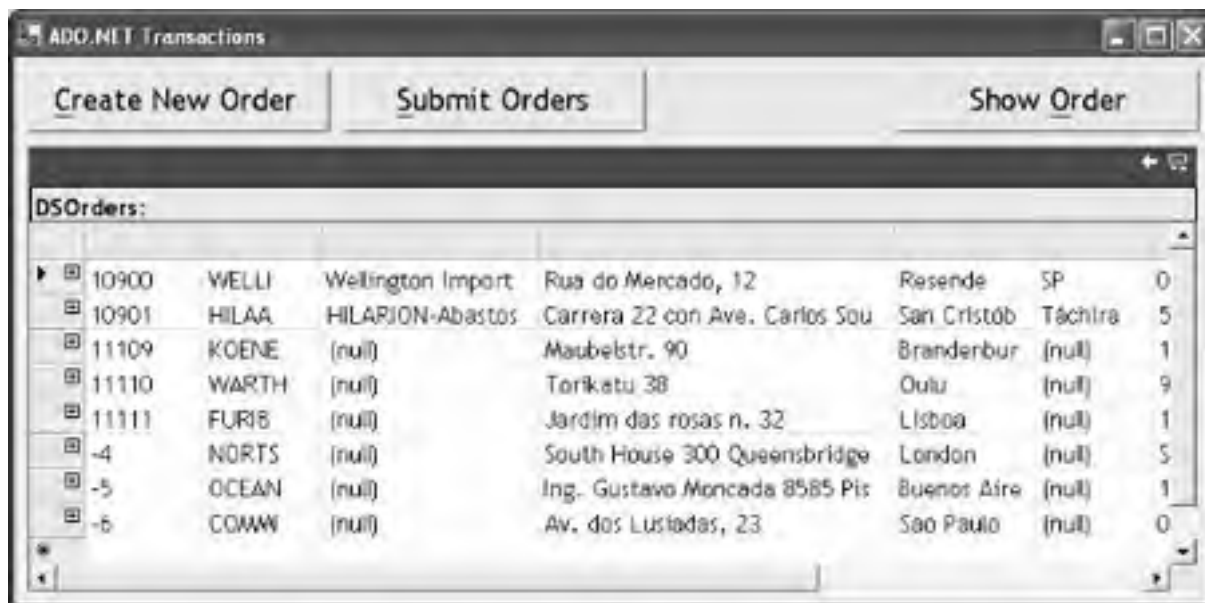


FIGURE 15.11 The Transaction project demonstrates how to insert new orders in transactional mode.

In this section we'll explore the techniques for maintaining the relation between headers and details, which are based on Identity values. In the following section we'll discuss the second requirement, namely how to implement transactions with ADO.NET.

Handling Identity columns is a crucial issue in developing disconnected data-driven applications. The best way to handle this situation is to stop using Identity columns and replace them with GUID columns. The GUID created at the client will be unique even after the corresponding row is inserted into the database, and the relation will remain valid. This option, however, is out of the question for many existing databases. It's trivial to change the schema of the database, but this would break all the applications that use the database.

The trick in handling Identity columns is to make sure that the values generated by the DataSet will be replaced by the database. We do so by specifying that the Identity column's starting value is -1 and its autoincrement is -1 . The first ID generated by the DataSet will be -1 , the second one will be -2 , and so on. Negative Identity values will be rejected by the database, because the AutoIncrement properties in the database schema are positive. By submitting negative Identity values to SQL Server we make sure that new, positive values will be generated and used by SQL Server.

We must also make sure that the new values will replace the old ones in the related rows. In other words, we want these values to propagate to all related rows. The DataSet allows you to specify that changes in the primary key will propagate through the related rows with the UpdateRule property of the Relation.ChildKeyConstraint property. Each relation exposes the ChildKeyConstraint property, which determines how changes in the primary key of a relation affect the child rows. This property is an object that exposes a few properties of its own. The two properties we're interested in are the UpdateRule and DeleteRule (what happens to the child rows when the parent row's primary key is changed, or when the parent row is deleted).

You can use one of the following rules:

Cascade Foreign keys in related rows change every time the primary key changes value, so that they'll always remain related to their parent row.

None The foreign key in the related row(s) is not affected.

SetDefault The foreign key in the related row(s) is set to the DefaultValue property for the same column.

SetNull The foreign key in the related rows is set to Null.

As you can understand, setting the UpdateRule property to anything other than Cascade will break the relation. If the database doesn't enforce the relation, you may be able to break it. If the relation is enforced, however, the UpdateRule must be set to Rule.Cascade, or else the database will not accept changes that violate its referential integrity.

If you set the UpdateRule to None, you may be able to submit the order to the database. However, the detail rows may refer to a different order. This will happen when the ID of the header is changed because the temporary value is already taken. The detail rows will be inserted with the temporary key and they'll be added to the details of another order. Notice that no runtime exception will be thrown and the only way to catch this type of error is by examining the data inserted into the database by your application. By using negative values at the DataSet, we make sure that the ID of both the header and all detail rows with a negative ID will be rejected by the database.

THE TRANSECTIONS PROJECT

Now we can look at the code of the Transactions project, which inserts new orders to the Northwind database using DataAdapters. Create a form with the controls you see in Figure 15.11. The Create New Order button creates a random order and adds it to the client DataSet. This DataSet is bound to the DataGrid control, and you can view the new orders and their details on the control. The Submit Orders sends all the new orders to the database. The third button allows you to retrieve an order by its ID and add it the local DataSet.

Let's start by setting up the DataAdapters, to move data in and out of the Northwind database, and the DataSet, where we'll store our data at the client. Drop the Customers table on the form and rename the SqlDataAdapter that will be added to the form to `DACustomers`. Then drop the Products, Orders, and Order Details tables on the form and rename the corresponding DataAdapters to `DAProducts`, `DAOrders`, and `DADetails`, respectively. Each new order will be submitted to the Orders and Order Details tables in the database, so we'll create a DataSet with these two tables.

Configure the DataAdapters with the SELECT statements shown next:

DACustomers DataAdapter

```
SELECT    CustomerID, CompanyName, Fax, Country,
          PostalCode, Region, City, Address, ContactTitle, ContactName
FROM      Customers
```

Do not generate INSERT/UPDATE/DELETE statements for this DataAdapter, because we won't update the Customers table; we'll use it to look up customer information.

DAProducts DataAdapter

```
SELECT ProductID, ProductName, UnitPrice  
FROM Products
```

Do not generate INSERT/UPDATE/DELETE statements for this DataAdapter, because we won't update the Products table; we'll use it to look up product information.

[Team Fly](#)

 Previous

Next 

DAOrders DataAdapter

```
SELECT    OrderID, CustomerID, EmployeeID, OrderDate,
          RequiredDate, ShippedDate, ShipVia, Freight,
          ShipName, ShipAddress, ShipCity, ShipRegion,
          ShipPostalCode, ShipCountry
FROM      Orders
WHERE     OrderID = @orderID
```

DADetails DataAdapter

```
SELECT    OrderID, ProductID, UnitPrice, Quantity, Discount
FROM      [Order Details]
WHERE     OrderID = @orderID
```

Then generate two DataSets, one for storing the static tables (Customers and Products) and another one with the updateable tables (Orders and Order Details). The first DataSet contains the tables that need not update the database and will be used as lookup tools to select customers and products for our orders. Call this DataSet `DSTables` and add the Customers and Products tables to it. The other DataSet is the `DSNewOrder` DataSet and it contains the Orders and Order Details tables. You must also add a relation between the two tables of the `DSNewOrder` DataSet based on the OrderID column of the two tables. The parent table in the relation is the Orders table and the child table is the Order Details table. The default name of this relation is rather clumsy, so change it to `OrdersOrderDetails`, because we'll use it in our code.

When the form is loaded we must populate the `DSTables` DataSet with the following statements:

```
DACustomers.Fill(DsTables1, "Customers")
DAProducts.Fill(DsTables1, "Products")
```

We must also set the properties of the Orders DataTable, so that it can handle the relations as discussed earlier in this section. The following statements set the Identity attributes of the OrderID column of the Orders DataTable.

```
DsOrders1.Orders.OrderIDColumn.AutoIncrement = True
DsOrders1.Orders.OrderIDColumn.AutoIncrementSeed = -1
DsOrders1.Orders.OrderIDColumn.AutoIncrementStep = -1
Finally, we must set the UpdateRule for the relation between the Orders and Ord
DsOrders1.Relations("OrdersOrderDetails")._
    ChildKeyConstraint.UpdateRule = Rule.Cascade
```

These are the statements that must be executed in the form's Load event handler. The `DSOrders` DataSet will be populated with new orders at runtime and is initially empty. The code behind the Create New Order button creates random new orders to populate the Orders and Order Details tables of the `DSOrders` DataSet. The customer to whom the order belongs is selected randomly from

the Customers table and the order's shipping address is set to the customer's address. The order's body is filled with random products. We set the number of detail lines (a random value between 3 and 10) and then we select as many random rows from the Products table. We create a `detailLine` variable to represent each detail line (the type of the variable is `DSOrders.Order_Details`), assign values to its columns, and then add it to the Order Details DataTable. The `OrderID` column of each detail line is set to the `OrderID` column of its parent row. Listing 15.10 shows the code behind the Create New Order button on the form of the application:

LISTING 15.10: CREATING A NEW ORDER

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

    Dim rnd As New System.Random
    Dim custrow As Integer
    custrow = rnd.Next(0, DsTables1.Customers.Rows.Count - 1)
    Dim customerRow As DSTables.CustomersRow
    customerRow = DsTables1.Customers.Rows(custrow)

    Dim hdrOrder As DSOrders.OrdersRow
    Dim TBLOrders As DataTable = DsOrders1.Orders
    hdrOrder = TBLOrders.NewRow
    hdrOrder.CustomerID = customerRow.CustomerID
    hdrOrder.OrderDate = Now.Today.ToShortDateString
    If Not customerRow.IsAddressNull Then _
        hdrOrder.ShipAddress = customerRow.Address
    If Not customerRow.IsCityNull Then _
        hdrOrder.ShipCity = customerRow.City
    If Not customerRow.IsPostalCodeNull Then _
        hdrOrder.ShipPostalCode = customerRow.PostalCode
    If Not customerRow.IsCountryNull Then _
        hdrOrder.ShipCountry = customerRow.Country
    DsOrders1.Orders.Rows.Add(hdrOrder)

    Dim detLines As Integer = rnd.Next(3, 10)
    Dim detLine As Integer
    Dim TBLDetails As DataTable = DsOrders1.Order_Details
    Dim prodRow As DSTables.ProductsRow
    Dim detailLine As DSOrders.Order_DetailsRow
    For detLine = 1 To detLines
        detailLine = TBLDetails.NewRow
        detailLine.OrderID = hdrOrder.OrderID
        prodRow = DsTables1.Products.Rows(rnd.Next(1, 77))
        detailLine.ProductID = prodRow.ProductID
        detailLine.Quantity = rnd.Next(1, 100)
        detailLine.UnitPrice = prodRow.UnitPrice
        detailLine.Discount = rnd.NextDouble * 0.3
    Next
```

```
Try
    TBLDetails.Rows.Add(detailLine)
Catch ex As Exception
    Console.WriteLine('Could not add product"& _
                    detailLine.ProductID)
End Try
Next
End Sub
```

The Add method that adds each detail line to the Order Details DataTable is embedded in a structured exception handler. The OrderID and ProductID columns of the Order Details table in the Northwind database form a unique constraint: an order may not contain two detail lines that refer to the same product. Because the products are selected randomly, it's possible that a new detail line may refer to the same product as an existing one. When this happens, the DataSet will reject the second row and a runtime exception will be thrown. Our code handles this condition silently by simply ignoring the row that refers to an existing product and continues.

As new orders are entered, they're assigned a negative ID and are automatically displayed on the DataGrid control, which is bound to the DSOrders DataSet. You can verify that headers and their details are related with negative ID values. You can also retrieve any order from the database and add it to the DSOrders DataSet with the Show Order button. The existing order will be displayed along with the new ones, but it will have its final ID (which is a positive value).

To commit the new orders, click the Submit Orders button. If they're inserted into the database successfully, the new OrderID values will be returned to the DataSet and they will appear on the DataGrid control. The InsertCommand of the DataAdapter executes an INSERT statement to insert the new row and then retrieves the newly inserted row and returns it to the DataSet. Clicking the Submit Orders button without entering a new order has no effect, because the DataSet contains no changes and there's nothing to be submitted to the database.

Take a look at the IDs of the orders shown in Figure 15.11. Initially, we retrieved the orders with an ID value of 10900 and 10901 with the Show Order button. Then we added a couple of new orders and submitted them to the database. The IDs assigned to these orders are 11109, 11110, and 11111. Then we added three more new orders, just before capturing the screen of Figure 15.11. These orders will be assigned their final IDs the next time the Submit Orders button is clicked again.

The code that inserts the new orders to the database is quite trivial; it just calls the Update method of the two DataAdapters to insert the appropriate rows to the Orders and Order Details tables, in that order, as shown in Listing 15.11.

LISTING 15.11: SUBMITTING ORDERS TO THE DATABASE

```
Private Sub btnSubmitOrder_Click(ByVal sender As System.Object, _
                                ByVal e As System.EventArgs) _
    Handles btnSubmitOrder.Click
    DAOrders.Update(DsOrders1, "Orders")
```

```
DADetails.Update(DsOrders1, "Order Details")  
End Sub
```

[Team Fly](#)

 Previous

Next 

We were able to update the underlying tables with a few trivial statements, because we've configured the DataSet and the DataAdapters properly. If you don't set the AutoIncrement properties and the UpdateRule, you wouldn't be able to commit the orders to the database and be sure that the relation between the two tables remains in effect.

The suggested approach will work, but not always as expected. What if a row can't be added to the Orders or the Order Details table? The remaining rows will be added and we may end up with an order that contains fewer detail rows than it should, a header without detail lines, or even detail lines without a header (orphan rows in the Order Details table). This brings us to the last topic discussed in this chapter, the topic of transactions.

Performing Transactions with the DataAdapter

Some actions against a database must be performed as a single operation: they must either succeed or fail. Orders and invoices are typical examples, and so is the transferring of funds between accounts. If a detail line can't be entered into the Order Details table, then the entire order must be rejected. Otherwise, we'll end up with a partial order. Likewise, we can't remove an amount from one account and not deposit it to another account. Operations involving multiple steps that must either succeed or fail as a whole are known as *transactions*.

All DBMSs support transactions and this is one of the most important topics in database programming: any non-trivial application involves transactions. While a transaction is in process, the participating rows are locked and other applications can't access them. The DBMS—SQL Server in our case—takes care of locking the rows during the transaction and unlocking them when the transaction completes. It's therefore crucial that transactions are performed as fast as possible. Imagine a situation where a transaction is initiated when the user starts entering an invoice and completes when the invoice is ready. Rows will be unnecessarily locked for long periods of time, and users won't be able to complete their own transactions if one of the rows they need is locked by another user.

In ADO.NET, transactions are implemented through a Transaction object. As with most ADO.NET classes, the Transaction class is an abstract one and you must use an OleDbTransaction or a SqlTransaction, depending on the database you're writing to. Moreover, all the steps of a transaction must be performed through the same Connection object. If your application calls for transactions that involve multiple databases, you must use the COM+ services, which are discussed in Chapter 16.

First, you initiate a Transaction object by calling the BeginTransaction method of a Connection object. The Transaction object must be used with all the Command objects that will participate in the transaction. The Command class provides a Transaction property, which must be set to a Transaction object. All the Command objects that refer to the same Transaction object are executed in the context of the same transaction. If you're using DataAdapters, you must set the Transaction property of the InsertCommand, DeleteCommand, or UpdateCommand objects of the DataAdapter, depending on the type of action you want to perform. To end a successful transaction, you must call the Transaction object's Commit method. If the transaction failed, you call the RollBack method of the Transaction object to

restore the affected rows to their version before the transaction began.

To commit the rows of an order in the context of a transaction, you must assign the Transaction object to the Transaction property of the DAOrders.InsertCommand and DADetails.InsertCommand objects. Then you can perform the steps of the transaction as you would if the same steps were executed independently of one another. In other words, call the Update method of both DataAdapters. If all steps are successful, you can call the Transaction object's Commit method to finalize the transaction.

The changes made to the database by the commands of the transaction will become visible as soon as the Commit method is executed. If one of the steps fails, you must call the Rollback method of the Transaction object to abort the transaction. To detect an error condition that may prevent the transaction from completing successfully, you can embed the statements that update the database into a structured error handler. The following code segments outline a transactional update operation:

```
Dim CN As New SqlClient.SqlConnection()
' Set up the SqlConnection object
' and open a connection to the database
CN.Open()
' Create and initialize a Transaction object
Dim TRN As SqlClient.SqlTransaction
TRN = CN.BeginTransaction()
Try
    ' Set up a Command object
    Dim CMD1 As New SqlClient.SqlCommand()
    CMD1.Connection = CN
    CMD1.CommandText = . . .
    CMD1.CommandType = . . .
    ' Execute the commands in the context
    ' of the Transaction object
    CMD1.Transaction = TRN
    CMD1.ExecuteNonQuery
    ' Execute additional commands in the
    ' context of the same transaction
    Dim CMD2 As New SqlClient.SqlCommand()
    CMD2.Connection = CN
    CMD2.CommandText = . . .
    CMD2.CommandType = . . .
    ' Execute the second command in the context
    ' of the same Transaction object
    CMD2.Transaction = TRN
    CMD2.ExecuteNonQuery
    ' and finally commit all the actions
    TRN.Commit
Catch exc As Exception
    ' Handle errors
    TRN.Rollback
End Try
CN.Close
```

The code is substantially simpler if you're using DataAdapters to submit data to the database. In this case, you don't have to set up your own Command objects; you simply call the Update method of each DataAdapter in the context of a transaction.

Revise the code of the Submit Orders button of the Transactions project, so that the calls to the Update method of the two DataAdapters take place in the context of a transaction, as shown in Listing 15.12.

LISTING 15.12: SUBMITTING ORDERS IN THE CONTEXT OF A TRANSACTION

```
Private Sub btnSubmitOrder_Click(ByVal sender As System.Object, _  
                                ByVal e As System.EventArgs) _  
                                Handles btnSubmitOrder.Click  
    Dim TR As SqlClient.SqlTransaction  
    SqlConnection1.Open()  
    TR = SqlConnection1.BeginTransaction  
    DAOrders.InsertCommand.Transaction = TR  
    DADetails.InsertCommand.Transaction = TR  
    Try  
        DAOrders.Update(DsOrders1, "Orders")  
        DADetails.Update(DsOrders1, "Order Details")  
        TR.Commit()  
    Catch ex As Exception  
        MsgBox ex.Message  
        TR.Rollback  
    Finally  
        SqlConnection1.Close()  
    End Try  
End Sub
```

This code makes sure that no partial orders will be recorded. However, it will work as expected only if you submit one order at a time. If you enter several new orders at the client DataSet and submit them together, should a single detail line of a single order fail, then no order will be committed to the database. Orders and invoices are usually submitted to the database as soon as they're entered, so the code shown here will work with most real-world applications. The transaction's scope, however, is much larger than it need be. We want to prevent the insertion of a partial order to the database, but not reject all orders because only one of them failed.

To submit each order in its own transaction context, we must modify the code a little, so that it extracts the header and detail rows of each order and submits them to the database in the context of a separate transaction. This way, if one transaction fails, it will not affect the others. We'll show you the code, which also demonstrates how to use an overloaded form of the Update method to submit arrays of rows to the database.

The revised code creates two arrays of DataRow objects for each order, one with a single header row and another with the order's details. Then it calls the Update method of the two DataAdapters passing the appropriate array as argument. The process is repeated for each order, and each time the code sets up a different Transaction object. Listing 15.13 shows the revised code that submits each order to the database in the context of its own transaction.

LISTING 15.13: SUBMITTING MULTIPLE ORDERS IN SEPARATE TRANSACTION CONTEXTS

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

```
Dim TR As SqlConnection.SqlTransaction
SqlConnection1.Open()
Dim headerRows(0) As DSOrders.OrdersRow
Dim detailRows() As DSOrders.Order_DetailsRow
Dim iRow As Integer
For iRow = 0 To DsOrders1.Orders.Count - 1
    headerRows(0) = DsOrders1.Orders.Rows(iRow)
    detailRows = headerRows(0).GetChildRows(' 'OrdersOrderDetails"
    TR = SqlConnection1.BeginTransaction
    DAOrders.InsertCommand.Transaction = TR
    DADetails.InsertCommand.Transaction = TR
    Try
        DAOrders.Update(headerRows)
        DADetails.Update(detailRows)
        TR.Commit()
    Catch ex As Exception
        TR.Rollback()
        headerRows(0).Delete()
        MsgBox(ex.Message)
    End Try
Next
SqlConnection1.Close()
End Sub
```

TESTING THE TRANSACTIONAL UPDATES

You'll want to test the code and verify that it works as expected. To force a few transactions to fail, we must enter data that violate a database constraint. We'll choose a constraint that's not enforced by the DataSet, so that we can enter invalid data in the DataSet and have the database itself reject the bad data. One such constraint is that the price of each detail line is a positive value. Change the statement that assigns a value to the UnitPrice column of each detail row as follows:

```
detailLine.UnitPrice = prodRow.UnitPrice - 8
```

Some of the detail lines will have a negative price and the database will reject them. Most of the products are more expensive than \$8, so some of the orders will contain valid items. Run the application, enter a few new orders, and submit them to the database. As you create new orders with the Create New Order button, you can view each order's detail lines on the DataGrid and make sure that the set of new orders includes valid and invalid orders. As the program attempts to commit the orders to the database, you'll see a message box with the error description for each order that failed.

Some of the random orders generated by the application will fail to update the underlying tables in the database. These orders must also be removed from the DataSet. To remove an order, the program deletes the order's header by calling the Delete method of the DataRow object that represents this header of the invalid order. Because the UpdateRule was set to Cascade, when the parent row is deleted so are the child rows. The DataSet contains only the orders that were committed to the database successfully, in addition to new orders that haven't been committed yet and the orders you've added to the DataSet with the Show Order button.

[Team Fly](#)

 Previous

Next 

(Then add code here to actually save this file.)

Yet another alternative reads and writes individual pieces of data. The size of these pieces is up to you when you define the variable used to write and when you define the read mode (as in `r.ReadByte()` versus `r.ReadBoolean` or `r.ReadInt32`, and so on).

Listing 2.12 is an example of how to create a file and store bytes into it.

LISTING 2.12: WRITING AND READING DATA IN VARIOUS SIZE UNITS WITH THE BINARYWRITER

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

    Dim fs As FileStream = New FileStream('c:\test1.txt', FileMode.Create)

    Dim w As BinaryWriter = New BinaryWriter(fs)

    Dim i As Byte

    ' store 14 integers in a file (as bytes)
    For i = 1 To 14
        w.Write(i)
    Next

    w.Close()

    fs.Close()

    'And here' s how you read individual bytes from a file...
    ' Create the reader for this particular file:
    fs = New FileStream("c:\test1.txt", FileMode.Open, FileAccess.Read)
    Dim r As New BinaryReader(fs)

    ' Read the data, in bytes, from the Test1.txt file:
    For i = 1 To 14
        Debug.WriteLine(r.ReadByte())
    Next i
    r.Close()
    fs.Close()
```

BinaryReader and BinaryWriter can read and write information in quite a few different data types. Type this line, and when you type the . (period), you see the list of data types:

```
Debug.WriteLine(r.
```

Also note that there are several FileModes you can define. CreateNew, the mode used in the preceding example, breaks with an error if the file already exists.

TIP To replace an existing file, use `FileMode.Create` instead of `FileMode.CreateNew`.

computer to provide a rich user experience), or a Web application. If the client is a Web application, the interaction with the user is not as rich, because Web applications can't interact instantly with the user as Windows forms do. Web applications aren't called limited or restricted clients, as one would expect, but the choice of the term "rich client" for Windows applications says it all. The major limitation of Web applications is that they don't interact directly with the user: they submit information to the server and then display the results returned by the server.

In a client/server environment, the database represents the data tier: it's the part of the application that executes queries. Everything else belongs to the presentation tier, which consists of the code that interacts with the user. Business logic (the code that implements basic operational rules that reflect the corporation's policies) also belongs to the presentation tier. The client/server architecture is great for many applications, but it wasn't designed for the Web. The current architecture for building applications is the multi-tier architecture, which we'll discuss later.

To better understand the multi-tier architecture, you should consider for a moment the problems of client/server architecture. Many of you have written client/server applications with VB6 and you may already be familiar with them. Client/server applications don't scale very well. They assume a connection to the database and use it to pass data back and forth all the time. As more and more clients are added to the network, the database server spends more time and resources to maintain the client connections. Moreover, VB6 developers based their interfaces on the data-bound controls, which submit changes to the database as soon as they occur. This resulted in applications that don't scale well. Every client maintains its own connection to the database, which may be acceptable for a few dozen to a few hundred local users. Take this application to the Web and the server can't keep up with the number of requests.

The problem of maintaining database connection for the duration of the application was eliminated by the disconnected nature of ADO.NET. With ADO.NET, clients can no longer bind their interface directly to the database; instead, they must request the data, store them to the client (usually in a DataSet), and process them locally. While the DataSet is being processed at the client, no load is placed to the database server. After processing the data, the application requests a new connection to the database and submits the changes.

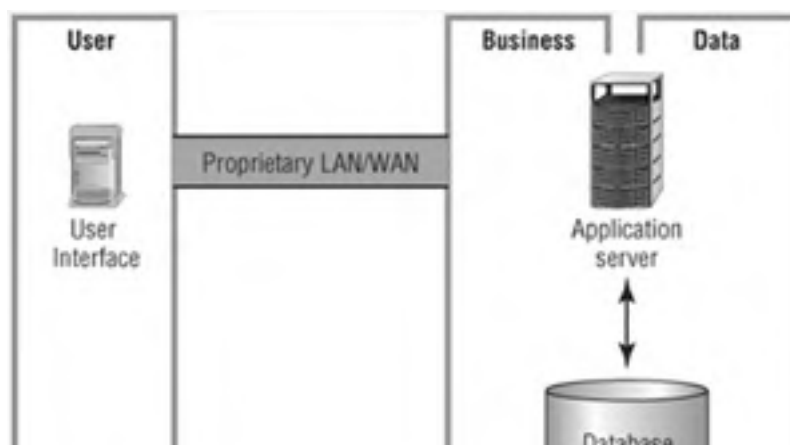




FIGURE 16.1 The client/server architecture

In addition to the data and presentation tiers, multi-tier architectures introduced a third layer of functionality, the *middle tier* (or *business tier*, or *application tier*). This is the code that implements the business logic. Business logic is the part of the application that implements rules specific to a corporation. Such rules include the establishment of customer credit, discount policies, the calculation of insurance premiums, and so on. All components that don't clearly belong to the other two tiers belong to the middle tier. In short, the middle tier is a layer of code between the client and the database, which isolates the other two tiers. The obvious advantage of this architecture is that components that don't belong to the presentation tier are isolated and can be maintained and deployed separately from the presentation tier. This isn't the only advantage of introducing another tier to the application, nor the most important one, in our view.

What Exactly Is a Business Rule?

Let's consider a typical example of a business rule. Every corporation has a discount policy, which may change quite often, because it's dictated by business needs. This is a business rule that must be implemented in code. The most crucial part of your application is the code that implements the business rules, because they must be implemented accurately and efficiently. They affect the company's daily operations and it's usually the management that determines them. The discount policy can make a huge difference in the company's sales. If you incorporate it in the client application's code, then every time the corporation needs to revise this rule, you must modify the client application and deploy it to all the workstations. This is a very inefficient deployment approach, especially if the client application is running on a large number of workstations. If the business rule is implemented in a middle-tier component (see Figure 16.2), which is deployed on a single machine, every time the corporation's discount policy changes, we need only revise a single component and deploy it to a single machine. All clients will see the new version of the component and will use the new discount policy.

So far, we've been using the terms "tiers" and "layers" indiscriminately—and we'll continue to do so. Technically, there's a fine difference between the two terms. Layer refer to the logical separation, while tier refers to a physical separation. When the layers of the application are distributed on different computers, we talk about the tiers of the application. Once you learn to design your applications with distinct layers, you'll be able to implement tiers by deploying your layers on different machines.

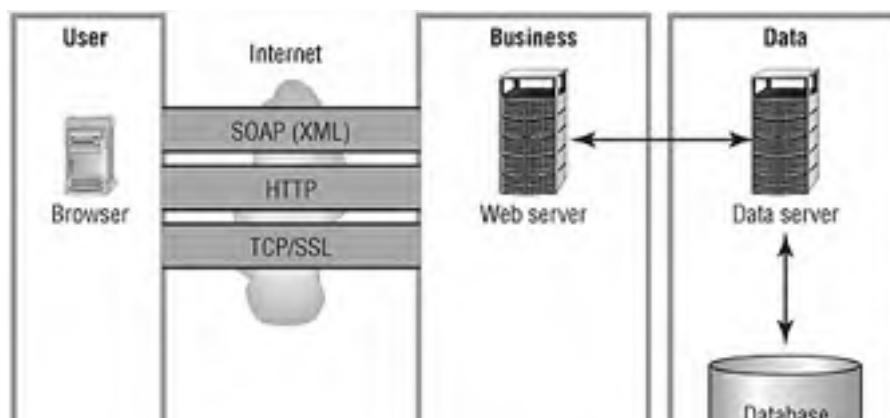




FIGURE 16.2 Multi-tier architecture

The real advantage of the using middle-tier components is that we no longer rely on multiple developers to understand and implement the business rules, which may lead to inconsistencies in an application (same rule implemented differently in different parts of the application). As a VB6 developer, you've learned how to break your applications into components and reuse them in your code. With VB.NET, componentizing an application comes naturally, because classes permeate the design of the new language. In our view, the biggest advantage of multi-tier architecture is that it encourages the separation of the application into distinct layers of functionality. Even if you don't plan to deploy these layers on different machines, componentizing large applications has distinct benefits, such as maintainability and code reuse.

Another benefit of the middle tier is that it allows you to create custom objects to exchange information with the presentation tier. Instead of passing a DataTable with the customers, we can create a collection of Customer objects, each one representing a different customer. The custom object can expose methods for the basic operations against the database as well. The Customer class, for example, may expose methods such as GetCustomerByID, GetCustomersByName, AddCustomer, EditCustomer, and so on. The developers working at the presentation tier can use these methods to interact with the database and never have to execute statements directly against the database. A well designed middle-tier component can totally isolate the presentation tier from the data tier. If you later decide to introduce some changes in the database, you need only change a few members of a middle-tier component and the rest of the application will work as is.

Microsoft suggests using DataSets to pass data between tiers. DataSets are very efficient and, in effect, they're small databases residing in the client computer's memory. If you plan to use data-binding in your presentation tier, go ahead and use DataSets—you don't have any other options, anyway. Professional applications do not make extensive use of data-binding techniques, so you can choose how to pass data between tiers. In our experience, using custom objects that represent business entities is far more convenient than using DataSets.

Building custom objects for your application is not a trivial task. You must first understand the business needs of an application and then implement them in code. This means a thorough design of the application, which will inevitably lead to a more robust, easier-to-maintain application. Of course, it also means a substantial effort in designing the application before coding it. Practically speaking, most of the small projects would never finish if we had to use custom objects. However, the use of objects becomes a necessity for large projects. When you have many developers working in tandem, you must standardize the way they code. The best method of standardizing your development team is to hide as many details as possible, and this can be achieved through custom objects.

Componentizing a project is the best method to achieve code reuse (software's Holy Grail). A component is written once and used in many places in an application, or by different applications. As a team leader, you can have a large number of developers working on an application's presentation tier and make sure that none of them programs directly against the database. If you provide classes to represent business entities, developers working at the presentation level can program against these objects.

The last, but not least, of the benefits of using a middle tier is that it isolates the presentation tier from the data tier. What this means is that no developer can execute queries against the database directly. All developers who need to enter a new invoice will call a method of the middle-tier component, passing the order's values as arguments. The method's code determines whether the stock of the items will be affected, or whether the customer's balance will be affected. You don't run the risk of inserting a partial order because one of the developers didn't implement a proper transaction. You

can even change the method's code so that it adjusts the stock of the items and re-deploy it. The clients need not be aware of this change, and the presentation tier will function with the revised middle-tier component.

Designing with Middle-Tier Components

To demonstrate the design of middle-tier components we'll build a "true" middle-tier component, as opposed to simple components that convert units or perform other trivial tasks. One of this book's examples is an application that prepares orders and invoices. We've used this example in several chapters, because we consider it to be a very practical component of every business application. Figure 16.3 shows the interface of the application, which is based on the ListView control. To enter a new detail line, the user can enter either the ID of the desired product or its name. If multiple products match the description, their names appear in a drop-down list where the user can select the desired one. This is the NWOrders application we present in Chapter 18.

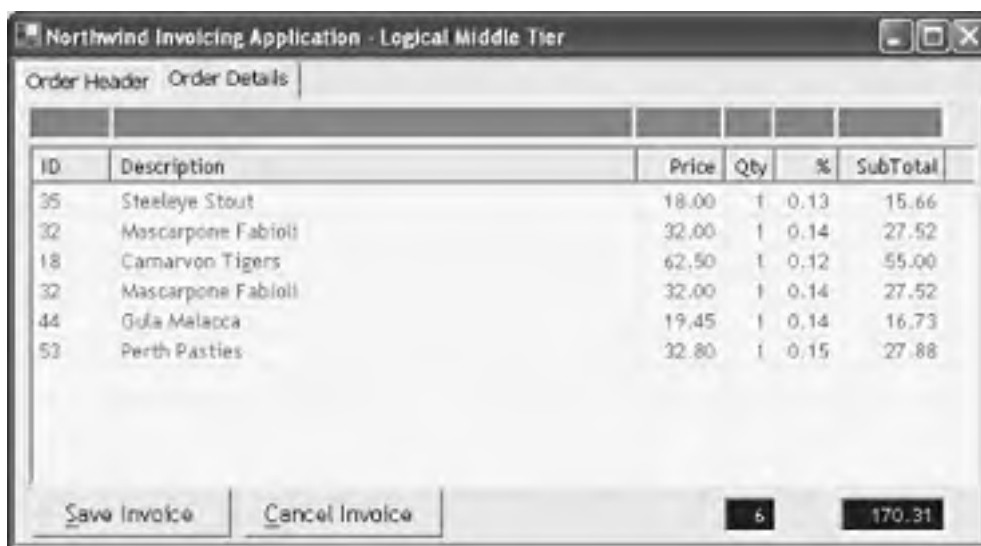


FIGURE 16.3 The user interface of an invoicing application

Once a product is selected, the user can set the quantity and the discount. A real application shouldn't let the user determine the discount. Corporations have discount policies, which are implemented in the application's business logic. Clearly, letting the user determine the discount is not wise. Implementing the discount policy in the presentation tier is doable, but not recommended. If the same rule is to be implemented by several developers, you must make sure that they all understand the corporation's discount policy. Even then, you have to deal with changes in this business rule. What happens when the corporation changes its discount policy? As mentioned earlier, deploying the application on a large number of workstations should be avoided, when possible.

We're going to postpone for a while the issue of deployment and focus on a component that implements a basic business rule, the company's discount policy. We'll implement this rule as a component and ask the presentation-tier developers to simply reference it in their code. You can then change the implementation of the middle tier and generate a new DLL, and all client applications will see the new DLL. This allows you to revise the discount policy as dictated by business needs. Later in this chapter you'll see how to deploy the new component on a single machine and have all clients request the discount from this computer.

Let's start with a quick overview of the application's architecture. The details of the application are in the code that prepares the invoice, and they're discussed in Chapter 18. Here we'll focus on the

[Team Fly](#)

 Previous

Next 

application's middle-tier component, which performs some operations against the database. These operations are implemented as methods; they are the following:

GetProductsByName Accepts a product name (or part of a product's name) as argument and returns an array of objects that represent the products, whose names match the string passed to the method as argument.

GetProductByID Accepts a product ID as argument and returns an object that represents the specified product.

AddOrder Accepts as argument an object that represents the entire order and commits it to the database. If the order is committed successfully, the method returns the ID of the new order; otherwise a negative value is returned.

With these three operations in place, you can focus on the presentation tier's code and ignore the specifics of the database. You can pass the middle-tier component to another developer, who can write a functional interface for preparing orders/invoices without knowing anything about the database. A team of developers can use this component to develop a Windows application, while another team can develop a Web application. The foundation of both applications will be the component that interacts with the database. You can even revise the middle-tier component later to work with a different database, or a remote database. The presentation tier need not be touched. This is the middle-tier contribution to your application development team: it completely decouples the presentation tier from the rest of the application. Of course, if some changes in the presentation tier require additional parameters (the product's measurement unit or the currency, for example), you'll have to revise the code in multiple layers and re-deploy the application. In a well designed application, however, you'll be able to revise one tier independently of the others. To convince yourself, you should try to edit the middle-tier component of this section's sample application, so that it works with a different database. It's almost trivial; each product has an ID, a description, and a price, and this is all the information you need to pass between the database and the middle-tier component.

The middle-tier component of the invoicing application uses two custom objects to exchange data with the presentation tier: the `ProductPrice` object, which represents a product and its price, and the `OrderedProduct` object, which represents a detail line in the order. The definition of the classes that implement the two custom objects are shown next:

```
Public Class ProductPrice
    Public ProductName As String
    Public ProductID As String
    Public ProductPrice As Decimal
    Public Overrides Function ToString() As String
        Return ProductName
    End Function
End Class

Public Class OrderedProduct
    Public ProductID As String
    Public ProductPrice As Decimal
    Public ProductQTY As Integer
    Public ProductDiscount As Decimal
End Class
```


The two classes abstract the structure of the database by providing all the information needed to prepare and commit an order to the database. It doesn't really matter where the data originates, or how it's structured. You can easily modify the implementation of the component's methods to make them work with a different database, even with a DataSet that's persisted in XML and never committed to a database.

Note that the ProductPrice class overrides its ToString method. This simple trick allows us to add instances of the ProductPrice class to a ListBox control. We simply pass an object to the Add method of the control's Items collection. The string displayed on the control is the same string returned by the ToString method. However, the item added to the ListBox control is a ProductPrice object, which carries with it the product's ID and price, in addition to the product's name. To find out the ID of the selected item on the control, you can cast the selected item to the ProductPrice type and request its ProductID property:

```
Dim SelProductID As Integer  
SelProductID = CType(ListBox1.SelectedItems(0), ProductPrice).ProductID
```

DESIGNING A DISCOUNT POLICY COMPONENT

The component that implements the discount policy should be designed so that it can accommodate future policies as well. We'll create a method that accepts a few arguments and returns the discount as a decimal value. What kind of information do we usually need to calculate discounts? Most discount policies are based on two pieces of information: who's buying and what they're buying. Our method, therefore, will accept two arguments: the product's ID and the customer's ID, and it will return the discount for the specific product.

This scheme is quite flexible. You can group the customers and offer different group discounts that apply to all products (in this case, the product's ID is irrelevant). You can also maintain different price lists for each group of customers. The component can determine the customer's group from the customer's ID and then look up the discount for the desired product in the current price list. An even more complicated technique is to offer a discount based on past purchases of the same customer. Actually, this is the discount policy we'll implement in our sample code. It's quite complicated, because it must execute a non-trivial query against the database to find out the total amount spent in the past by a specific customer for a specific product. The discount is proportional to this amount (with an upper limit, of course).

As you realize, all policies use two pieces of information to determine the discount: the customer and product IDs. The implementation of the component may change at will, as long as the business policy doesn't require additional data. To implement the discount policy, we'll build a class that exposes a single method, the GetItemDiscount method, whose implementation is shown in Listing 16.1.

LISTING 16.1: THE GETITEMDISCOUNT METHOD OF THE BUSINESSLAYER COMPONENT

```
Public Class BusinessLayer  
    Public Shared Function GetItemDiscount(  
        ByVal CustomerID As String, ByVal ProductID As Integer)  
        As Integer
```

```
Dim CMD As New SqlClient.SqlCommand()  
CMD.CommandText = "GetItemDiscount"  
CMD.CommandType = CommandType.StoredProcedure
```

[Team Ely](#)

 Previous

Next 

```
CMD.Parameters.Add(New SqlClient.SqlParameter( _
    '@ProductID', ProductID))
CMD.Parameters.Add(New SqlClient.SqlParameter( _
    '@CustomerID', CustomerID))
Dim CN As New SqlClient.SqlConnection()
CN.ConnectionString = "initial catalog=Northwind;" & _
    "integrated security=SSPI;" & _
    "persist security info=False;" & _
    "workstation id=POWER Toolkit;packet size=4096"
CN.Open()
CMD.Connection = CN
Dim discount As Integer
discount = CMD.ExecuteScalar
' additional discount processing statements here !
Return CType(discount, Integer)
End Function
End Class
```

The code is almost trivial: it calls a stored procedure, the `GetItemDiscount` stored procedure, passing as arguments the values passed to the method by its calling application. The `GetItemDiscount` stored procedure goes through the specified customer's past orders and calculates the discount. Listing 16.2 shows the `GetItemDiscount` stored procedure, which you must attach to the Northwind database before you run the application.

LISTING 16.2: THE GETITEMDISCOUNT STORED PROCEDURE

```
CREATE PROCEDURE GetItemDiscount
@CustomerID nchar(5),
@ProductID int
AS
DECLARE @CustomerTotal int
SET @CustomerTotal =
    (SELECT ROUND(SUM(unitprice*quantity*(1-discount)),0,0)
     FROM [Order Details] INNER JOIN Orders
        ON Orders.OrderID = [Order Details].OrderID
WHERE Orders.CustomerID = @CustomerID AND
      [Order Details].ProductID=@ProductID)
IF @CustomerTotal IS NULL
    SELECT 12
ELSE
BEGIN
    IF @CustomerTotal < 1200
        SELECT 12 + @CustomerTotal / 100
    ELSE
        SELECT 24
END
```

Converting the BusinessLayer Class to a Web Service

Converting an existing class to a Web service and exposing its methods through a web server is almost trivial. Start by copying the folder of the NWOrders application to another location and change its name to `NWOrdersWebService`. Copy the code of the `BusinessLayer` and `Orders` classes and then delete the two classes from the project. The middle-tier components will become available to our application through a Web service, so they need not be part of the new client application. We'll return to the `NWOrdersWebService` project and revise its code a little, so that it will interact with the database through the new Web service we're going to build now.

Start a new instance of Visual Studio and create a Web service project. Name the new project `BusinessLayer`. To make the application a little more challenging, we'll implement all the classes that make up the application's middle tier into a Web service, not just the discount policy. Any client, whether a Windows or a Web application, will be able to interact with the database and prepare invoices by calling the methods of the Web service through the Web.

To turn a class into a Web service, we create a new class that imports the `System.Web.Services` namespace and inherits from the `System.Web.Services.WebService` class. In addition, the methods that must be exposed as web methods must be prefixed with the `<WebMethod>` attribute. Listing 16.3 shows the code of the `BusinessLayer` Web service. The classes that define the custom objects exposed by the Web service and the code of the original class's methods are identical to the ones found in the `NWOrders` project.

LISTING 16.3: THE BUSINESSLAYER WEB SERVICE

```
Imports System.Web.Services

<System.Web.Services.WebService (Namespace:=
    "http://tempuri.org/BooksBLayer/Service1")> _
Public Class BusinessLayer
    Inherits System.Web.Services.WebService

    Public Class OrderedProduct
        Public ProductID As String
        Public ProductPrice As Decimal
        Public ProductQTY As Integer
        Public ProductDiscount As Decimal
        Public Sub New()

        End Sub
    End Class

    Public Class ProductPrice
        Public ProductName As String
        Public ProductID As String
        Public ProductPrice As Decimal
        Public Sub New()

        End Sub
    End Sub
End Class
```



```
Public Overrides Function ToString() As String
    Return ProductName
End Function
End Class
```

```
<WebMethod()> _
Public Function GetItemDiscount(ByVal CustomerID As String, _
                                ByVal ProductID As Integer) As Integer
    Dim CMD As New SqlClient.SqlCommand
    CMD.CommandText = 'GetItemDiscount'
    CMD.CommandType = CommandType.StoredProcedure
    CMD.Parameters.Add(New SqlClient.SqlParameter( _
        "@ProductID", ProductID))
    CMD.Parameters.Add(New SqlClient.SqlParameter( _
        "@CustomerID", CustomerID))
    Dim CN As New SqlClient.SqlConnection
    CN.ConnectionString = "initial catalog=Northwind;" & _
        "integrated security=SSPI;" & _
        "persist security info=False;" & _
        "workstation id=POWER Toolkit;packet size=4096"
    CN.Open()
    CMD.Connection = CN
    Dim discount As Integer
    discount = CMD.ExecuteScalar
    ' additional discount processing statements here !
    Return CType(discount, Integer)
End Function
```

```
<WebMethod()> _
Public Function GetProductsByName(ByVal ProductName As String) _
    As ProductPrice()
    Dim CMD As New SqlClient.SqlCommand
    CMD.CommandText = "GetProductsByName"
    CMD.CommandType = CommandType.StoredProcedure
    CMD.Parameters.Add(New SqlClient.SqlParameter( _
        "@productName", ProductName))
    Dim CN As New SqlClient.SqlConnection
    CN.ConnectionString = "initial catalog=Northwind;" & _
        "integrated security=SSPI;" & _
        "persist security info=False;" & _
        "workstation id=POWER Toolkit;packet size=4096"
    CN.Open()
    CMD.Connection = CN
    Dim DR As SqlClient.SqlDataReader = CMD.ExecuteReader()
    Dim P(999) As ProductPrice
    Dim prod As New ProductPrice
    Dim i As Integer
```

```
While DR.Read
    prod = New ProductPrice
    prod.ProductID = DR.Item("ProductID")
    prod.ProductName = DR.Item("ProductName")
    prod.ProductPrice = DR.Item("UnitPrice")
    P(i) = prod
    i = i + 1
End While
ReDim Preserve P(i - 1)
If i > 0 Then
    Return P
Else
    Return Nothing
End If
End Function

<WebMethod()> Public Function _
    GetProductByID(ByVal ProductID As Integer) As ProductPri
    Dim CMD As New SqlClient.SqlCommand
    Dim CN As New SqlClient.SqlConnection
    CN.ConnectionString = "initial catalog=Northwind;" & _
        "integrated security=SSPI;" & _
        "persist security info=False;" & _
        "workstation id=POWERTOOLKIT;packet size=4096"
    CN.Open()
    CMD.Connection = CN
    CMD.CommandText = "GetProductByID"
    CMD.CommandType = CommandType.StoredProcedure
    CMD.Parameters.Add(New SqlClient.SqlParameter( _
        "@productID", ProductID))
    Try
        Dim DR As SqlClient.SqlDataReader = CMD.ExecuteReader()
        Dim P As ProductPrice
        If DR.Read Then
            P = New ProductPrice
            P.ProductID = DR.Item("ProductID")
            P.ProductName = DR.Item("ProductName")
            P.ProductPrice = DR.Item("UnitPrice")
        End If
        Return P
    Catch ex As Exception
        Return Nothing
    End Try
End Function

<WebMethod()> _
Public Function AddOrder(ByVal customerID As String, _
    ByVal empId As Integer, _
```

```
ByVal Order() As OrderedProduct) _
As Boolean
' Commits the order passed through the
' function's arguments to the database
Dim CMD As New SqlClient.SqlCommand
CMD.CommandText = 'AddHeader'
CMD.CommandType = CommandType.StoredProcedure
CMD.Parameters.Add(New SqlClient.SqlParameter( _
"@customerID", customerID))
CMD.Parameters.Add(New SqlClient.SqlParameter( _
"@employeeID", empID))
Dim CN As New SqlClient.SqlConnection
CN.ConnectionString = "initial catalog=Northwind;" & _
"integrated security=SSPI;" & _
"persist security info=False;" & _
"workstation id=POWERTOOLKIT;packet size=4096"
CMD.Connection = CN
Dim TRN As SqlClient.SqlTransaction
CN.Open()
TRN = CN.BeginTransaction
CMD.Transaction = TRN
Try
Dim OrderID As Integer
OrderID = CMD.ExecuteScalar()
CMD = New SqlClient.SqlCommand
CMD.Connection = CN
CMD.Transaction = TRN
CMD.CommandText = "AddDetailLine"
CMD.CommandType = CommandType.StoredProcedure
CMD.Parameters.Add(New SqlClient.SqlParameter( _
"@OrderID", Data.SqlDbType.Int))
CMD.Parameters.Add(New SqlClient.SqlParameter( _
"@ProductID", Data.SqlDbType.Int))
CMD.Parameters.Add(New SqlClient.SqlParameter( _
"@Quantity", Data.SqlDbType.Int))
CMD.Parameters.Add(New SqlClient.SqlParameter( _
"@Price", Data.SqlDbType.Money))
CMD.Parameters.Add(New SqlClient.SqlParameter( _
"@Discount", Data.SqlDbType.Real))
Dim Item As OrderedProduct
For Each Item In Order
CMD.Parameters("@OrderID").Value = OrderID
CMD.Parameters("@ProductID").Value = Item.ProductID
CMD.Parameters("@Quantity").Value = Item.ProductQTY
CMD.Parameters("@Price").Value = Item.ProductPrice
CMD.Parameters("@Discount").Value = Item.ProductDisc
CMD.ExecuteNonQuery()
Next
```

```
        Catch exc As Exception
            TRN.Rollback()
            CN.Close()
            Return False
        End Try
        TRN.Commit()
        CN.Close()
        Return True
    End Function
```

End Class

Once the Web service is in place, you can test it by executing the application (that is, you don't have to write a client to test the Web service). Visual Studio will invoke Internet Explorer and will give you a list of the methods exposed by the Web service. Press F5 to run the project; Internet Explorer will open, displaying the description of the Web service, as shown in Figure 16.4.

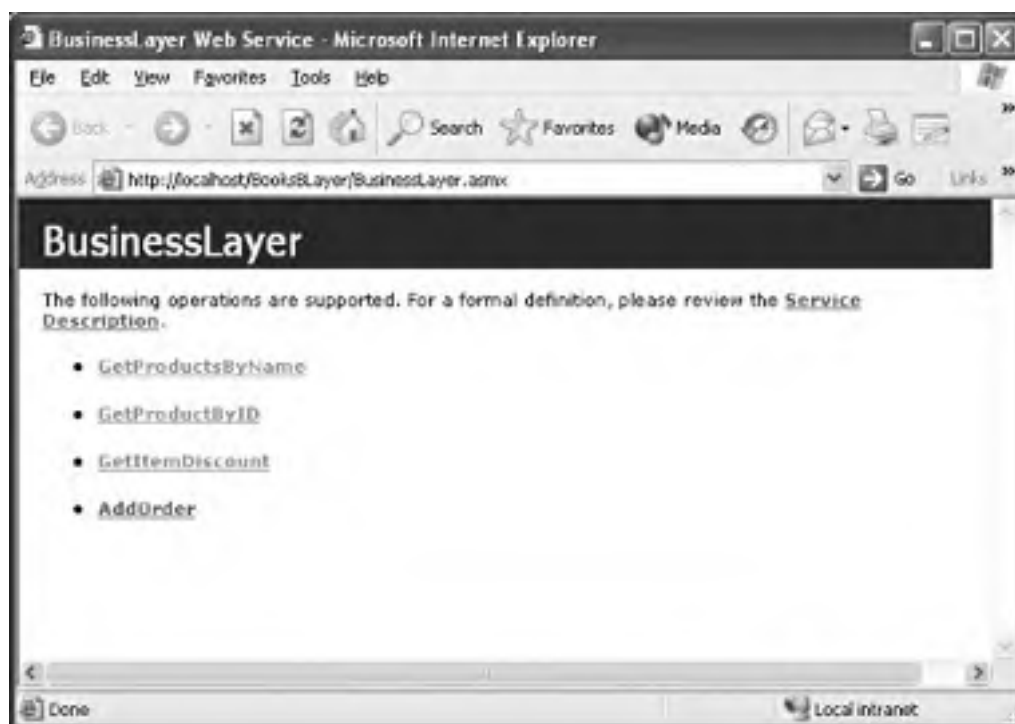


FIGURE 16.4 Viewing a description of your Web service's method

In addition to viewing the methods of the Web service, you can execute these methods from within the browser and make sure they behave as expected. You won't be able to test the methods that accept custom objects as arguments, but you will still be able to test much of the Web service's functionality without building a test project.

To test the `GetItemDiscount` method, click the corresponding hyperlink in the window of Figure 16.4 and you will see another form, shown in Figure 16.5, prompting you to enter the arguments required by the method. If you scroll down the form of Figure 16.5 you will see the description of the request that will be made to the service and the expected response. Web services are exposed to remote clients as URLs to specific applications. The arguments are passed just like the parameters of a form when submitted to a web server as part of a request.

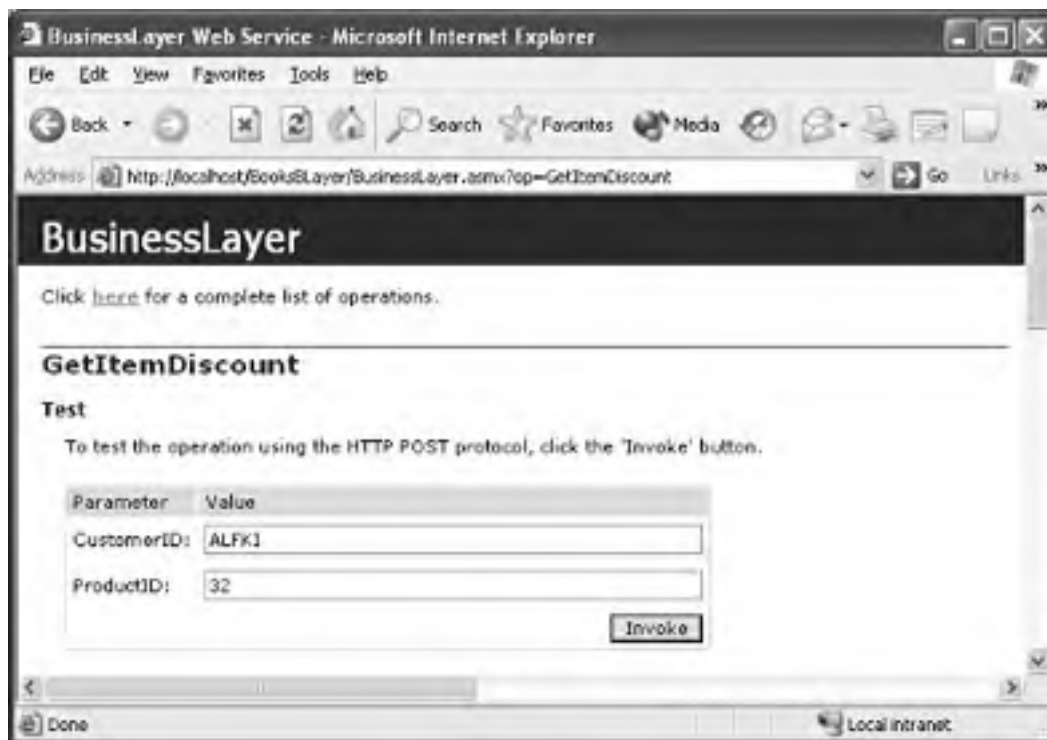


FIGURE 16.5 Preparing the parameters to be passed to a web method

Click the Invoke button and you will see yet another form, shown in Figure 16.6, with the Web service's response. The response of the GetItemDiscount method is just an integer value. If you retrieve a product by name, you will see the XML description of an instance of the ProductPrice class (see Figure 16.7). The Web service serialized an instance of the object in XML format and passed it to the client.

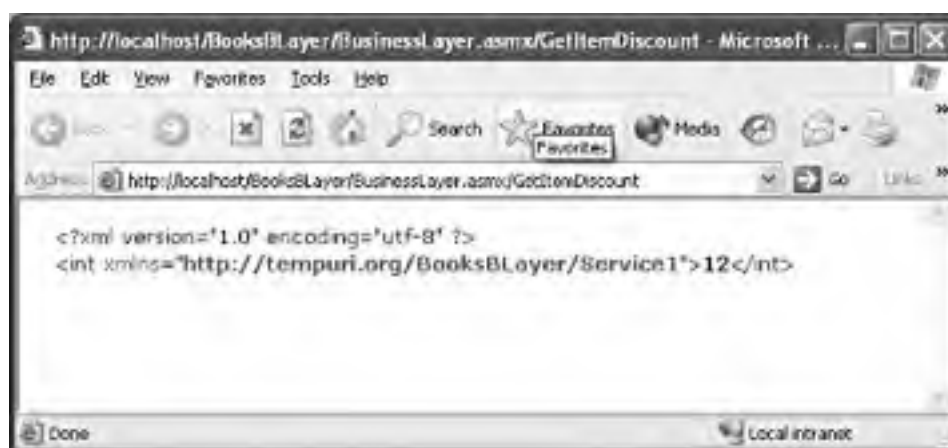


FIGURE 16.6 Testing the GetItemDiscount method

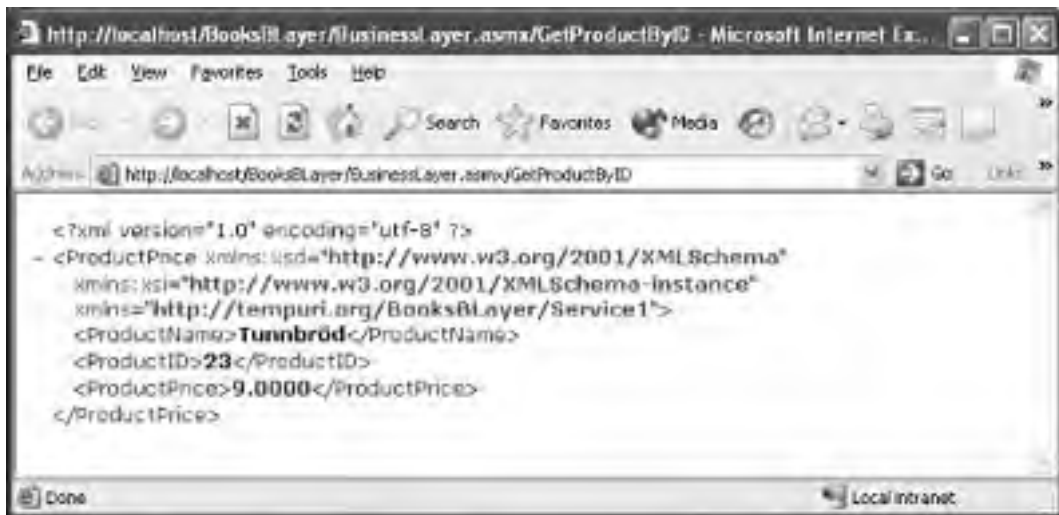


FIGURE 16.7 Testing the GetProductByID method

If you attempt to test the `GetProductsByName` method using a large table, you may get an exception. The array that holds the names of the selected products is dimensioned to hold 1,000 products. Should the query retrieve more than 1,000 products, an exception will be thrown. The best method to handle this situation is to limit the number of selected rows from within the corresponding stored procedure to 100, or 1,000, using the TOP N clause. Users may specify generic search criteria that will result in thousands of rows. However, your application need not download a very large number of rows to the client (users should be able to identify the product they're trying to sell more precisely).

The code of the Web service is practically identical to the code of the original `BusinessLayer` component. The new class is derived from the `System.Web.Services.WebService` class, and all methods are prefixed with the `<WebMethod>` attribute, which makes it possible to call them through HTTP. They're executed on the same machine on which the web server is running and return their result in SOAP format (which is basically XML) to the client. As you will see, the client will parse the result without any code on your part and use it to prepare the invoice. Let's see how we can use the `BusinessLayer` Web service in our client application.

REFERENCING A WEB SERVICE

Now we'll switch to the client application that prepares invoices and revise its code so that it will use the newly created Web service. We'll revise the application a little, but we won't have to rewrite the presentation tier's code. Actually, we'll only add to our project a reference to the Web service and we'll edit the declarations of certain objects, which are now supplied by the Web service.

Make a copy of the `NWOrders` application's folder (it's the `WebInvoiceBLayer` sample project) and open it with the IDE. Throw away the `BusinessLayer` and `OrderClass` classes. We just moved the functionality of these two classes into a Web service, so we'll add a reference to the Web service in the place of the two classes. Right-click the solution's name and from the context menu select `Add Web Reference`. The dialog box of Figure 16.8 will appear on your desktop, which prompts you to select the source of the Web service. In our case the Web service resides on the local machine, so click the `Web Services On The Local Machine` hyperlink.





FIGURE 16.8 Adding a Web reference to a .NET project

In the next window you will see a list of all Web services on the local machine. Find and click the hyperlink that corresponds to the desired Web service (its name is Service1 and its URL is <http://localhost/BusinessLayer/Service1.asmx>) and it will be automatically added to your project. The localhost item will appear under the Web References branch in the Solution Explorer, which means that you can use the Web service in the application's code, just like any other component. If you revise the Web service's code, you must refresh the definition of the Web service at the client (force the CLR—Common Language Runtime—to take a trip to the server and retrieve the latest definition of the Web service) by selecting Update Web References from the localhost item's context menu. As you can guess, you can add references to many Web services, on the same or different computers. For each Web reference, a new item will be added to the Web References section of the Solution Explorer, and this item will be named after the computer on which the Web service resides.

Now go to the project's code and change the references to the middle-tier component. To create a reference to the BusinessLayer component, use the following statement:

```
Dim Invoice As New localhost.BusinessLayer
Likewise, to retrieve a product by its ID, use the following statements:
Dim Product As localhost1.ProductPrice
Dim BLayer As New localhost.BusinessLayer()
Product = Invoice.GetProductByID(txtID.Text.Trim)
```

The rest of the code requires similar minor changes. It's like changing the name of the middle-tier component. This time the middle-tier component is not a class of the project but a component running under IIS, and it exposes its functionality as web methods. You can either open the project and examine its code, or paste the code of the original application, NWOrder, and fix the problems that will be caused by the deletion of the original middle-tier component.

WEB APPLICATIONS VERSUS RICH-CLIENT APPLICATIONS

As you can understand, it's also possible to build a Web application for preparing orders. All you have to do is create a new presentation tier, which this time will be a WebForm, so that users can connect to it from any workstation that can access the Web. This isn't a trivial task, but if you're familiar with Web applications, you can certainly design a function application for preparing invoices that can be invoked from within Internet Explorer. Of course, the Web application won't react to the Enter keystroke instantly as the equivalent Windows application does, nor will you be able to switch between the order's headers and details lines with the arrow keys.

Why resort to a Web application, which can't provide the rich user experience of a Windows application? You can actually distribute the Web application to a number of clients (either on the same network or remote). The presentation tier's code uses the .NET Framework and will execute on any system on which the .NET runtime has been installed. To communicate with the database, the presentation tier's code uses a Web service, which runs on a web server and can be accessed from anywhere. As you may recall from Chapter 10, you can deploy the application from the same web server that hosts the application's Web service. You don't even have to install the application on the clients—just give them the URL of the application on the web server.

As you can understand, Web services allow you to combine the convenience of distributing Web applications with the rich user experience of a Windows application. During the last few years there's been an increased demand for Web applications, because they can be deployed easily to a large number of clients and they enable users to access the application from anywhere on the web. The same is more or less true for Windows applications that make use of Web services. We expect to see a departure (or, should we say return) from Web-based applications to Windows, or rich-client, applications.

[Team Fly](#)

 Previous

Next 

Converting the BusinessLayer to a Remote Service

The second technique for invoking components on remote systems is remoting. Remoting is a new technique introduced with .NET that is similar to using Web services, but is faster and gives you more control over the format of the data moved back and forth between the server and the client. Remoting is not as simple to set up, either. To remote a component, we need an application to host our component and make its services available to a number of remote clients through a TCP or HTTP port. The application that will host the component could be a Windows application, or console application, or an ASP application. We'll use an ASP application for the purposes of this example, basically because an ASP application is always available and need not be started, unlike Windows and console applications.

Create a new ASP.NET Web Application project and name it DiscountServer. When the ASP project is created, the item Service1.asmx is automatically added to the project. This is where you'd normally place your application's code, but you don't need to do anything about this file. Just add the component we've used in the previous example to the project. Because the component will pass instances of custom objects to the clients and back, the class must be marked as serializable. Create a new class, the ProductPrice class, and insert in it the code of the BusinessLayer class. Listing 16.4 is the code of the ProductPrice class (we're not showing the code of the methods, because it's no different than the code shown in Listing 16.3).

LISTING 16.4: THE REVISED BUSINESSLAYER COMPONENT

```
<Serializable()> _
Public Class Product
    Public Class Price
        Public ProductID As Integer
        Public ProductName As String
        Public ProductPrice As Decimal
        Public ProductDiscount As Decimal
    End Class
End Class

Public Class OrderedProduct
    Public ProductID As String
    Public ProductPrice As Decimal
    Public ProductQTY As Integer
    Public ProductDiscount As Decimal
    Public Sub New()

    End Sub
End Class

<Serializable()> Public Class NewProduct
    Public Class ProductPrice
        Public ProductName As String
        Public ProductID As String
        Public ProductPrice As Decimal
        Public Sub New()
```

```
        End Sub
        Public Overrides Function ToString() As String
            Return ProductName
        End Function
    End Class

    Public Function GetItemDiscount(ByVal CustomerID As String, _
                                    ByVal ProductID As Integer) _
        As Integer
        . . .
    End Function

    Public Function GetProductsByName(ByVal ProductName As String) _
        As ProductPrice()
        . . .
    End Function

    Public Function GetProductByID(ByVal ProductID As Integer) _
        As ProductPrice
        . . .
    End Function

    Public Function AddOrder(ByVal customerID As String, _
                             ByVal empId As Integer, _
                             ByVal Order() As OrderedProduct) _
        As Boolean
        . . .
    End Function

End Class
```

To expose the functionality of your middle-tier component through remoting, you must add a few statements in the application's configuration file. Double-click the item Web.config in the Solution Explorer, and when the configuration file's code appears in the editor's window, enter the statements in Listing 16.5 right after the <configuration> tag.

LISTING 16.5: CONFIGURING AN ASP APPLICATION TO HOST A REMOTABLE COMPONENT

```
<system.runtime.remoting>
  <application>
    <service>
      <wellknown mode=""singlecall" type="Discounts,
        DiscountServer.Discounts" objecturi="Discounts.soap">
```

To create a way of accessing Form1, create an object variable in a Module; then assign Form1 to this variable when Form1 is created. The best place to do this assigning is just following the InitializeComponent() line in the Sub New() constructor for Form1.

Create a module (choose Project ➤ Add Module). Modules are visible to all the forms in a project. In this module, you define a public variable that points to Form1. In the Module, type this:

```
Module Module1
    Public f1 As Form1()
End Module
```

Notice that the NEW keyword was not employed here. You are merely creating an object variable that will be assigned to point to Form1. Click the + next to "Windows Form Designer generated code" in Form1's code window. Locate Form1's constructor (Public Sub New) and just below the InitializeComponent() line, type this:

```
InitializeComponent()
F1 = Me
```

This assigns Me (Form1, in this case) to the public variable F1. Now, whenever you need to communicate with Form1 from any other form, you can use F1. For example, in Form2's Load event, you can have this code:

```
Private Sub Form2_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    F1.BackColor = Color.Brown
End Sub
```

Using Handles

In VB6 and earlier, each event of each object (or control) was unique. When the user clicked a particular button, that button's unique event procedure was triggered. In VB.NET, each event procedure declaration ends with a Handles command that specifies which event *or events* that procedure responds to. In other words, you can create a single procedure that responds to multiple different object events.

To put it another way, in VB.NET an event can "handle" (have code that responds to) whatever event (or multiple events) you want it to handle. The actual sub name (such as Button1_Click) that you give to an event procedure is functionally irrelevant. You could call it Bxteen44z_Click if you want. It would still be the Click event for Button1 no matter what you named this procedure, as long as you provide the name Button1 following the Handles command.

In other words, the following is a legitimate event where you write code to deal with Button1's Click:

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

The `Handles Button1.Click` code does the trick.

[Team Ely](#)

 Previous

Next 


```
        </wellknown>
    </service>
    <cannels>
        <channel
            type= ' 'system.Runtime.Remoting.Channels.Http.HttpChanr
                system.Runtime.Remoting">
        </channel>
        <channel type="system.Runtim.Remoting.Channels.Tcp.TcpChannel
            System.Runtime.Remoting">
        </channel>
    </cannels>
</application>
</system.runtime.remoting>
```

The configuration file shown here registers the class as a remote component with the web server and tells IIS that clients can call the component's members either through an HTTP or a TCP channel. Clients that connect over the Internet and through a firewall will use an HTTP channel, and the data to be exchanged will be serialized in SOAP format. Clients on the same local area network as the web server can make calls using binary serialization, through a TCP port. Notice that you don't have to serialize any objects in your code; this is what remoting does for you. Just remember to "decorate" the class with the `<Serializable>` attribute.

This is all it takes to host the component on a web server through remoting. Build the application and then switch to the client. Make a copy of the NWOrders application (the new project is called RemoteOrders) and revise its code so that it communicates with the database through the methods of the DiscountServer service. Get rid of the two classes that implement the middle tier of the original project to make sure you're not calling a local component by mistake.

First, you must import the following two namespaces to the project:

```
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels
```

Then you must register the remote server with the following statement, which must appear in the form's Load event handler:

```
RemotingConfiguration.RegisterWellKnownClientType( _
    GetType(DiscountServer.NewProduct), _
    "http://localhost/DiscountServer")
```

This statement registers the `DiscountServer.NewProduct` type with the DiscountServer component. It basically tells the CLR that the application will act as a client for a remote service specified by its URL and that the client application will exchange information with the remote server through the objects of the `DiscountServer.NewProduct` class. The class is serializable and the server will be passing to the client application instances of the `ProductPrice` and `OrderedProduct` classes, which are defined in the component's code. You must also add to the project a reference to the DiscountClass. The file to be referenced is the `DiscountServer.dll` file in the ASP application's Bin folder. When you use remoting, both the server and the client must have a reference to the classes they will use to

[Team Fly](#)

 Previous

Next 

exchange data. Web services, on the other hand, do not share this information with their clients. The client of a Web service finds out about the objects recognized by the Web service when you add a Web reference to the project.

The test project's code is almost identical to the original NWOrders project. They differ in the statements that create the instances of the business objects. The following statements create the array with the order's detail lines and then populate it with the items on the grid. Each element of the array is of the OrderedProduct type, which has been registered as a remote server in the Form's Load event handler.

```
Dim orderedItems(lvOrderGrid.Items.Count - 1) As _
    DiscountServer.OrderedProduct
Dim P As DiscountServer.OrderedProduct
Dim i As Integer
For i = 0 To lvOrderGrid.Items.Count - 1
    P = New DiscountServer.OrderedProduct
    P.ProductID = CInt(lvOrderGrid.Items(i).Text)
    P.ProductPrice = CDec(lvOrderGrid.Items(i).SubItems(2).Text)
    P.ProductQTY = CInt(lvOrderGrid.Items(i).SubItems(3).Text)
    P.ProductDiscount = CDec(lvOrderGrid.Items(i).SubItems(4).Text)
    orderedItems(i) = P
Next
```

To submit the order to the database, we create a new instance of the NewProduct class and call its AddOrder method, passing the necessary arguments:

```
Dim INV As New DiscountServer.NewProduct
If INV.AddOrder(lstCustomers.SelectedValue, _
    cmbEmployees.SelectedValue, orderedItems) Then
```

In this section we've presented two techniques that allow you to host your application's middle tier on a remote web server. Both techniques allow you to deploy the middle-tier component on an application server that can service multiple clients, either on the same network or through the Web. Moreover, the middle tier's code can be modified at any time and deployed on a single machine. The client application need not be revised or deployed again to all workstations that make use of the middle-tier components. You should open the NWOrdersWebService and RemoteOrders projects and examine their code. Even though they contact a class on a remote server, their code is almost identical. It's almost the same as the code of the NWOrders application, which is a Windows application that uses a logical middle-tier component. We were able to move the middle tier's components to a server and use it remotely with very few changes.

Using COM Components with .NET Clients

Most large-scale applications written in VB6 make use of COM components, and developers have invested a lot of time and effort in these components. The designers of .NET made sure that the investment in COM won't be wasted and that existing COM components will interoperate with .NET applications. In this section you'll see what it takes to use an existing COM component from within your .NET applications.

[Team Fly](#)

 Previous

Next 

To understand the intricacies involved in calling a COM component from within your .NET application, you must consider the role of the Common Language Runtime (CLR) in developing .NET applications. Code written for the CLR is called managed code, and it requires the CLR for its execution. Some of the unique features the CLR provides are language interoperability and garbage collection. COM components are not aware of the CLR and are written in unmanaged code (that is, any code not written to operate within the CLR). So it seems that a managed application can't call unmanaged code. To overcome this limitation, the designers of .NET decided to create a proxy between managed and unmanaged code. The proxy accepts commands from one component, translates them into a different format, and passes them to the second component. This is how managed code interoperates with COM components.

VB6 developers are quite familiar with proxies. When you configure a COM component to run as a COM+ application on a server, clients connect to it through a proxy. The proxy is installed on the client with an MSI package, which is generated by the Component Services Explorer. The proxy makes the VB6 client think that it talks to a component that runs in the same memory space, while in reality it makes calls to a remote component, which is running on the application server. Something similar happens with the proxies generated by .NET for the COM components. The .NET code thinks that it's talking to another .NET component, while the COM component thinks that it's talking to another COM component. The proxy is a mediator between the two (otherwise incompatible) layers.

The way .NET implements this proxy is through a so-called runtime-callable wrapper (RCW). A RCW is a piece of software that sits between managed and unmanaged code and makes possible the interoperation between the two worlds. COM components contain metadata, which describe their public interface (the signatures of its methods and the types of its properties). One of the tools that comes with Visual Studio is the `tlbimp.exe` tool (Type Library Importer), which reads this data from the COM component and creates an assembly that managed code can use to call the COM component. The proxy is a DLL component that you can reference from within your .NET project. The `tlbimp` tool accepts as arguments the name of the COM component, followed by the `/out` switch and the name of the RCW:

```
Tlbimp Component.dll /out:NETComponent.dll
```

Of course, there's a simpler method to call a COM component from within a .NET application, by simply referencing the COM component from within the .NET application. When you add a reference to a COM component to the project, the corresponding RCW is created automatically. Let's look at the process through an example.

Using ActiveX Controls in .NET

ActiveX controls are COM components, which means you can still use them with your .NET applications. One of the most useful ActiveX controls is the MS Script control, which (strangely) hasn't been replaced with an equivalent purely .NET control. The Script control allows you to evaluate expressions at runtime, and in many situations it's extremely valuable. The Script control understands VBScript and can be used to evaluate math expressions,

manipulate strings and dates, even perform test and repeat operations with the usual loops. Figure 16.9 shows the interface of the COMCalculator project. You can enter any math expression you wish in the top TextBox control and the value of the independent variable X in the second TextBox, then click the Evaluate Expression

[Team Ely](#)

 Previous

Next 

button to calculate the expression for the specified value of x . This is a very simple calculator, but you can't implement something equivalent with straight VB code (at least, not with a few lines of code, as we will do in our application). If you don't mind putting together an elaborate interface, you can build an advanced calculator by exploiting the functionality of the Script control. You can also write functions that perform complicated tasks and execute them by calling a single method. The example of this section is trivial, because our goal is to demonstrate how to interoperate with ActiveX controls from within your .NET applications, but you may wish to take a closer look at the functionality of this control. You can even expose your application's objects and allow developers to program against them with VBScript.

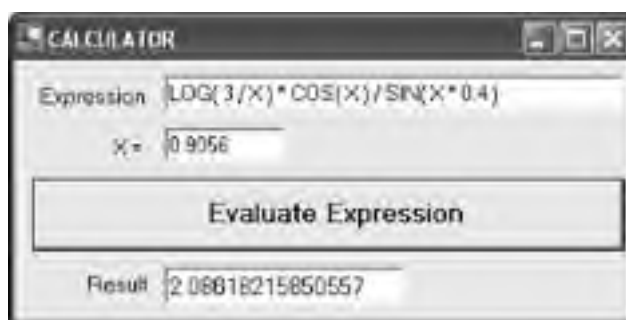


FIGURE 16.9 The COMCalculator is a simple, but very functional, application.

The Microsoft Script control comes with VB6 and there's a good chance that many readers already have this component installed on their systems. If not, you can always download it from Microsoft, at <http://msdn.microsoft.com/scripting> (just follow the link to the downloads).

Start a new project, name it COMCalculator, and then add a reference to the Script control. Open the Project menu and select Add Reference. When the Add Reference dialog box appears, switch to the COM tab and locate the item Microsoft Script Control 1.0 (or whatever the current version will be at the time you download the component). Select the ActiveX control and add a reference to it (see Figure 16.10).



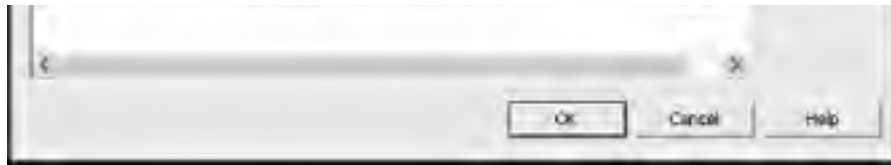


FIGURE 16.10 Referencing a COM component from within a .NET application code

After adding the Script control to the project, switch to the project's Solution Explorer and expand the branch References, where you will see the reference to the component MSScriptControl. Click the Show All Files button and expand the obj item in the Solution Explorer. The following DLL has been added to the application:

```
Interop.MSScriptControl.dll
```

This is the proxy we discussed earlier—it's an interoperability layer. For every reference to a COM component you add to a project, .NET creates a new RCW, which has the same name as the control and is prefixed with the "Interop" string.

PROGRAMMING THE SCRIPT CONTROL

To evaluate an expression with the Script control, you must append the statement that assigns a value to the independent variable with the AddCode method and then call the control's Eval method, passing the expression to be evaluated as argument. Listing 16.6 is the code behind the Evaluate Expression button:

LISTING 16.6: EVALUATING AN EXPRESSION WITH THE SCRIPT ACTIVE X CONTROL

```
Private Sub btnEvaluate_Click(ByVal sender As System.Object, _  
                             ByVal e As System.EventArgs) _  
    Handles btnEvaluate.Click  
    Dim SC As New MSScriptControl.ScriptControl  
    SC.Language = "VBScript"  
    SC.AddCode("X=" & Convert.ToDouble(txtX.Text))  
    Dim result As Double  
    result = Convert.ToDouble(SC.Eval(txtExpression.Text))  
    txtResult.Text = result.ToString  
End Sub
```

The code is quite simple. The expression to be evaluated by the Script control can be as complicated as you wish. The following statements attach a function to the control and then execute it:

```
Dim S As String  
S = "Function Main()" & vbCrLf  
S = S & "    A = InputBox("Enter value of A")" & vbCrLf  
S = S & "    B = InputBox("Enter value of B")" & vbCrLf  
S = S & "    A = Cdbl(A)" & vbCrLf  
S = S & "    B = Cdbl(B)" & vbCrLf  
S = S & "    If A/B > 1 Then" & vbCrLf  
S = S & "        Main = A - B" & vbCrLf  
S = S & "    Else" & vbCrLf  
S = S & "        Main = A + B" & vbCrLf  
S = S & "    End If" & vbCrLf  
S = S & "End Function" & vbCrLf  
SC.AddCode(S)  
SC.ExecuteStatement("MsgBox(Main)")
```

The COMPlus Component

Start VB6 and create a trivial class with a single method that returns the current date. Call the project COMPlus, and in the Class1 class's code window enter the following method:

```
Public Function GetServerDate() As Date
    GetServerDate = Now
End Function
```

Create the executable, which is the COMPlus.dll file. To install the DLL as a COM+ application, switch to the Component Services snap-in (Control Panel ➤ Administrative Tools ➤ Component Services) and expand the branch Component Services ➤ Component Services ➤ Computers ➤ My Computer. Right-click the item COM+ Applications and from the context menu select New ➤ Application to start the COM+ Application Install Wizard. Skip the first introductory screen; on the next screen you'll be prompted to select the type of application you want to create. You can install either a prebuilt application (an MSI package created by the Component Services of another computer) or an empty application. Select the Create An Empty Application button and you'll be prompted to enter the new application's name. Set its name to COMPlusApplication, as shown in Figure 16.11.

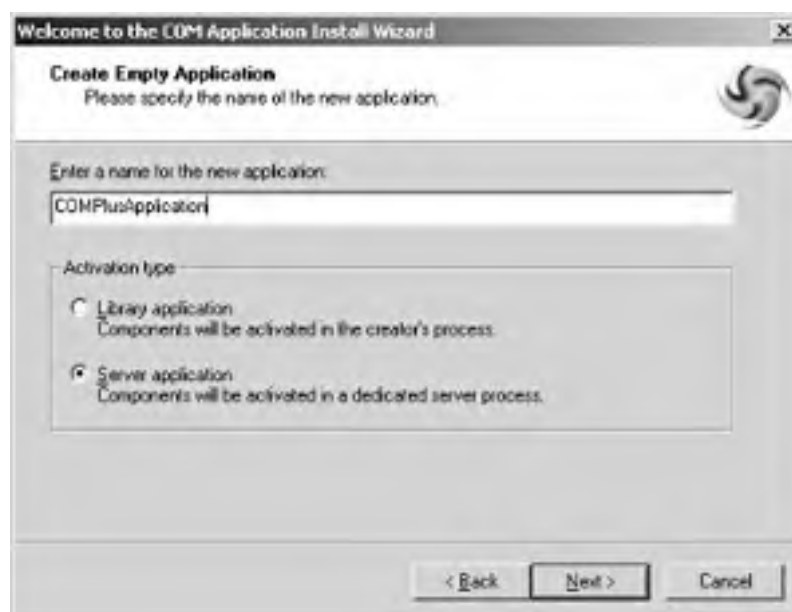


FIGURE 16.11 The Create Empty Application window of the COM Application Install Wizard

In the Activation Type zone you can select where the component will be activated. We want the component to run on the server machine, which is the machine on which the package is created. You can also create a library application. The library application will provide the DLL to the client, but the DLL will be loaded and executed in the same memory space as the calling application (in other words, each client will download the DLL and execute it locally). Click Next and you will see the Set Application Identity window, as shown in Figure 16.12, where you can specify the users who are allowed to execute this package. The default choice

"Interactive User" means the user who has requested the services of the COM+ component (that is, the user who started the client application). Normally, we create a new account for the users whom we want to execute the component remotely (an account such as AccountingUser, for example) and we make sure that the clients log on with this user ID. For the purposes of this sample, you need not make any changes in this window. Click Next one last time to create the COMPlusApplication application.

[Team Fly](#)

 Previous

Next 



FIGURE 16.12 The Set Application Identity window of the COM Application Install Wizard

If you switch now to the Component Services window, you'll see that you've created a new COM+ application, but it contains no components. We haven't specified yet the component to be hosted by the newly created COM+ application. Double-click the Components item under COMPlusApplication to open it and then right-click the opened item. From the context menu select New ➤ Component. This will start the COM+ Component Install Wizard, which will take you through the steps of installing a new component to the application. Skip the first welcome window; on the following one you'll be prompted about the type of component you want to install: a new component, a registered component, or a new event class. Select the first option and a File Open dialog box will appear, on which you can select the DLL with the component to add to the application. Locate the COMPlus.DLL and click OK to select it. The wizard will add the component to the COM+ application; you can right-click the component's name in the Component Services and view the properties of the new component.

Open the component's property pages and switch to the Activation tab. The Object Pooling section in this tab is disabled. VB6 components can't take advantage of object pooling, and this is a topic we'll discuss in some detail a little later in this chapter.

Exporting a Proxy and Testing It

So far you created a COM+ application that hosts a simple class. Now we're going to use this component from a remote machine. We'll set up a .NET client application that calls the methods of the COMPlus component from another computer. The COM component will be executed on the machine on which it's hosted. For example, the component may be able to access a database to which no client has access. Or it may perform operations on behalf of the client that require resources available only on the application server. Presumably, VB6 developers have a number of COM components that they will keep using with .NET. Many of these components may already be deployed as COM+ applications, and as you will see, you

can use them immediately.

To access a COM+ application's services from a remote computer, you must install the corresponding proxy on the client computer. The proxy is a DLL (basically, another COM+ application registered at the client), which talks to the client application running on the same computer and to the COM+ application's components on the remote server. In simple terms, it takes care of the plumbing between the two components, making them think that they talk directly to one another—which in turn simplifies our programming effort.

To create the proxy, right-click the COM+ application on the server and from the context menu select Export to start the Application Export Wizard. This wizard will generate an MSI package that can be used to install the COM+ application to another computer, either as a server application or as a proxy. Specify the path where the MSI package will be created, check the radio button Application Proxy, and click the Next button on the window shown in Figure 16.13. The wizard will generate an MSI package that you can use to install the component's proxy on a client. Copy the MSI file to another machine and run it to install the COM+ application's proxy.



FIGURE 16.13 Generating an MSI package for installing the component's proxy on the client computers

Now you can write a client to test your proxy. Start a new Windows application, add a reference to the component's proxy, and use it in your code. To add the corresponding reference to the client project, open the Add Reference dialog box, switch to the COM tab, and locate there the COMPlus component. After that, you can create instances of the COMPlus component in the client application's code and call its methods.

Building Serviced Components with .NET

COM+ applications were quite popular with VB6, so we should be able not only to reuse existing COM+ applications with our .NET applications, but to create new COM+ applications in managed code. Indeed, it's possible to build COM+ applications with .NET and take advantage of all the COM+ services. Let's start with an overview of the COM+ services. This will be a quick overview of COM+ for readers who have used COM+ with VB6 and an introduction to the EnterpriseServices class. The most important of the COM+ services are the following:

Automatic Transaction Processing This service allows you to perform

transactions through the Context object. Database and messaging transactions are committed or rolled back automatically. The big advantage of performing transactions through COM+ is that it simplifies transactions against multiple databases. To perform this type of transactions from a client application, you need to establish connections to multiple databases from every client (something that's neither practical nor very efficient).

Just In Time Activation (JITA) Activates an object when a method is called and deactivates it as soon as the method returns.

Object Pooling Maintains a pool of objects and client applications are given access to one of the existing objects, as opposed to creating new objects as needed. You have control of the minimum and maximum number of objects maintained in the pool, and you can control your server's load.

Queued Components A queued component provides asynchronous processing. Requests are queued and serviced according to the component's schedule. We will not discuss queued components in this chapter, but the idea is similar to passing messages to a queue (as discussed in Chapter 14).

Role-Based Security Applies security policy to a component based on a role. You can assign roles to the users and service (or deny) requests based on the user's role in the application.

COM+ interoperates with .NET, but there are no Framework classes specific to COM+. The functionality of COM+ in .NET is implemented with the `EnterpriseServices` namespace, which we'll explore in the following section. The components that will be hosted in COM+ are no longer called COM+ applications: their new name is *serviced components*. Other than a few differences in terminology and a different approach in building COM+ applications, there are no basic differences between the two. The `EnterpriseServices` class exposes the functionality previously available to COM+ components.

To demonstrate the process of building serviced components, we'll use a very simple example that demonstrates how to take advantage of object pooling. This feature is new to VB.NET—the object pooling service was not available with VB6 COM components.

OBJECT POOLING

Object pooling is a simple concept that can become invaluable in building scalable applications. We will assume that certain components must reside on an application server and these components are requested by a large number of clients. Let's also assume that some of these objects are expensive to set up (they take a lot of CPU time, or use system resources excessively). Setting up a connection to a database is a typical expensive operation. Setting up a database connection is a necessary operation, but it's not the kind of operation we want to perform repeatedly. A query, on the other hand, is an expensive operation, but a useful one. Establishing a connection to the database is only a prerequisite for executing the query, and we'd like to minimize the cost of the new connection.

Object pooling allows us to maintain a pool of objects that are ready to be reused. Every time a client needs a connection, we can retrieve a `Connection` object from our pool and assign it to the client, which can proceed by executing a query without having to wait for a new connection to be established. Enterprise Services allows us to specify the minimum and maximum number of objects in the pool. If we determine that the average load on the application server is 10 concurrent clients, we can create 10 or 12 objects and place them in the pool. These objects will remain alive in the pool, even if no client is using them. Under normal circumstances, there will be an object for each client that requests it.

If the server accepts more requests than the available objects, new ones will be created automatically and assigned to the client applications. However, these objects won't be added to the pool. They will be released as soon as they're no longer needed. We may also determine that the server can't gracefully

[Team Ely](#)

 Previous

Next 

You can even gang up several events into one, like this:

```
Private Sub cmd_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles cmdOK.Click, cmdApply.Click, cmdCancel.Click
```

For a working example of this ganging up, see the section in this chapter titled "Multiple Handles."

Runtime Handles

If you want to get really fancy, you can attach or detach an object's events to a procedure during run-time. The following example illustrates this.

Put three Buttons on a form. Now create this procedure to handle all three of these Buttons' Click events:

```
Private Sub AllTheButtonClicks(ByVal sender As Object, _
ByVal e As System.EventArgs) Handles Button1.Click, Button2.Click, Button3.Click
    RemoveHandler Button1.Click, AddressOf AllTheButtonClicks
    MsgBox(sender.ToString)
End Sub
```

Press F5 to run this little program and click Button1. The MessageBox appears. Click Button1 again and *nothing happens* because Button1's Click event has been detached—so this procedure no longer responds to any clicking of Button1. Button2 and Button3 still trigger this procedure. To restore (or create for the first time) a procedure's ability to handle an object's events, use the AddHandler command, like this:

```
AddHandler Button1.Click, AddressOf AllTheButtonClicks
```

Detecting Key Presses

The following example show how to use the KeyChar property to figure out which key the user pressed. You compare it to the Keys enumeration (a list of built-in constants that can be used to identify keypresses). The following code checks to see if the user presses the Enter key:

```
Private Sub TextBox1_KeyPress(ByVal sender As Object, _
ByVal e As System.Windows.Forms.KeyPressEventArgs) Handles TextBox1.KeyPress
    If Asc(e.KeyChar) = Keys.Enter Then
        MsgBox("ENTER")
    End If
End Sub
```

The following example shows how to detect if the user presses Ctrl+N on the form (to set the Form's KeyPreview property to True):

```
Private Sub Form1_KeyDown(ByVal sender As Object, _
ByVal e As System.Windows.Forms.KeyEventArgs) Handles MyBase.KeyDown
```


handle more than 50 concurrent sessions. In other words, maintaining 50 concurrent sessions is a hard upper limit for our server, and it's better to have some clients wait for an object to become available in the pool than to create objects. The 51st client will have to wait for an object to become available (that is, for another client application to free one of the objects). If you don't set the maximum number of objects in the pool, the delay will be spread among the clients. There's a cutoff point, beyond which we'd rather delay an isolated client than keep increasing the delay for all clients. In effect, by controlling the maximum number of objects in the pool we control the amount of resources consumed by these objects. The exact value of the minimum and maximum number of objects in the pool is determined empirically and depends on the application, the minimum/average/maximum client load, and the application server. A server with multiple processors can better handle components that execute a load of statements. If your components set up complicated data structures in memory, they can be handled more efficiently by a server with many gigabytes of memory.

IMPLEMENTING POOLING

Let's look at the process of building a pooled component. Every component that must be installed as a COM+ application must inherit from the `ServiceComponent` class and must reference the `EnterpriseServices` class. It must also have a strong name, like a regular COM+ component. Adding a strong name to an assembly is a straightforward process, and you'll see shortly how to create a strong name and then use it in a .NET assembly. Finally, you must assign attributes to the class and its members. These attributes will allow the CLR to create the proper COM+ application for your component. It's also possible to specify these attributes in the assembly's configuration file, or set them manually through the Component Server Explorer.

Start a new Class Library project and name it `PooledServer`. Then add to the project a reference to the `System.EnterpriseServices` namespace. Before writing any code, you should prepare a strong name for the application. If you have created a strong name you're using with your applications already, you can reuse it. To create a new strong name, open the Visual Studio Command Prompt and switch to the application's folder. You can create the file with the strong name in any folder, but then you must copy it to the new application's folder. Enter the following statement in the Command Prompt window:

```
sn -k ServerKey.snk
```

The `sn` command line tool will create the `ServerKey.snk` file in the current folder. This file contains a public and a private key and must be referenced by the component that must be assigned a strong name. Copy the `ServerKey.snk` file to the project's `Bin` folder. To reference the file with the keys in your component, add the following statement to your class:

```
<Assembly: AssemblyKeyFile('..\..\ServerKey.snk')
```

Finally, you must decorate the class's definition with the `ObjectPooling` attribute, in which you can specify the minimum and maximum size of the pool, as well as the setting of the `JustInTimeActivation` option:

<ObjectPooling (MinPoolSize:=3, MaxPoolSize:=12), JustInTimeActivation(True)>

[Team Fly](#)

 Previous

Next 

To demonstrate how object pooling can help the performance of a client application, we'll implement a constructor that takes 10 seconds to create an instance of a new object and another method that returns the name of the computer. We've chosen this return value for our sample method so that we can verify on which computer the component is running (should you install it on an application server). If you're using a single machine to test the pooled component, you'll see the difference in creating new objects, but the computer name will always be the same. Listing 16.7 shows the code of the PooledServer application:

LISTING 16.7: THE POOLEDSEVER CLASS'S IMPLEMENTATION

```
Imports System.EnterpriseServices

<ObjectPooling (MinPoolSize:=3, MaxPoolSize:=12), _
    JustInTimeActivation (True)> _
Public Class ExpensiveObject
    Inherits EnterpriseServices.ServicedComponent

    Protected Overrides Function CanBePooled() As Boolean
        Return True
    End Function

    Public Sub New()
        System.Threading.Thread.CurrentThread.Sleep(10000)
    End Sub

    Public Function GetComputerName() As String
        System.Threading.Thread.CurrentThread.Sleep(1000)
        Return Windows.Forms.SystemInformation.ComputerName
    End Function
End Class
```

Compile the class to create the PooledServer.dll in the application's Bin folder. The DLL isn't registered with COM+ automatically, but its attributes determine the settings for its proper configuration. To actually register the new component with the COM+ catalog, open a Command Prompt window, switch to the location of the DLL file, and execute the following command:

```
regsvcs PooledServer.dll
```

This command will register the new component with the COM+ catalog and you will be able to use it from within client applications. If you open the Component Services window now, you'll verify that a new COM+ application was created automatically—the PooledServer application—and that it hosts a single component—the ExpensiveObject component—as shown in Figure 16.14. Select the ExpensiveObject component and open its property page. Switch to the Activation tab, as shown in Figure 16.15, and you will see that the pooling parameters we specified with the appropriate attributes in our code have taken effect.

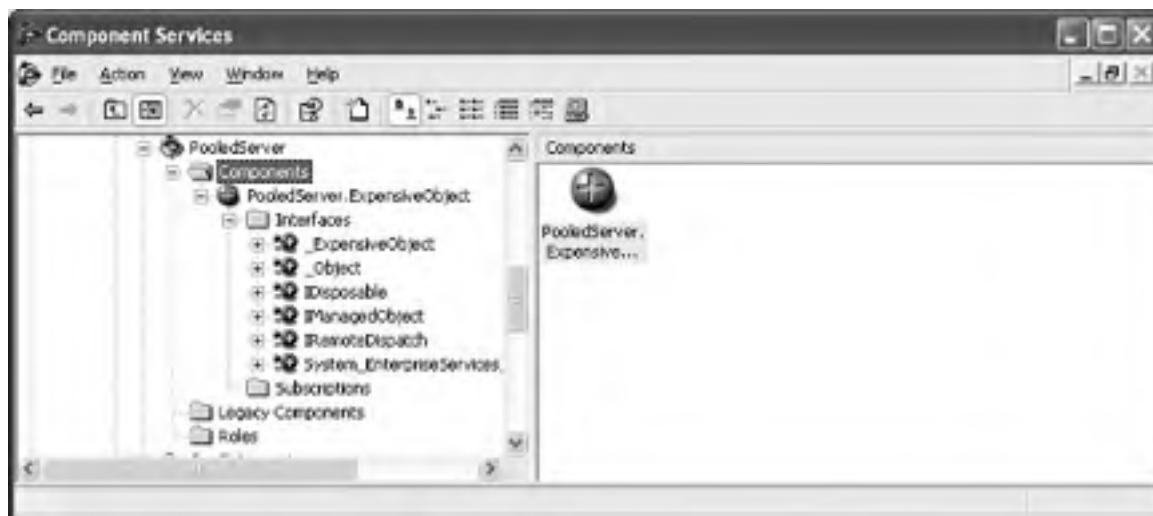


FIGURE 16.14 Adding a new application to the COM+ catalog with the regsvcs.exe tool



FIGURE 16.15 Use the Activation tab of the component's Property pages to change the object pooling parameters of an existing serviced component.

Now we're ready to test our new serviced component. Start a new project in Visual Studio—this time, a Windows application that will call the services of the PooledServer component. Name the new project PooledClient and add to it a reference to the PooledServer.dll component. Then enter the statement in Listing 16.8 in a button's Click event handler.

LISTING 16.8: TESTING THE POOLEDSERVER SERVICED COMPONENT

```
Imports System.EnterpriseServices  
Public Class Form1
```

```
Inherits System.Windows.Forms.Form
Private Sub Button1_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
    Handles Button1.Click
    Dim P As New PooledServer.ExpensiveObject
    MsgBox(P.GetComputerName)
    ServicedComponent.DisposeObject(P)
End Sub
End Class
```

This page intentionally left blank.

```
    If Xreader.nodetype = XmlNodeType.Element Then
        ele += 1
        If (Xreader.HasAttributes) Then
            att += Xreader.AttributeCount
        End If
    End If
End If

End While

m &= "Number of Elements: " & ele
m &= " Number of Attributes: " & att
MsgBox(m)

End Sub
```

This code creates a little XML parser. The Microsoft version of a SAX reader, the `XMLReader` object, is similar to the traditional SAX parser, but it's more flexible. You can customize the `XMLReader`'s behaviors. The reader's `.Read` method is quite versatile (though it cannot *modify* the XML stream by itself).

.NET offers several techniques you can use to read, search, or parse XML. In addition to the `XMLReader` class, you can use the .NET XPathNavigator API, which offers a cursor-based technology useful for exploring XML stores, in addition to supporting Xpath queries.

In the previous example, we asked for information on elements and attributes, while ignoring much of the other information this method reports. Here are the other `NodeTypes` the reader can tell you about: `CDATA`, `comment`, `Document`, `DocumentFragment`, `DocumentType`, `EndElement`, `EndElement`, `Entity`, `EntityReference`, `Notation`, `ProcessingInstruction`, `SignificantWhitespace`, `Text`, `Whitespace`, `XmlDeclaration`.

Following is a list of the properties of the `XMLReader` object that provide node information:

AttributeCount How many attributes are contained within the current node.

BaseURI The base URI. Duh.

Depth The depth of the current node (how many parent-child nests down it is) within the document's structure.

HasAttributes A Boolean (true or false) value specifying whether the node has any attributes.

HasValue A Boolean value specifying whether the node has any "value" (contents, such as text or numeric data).

IsDefault A Boolean value signifying whether the node's value is the default value specified in a DTD or XSD file.

IsEmptyElement A Boolean value indicating whether the node contains no data.

LocalName The name of the current node—but leaves off any namespace prefix.

Name The "fully qualified" name of the node—including any namespace prefix.

self-describing tags (understood by programmers) to self-describing data structures (understood by programs themselves, without the direct aid and translation of a programmer).

Table 17.1 contains the primary classes in the .NET DOM implementation. Table 17.2 contains the extended classes.

TABLE 17.1: THE .NET DOM FUNDAMENTAL CLASSES

| CLASS | DESCRIPTION |
|------------------------------|--|
| <code>XmlNode</code> | A node in an XML document |
| <code>XmlNodeList</code> | A collection of nodes |
| <code>XmlNamedNodeMap</code> | A collection of nodes that can be accessed directly from code using their node names |

TABLE 17.2: THE .NET DOM EXTENDED CLASSES

| CLASS | DESCRIPTION |
|---------------------------------------|--|
| <code>XmlDocument</code> | Represents an entire XML document (the top node and all the child nodes) |
| <code>XmlAttribute</code> | Represents an XML attribute |
| <code>XmlAttributeCollection</code> | A collection of attributes (all attributes are associated with a single XML element) |
| <code>XmlCDATASection</code> | A CDATA section in a document |
| <code>XmlCharacterData</code> | Offers several text-manipulation methods |
| <code>XmlComment</code> | An XML comment |
| <code>XmlDeclaration</code> | An XML declaration |
| <code>XmlDocumentFragment</code> | A section from a full XML document (not a complete document) |
| <code>XmlDocumentType</code> | A DOCTYPE declaration |
| <code>XmlElement</code> | An XML element |
| <code>XmlEntity</code> | An <code><!ENTITY ... ></code> declaration |
| <code>XmlEntityReference</code> | An entity reference node |
| <code>XmlImplementation</code> | A definition of the context for a set of <code>XmlDocument</code> objects |
| <code>XmlLinkedNode</code> | A node just before (such as a parent) or after (such as a child) the currently referenced node |
| <code>XmlNotation</code> | A notation declaration in a DTD or XSD file |
| <code>XmlProcessingInstruction</code> | A processing instruction |

`XmlSignificantWhitespace` Whitespace in an element or attribute

`XmlText` The value (the text content) of an element or attribute

[Team Fly](#)

 Previous

Next 

A schema is a structured framework, or diagram, that is meant to clarify and describe a set of related ideas. A blueprint, for example, is a schema that describes all the elements in what will be a house.

A schema is metadata—data about data. The metadata, or schema, for a database is the names and relationships between tables, fields, records, and properties of that database. Similarly, an XML document can be thought of as another kind of database. And an XML schema is a description of the structure of an XML document. If you own a grocery store, you can use XML to communicate your inventory needs to your suppliers as long as both of you are using the same XML schema to ensure conformity and, thus, comprehensibility.

VB .NET contains a tool, the XML Designer, which you can use to build and modify XML schemas and the resulting XML documents. This designer shows you both source code and diagrams of a schema.

Microsoft has selected XSD as its favored schema builder for several reasons. XSD permits you to group elements and attributes (to assist you in repeating the groups); it contains KEY and KEYREF statements which allow you to create one-to-many relationships and uniqueness constraints; it supports inheritance, namespaces, and extensibility (not simple XML tag extensibility; rather, the extension of a schema itself); and, last but not least, XSD employs XML to define the XSD schemas (so it has the benefits that XML offers over other data definition techniques).

Here's an example of a typical XSD schema. Notice that there is no actual data content here (no proper names such as *London St.* or numeric values such as a phone number). Instead, you have a list of elements (tags), their data type, and their structural relationship within the sequence. This is quite like creating an array or type of declared variable names:

```
Dim FirstName as String
Dim LastName as String
Dim ZipCode as Integer
Dim Address as String
```

Here's the "declaration" of a complex data type using XSD:

```
<?xml version="1.0" encoding="utf-8"?>
<schema targetNamespace="http://CardCorp.com/XMLSchema1.xsd" _
xmlns="http://www.Nams.com/Ars/XMLSchema2">
<complexType name="customerType">
  <sequence>
    <element name="LastName" type="string"/>
    <element name="FirstName" type="string"/>
    <element name="StreetAddress" type="string"/>
    <element name="City" type="string"/>
    <element name="State" type="string"/>
    <element name="ZipCode" type="integer"/>
  </sequence>
</complexType>
</schema>
```

The line that begins with `?xml` describes which version of XML this document uses. The

following line, beginning with `<schema`, describes which XSD schema file is being used and also specifies the namespace.

[Team Fly](#)

 Previous

Next 

Using XML Data Types

The type of an element or attribute can be either one of the standard types defined by the World Wide Web Consortium (W3C), or can be simple or complex types that you defined earlier in your schema.

Some programmers prefer to use only the string type, and then in applications convert the string to other types, as necessary. If you want to use data types supported by Internet Explorer 5 and above, include the following datatypes namespace:

```
<schema name="'YourSchema"  
    xmlns="urn:schemas-microsoft-com:xml-data"  
    xmlns:dt="urn:schemas-microsoft-com:datatypes">  
<!-- define your schema here -->  
</schema>
```

You are allowed to specify data types for attributes, as well as elements; however, with attributes there are fewer permitted types. The types permitted for use with attributes are: string, id, idref, idrefs, nmtoken, nmtokens, entity, entities, enumeration, and notation. For example:

```
<AttributeType name="Overdue" dt:type="idref"/>  
<attribute type="Overdue" />
```

If an attribute has an idref data type, that attribute contains a unique identifying value in the document (just as the Name property of a VB control must be unique within a given WebForm, or the ID in an HTML document must be unique in that document). However, there are two similar XML data types in a schema: id and idrefs.

Attributes with the type id are references to the *element* that uses the same id. Idrefs is like id, but idrefs contains a list of ids separated by spaces:

```
<LateCharge creditcard="First Second Third">
```

Note that elements only support the id attribute data type in Internet Explorer 5.01 and above:

```
<ElementType name="Factory">  
    <datatype dt:type="id">  
</ElementType>
```

DECLARING SIMPLE XML DATA TYPES

To declare simple (boolean, float and such) data types for attributes in your schema, just use the <datatype> element within an <AttributeType> element, like this:

```
<AttributeType name="MyAttributeType"/>  
  <datatype dt:type="int"/>  
</AttributeType>  
<ElementType name="Points">  
  <attribute type="MyAttributeType"/>  
<ElementType>
```


SPECIFYING CONTENT IN AN XML SCHEMA

You can use previously defined `ElementTypes` to build a new, more complex `ElementType`, as shown in Listing 17.2.

LISTING 17.2: BUILDING A COMPLEX ELEMENTTYPE

```
<schema xmlns="http://www.Nams.com/Ars/XMLSchema2">

  <ElementType name="LastName" />
  <ElementType name="FirstName" />
  <ElementType name="StreetAddress" />
  <ElementType name="City" />
  <ElementType name="LastName" />

  <ElementType name="Voter" order="seq">
    <element type="FirstName" />
    <element type="LastName" />
    <element type="StreetAddress" />
    <element type="City" />
  </ElementType>

</schema>
```

In this example, we built the `Voter` element out of four previously defined elements. We also used `<seq>` to specify that the four elements must be in sequence. XML that correctly follows the previous schema looks like this:

```
<Voter xmlns="Myschema:Voter-schema.xml">
  <FirstName>Jon</FirstName>
  <LastName>Popodoupolous</LastName>
  <StreetAddress>922 W. Archer St.</StreetAddress>
  <City>Bogotoa</StreetAddress>
</Voter>
```

EXTENDING A SCHEME

Schemas can be freely extended (you can add new elements and attributes that go beyond what is defined in the schema) as long as you follow a few rules. This means that a schema's content model is by default *open*, though you can force it to be closed if you wish by using the `model="closed"` attribute.

If you want to extend an open content model, you must remember that you can only add new, undeclared elements if they are in a new namespace. For example, here we add a new namespace with the prefix `n`:

```
<Voter xmlns= xmlns="Myschema:Voter-schema.xml" xmlns:n="urn:AnotherNamespace">
```

Now we can add an attribute from the *n* namespace, like this:

```
<FirstName>Jon</FirstName>
<LastName n:Alpha = 'P'>Popodoupolous</LastName>
```

or add a new element from the *n* namespace, like this:

```
<StreetAddress>922 W. Archer St.</StreetAddress>
<City>Bogotoa</StreetAddress>
<n:Country>Chile</n:Country>
</Voter>
```

Be warned that you can't add (or delete) any content from the model that would violate the content model rules. Recall that in the original schema we used the `order="seq"` attribute to require that the `FirstName`, `LastName`, `StreetAddress`, and `City` elements be sequential. You *are* allowed to add new attributes to these elements, but you are not allowed to violate the sequence of the elements. That is, you cannot remove one of the required four elements, or insert a new element above or within the sequence. The `<Voter>` element must begin with the four elements, though you can add additional elements *below* those four elements. Also, if you want to add more `FirstName` elements to `<Voter>` (to accommodate people like Prince Charles Albert James etc. etc. of England), you can do it, but these new `FirstName` elements must be appended to the sequence. In other words, you must add new `FirstName` elements after the `<City>` element. Otherwise you would violate the sequence as defined in the schema.

If for some reason you want to freeze a content model, and not permit any extensibility, simply use the following attribute in the schema:

```
<ElementType name="Voter" order="seq" model="closed">
```

With this directive in place, any added or deleted elements—any changes to the original schema, will not validate. It doesn't matter if you add new namespaces. Closed means closed.

The `order` attribute, as you've seen, can be used with the `seq` value in a schema to freeze a sequence of elements. You can also use the `one` value with the `order` attribute to require that only one single subelement (an element within another element) must be used from a list of possible subelements. For example, you might want to specify that only one of the following elements can be used—`<ZipCode>` or `<CountryCode>` but not both. Here's how:

```
<ElementType name="Code" order="one">
  <element type="ZipCode" />
  <element type="CountryCode" />
</ElementType>
```

But if you want to go in the other direction and throw all caution to the winds, use the `many` value with the `order` attribute. The `many` value says that there can be any number of subelements and they can appear in any order as well.

Sometimes you may need to specify an order on *some* of the subelements, but want to leave the rest of the subelements unaffected by the order rule. To do that, use the group element. For instance, we

could isolate the ZipCode or CountryCode elements by forcing them to be either/or, but not extending this rule to other subelements:

```
<ElementType name='Code'>
<group order="one">
  <element type="ZipCode" />
  <element type="CountryCode" />
</group>
  <element type="PhoneNumber" />
  <element type="Age" />
</ElementType>
```

In this example, either ZipCode or CountryCode will be used, but not both. However, Phone-Number and Age will both be required.

The group element has two other possible attributes: minOccurs and maxOccurs. These attributes define how often a given subelement can appear within a container element. You can specify max-Occurs using an integer (`maxOccurs="5"` means that no more than five of this subelement may appear) or an asterisk to indicate that there can be unlimited numbers of the subelement:

```
<element type="Voter" maxOccurs="*" />
```

The default value for maxOccurs is 1. But if the `content="mixed"` then the default value for maxOccurs is "*".

MinOccurs defines the minimum number of times that a given subelement can appear. MinOccurs defaults to 1, but if you set it to "0" the inclusion of that subelement is then optional.

USING THE CONTENT ATTRIBUTE

Elements can contain other elements (called subelements), and/or text, or simply be empty of content. You can use the values `textOnly`, `eltOnly`, `mixed`, and `empty` as values for the `content` attribute of an element—these values specify the permitted content for that element.

If you want to insist that an element contain text, but can contain no other elements (nor be empty), use the `textOnly` value:

```
<ElementType name="FirstName" content="textOnly"/>
```

If you use the `empty` value, no text or subelements are permitted. The `mixed` value permits you to use both text and subelements. Or you can specify that an element *must* contain subelements, but no other content; use this variation (*elt* is short for element):

```
<ElementType name="Voter" content="eltOnly"/>
```

TIP *If you've defined a datatype for an element, then textOnly becomes the default content specification.*

If the content attribute is eltOnly, then the order value defaults to seq. If the content attribute is mixed, then the order value defaults to many.

[Team Fly](#)

 Previous

Next 

Edit and Save

You must be able to programmatically edit XML, and save the results back to an XML store. In the following examples, you see how to delete a particular attribute or element throughout a document, as well as how to replace or add individual elements or entire nodes.

DELETING ATTRIBUTES

In the next example (Listing 17.4), you decide you want to delete all the id attributes from the book elements. You simply loop through the `XmlElement`s, stopping at each child node (in other words, all nodes within the general, outer mode node `<catalog>`). For each child node, you remove any attribute named id.

LISTING 17.4: DELETING AN ATTRIBUTE THROUGHOUT A DOCUMENT

```
Private Sub Form1_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles MyBase.Load  
  
    Dim Xdoc As New XmlDocument  
    Xdoc.Load("c:\books.xml")  
  
    Dim XElement As XmlElement  
  
    For Each XElement In Xdoc.DocumentElement.ChildNodes  
        XElement.RemoveAttribute("id")  
    Next  
  
    Dim writer As XmlTextWriter = New XmlTextWriter _  
        ("c:\bookschanged.xml", Nothing)  
    Xdoc.Save(writer)  
  
    End  
  
End Sub
```

After you execute this code, the new file, `c:\bookschanged.xml`, will contain the same XML, but with the id attributes stripped out.

DELETING ELEMENTS

Here's similar code that removes elements rather than attributes:

```
For Each XElement In Xdoc.DocumentElement.ChildNodes  
    Dim xmlList As XmlNodeList = XElement.GetElementsByTagName("author")
```

```
        For i = 0 To xmlList.Count - 1
            XElement.RemoveChild(xmlList(0))
        Next
    Next
```

REPLACING ELEMENTS

To replace elements, use code like this:

```
Dim root As XmlNode = myXMLdoc.DocumentElement
'Create a new node.
Dim XElement As XElement = myXMLdoc.CreateElement("price")
Xelement.InnerText = "99.99"
'Replace the first element (book ID = 101) with this new element
root.ReplaceChild(Xelement, root.FirstChild)
```

The DocumentElement property represents a document's root element, which is often the entire document itself. The CreateElement constructs a new node; you can create a new XML element simply by assigning a name ("price" here) and including any InnerText you care to add. The ReplaceChild method replaces the first <book> element with this new <price> element.

ADDING NEW ELEMENTS

Use the InsertAfter or InsertBefore methods to add new elements. Change the ReplaceChild method from the previous code example:

```
root.InsertAfter(Xelement, root.FirstChild)
```

The DOM includes four methods you can use to add new elements: InsertAfter, InsertBefore, AppendChild, PrependChild.

ADDING AN ENTIRE NODE

In Listing 17.5, you see how to add a whole node, all at once, to an existing document.

LISTING 17.5: ADDING A NODE TO A DOCUMENT

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

    Dim XDoc As New XmlDocument
    XDoc.Load("c:\books.xml")

    ' Create a new Book element
    Dim XElement As XElement = XDoc.CreateElement("book")
    ' Append this new element to the collection of children of th
```

```
' root element which is <catalog>.
XDoc.DocumentElement.AppendChild(XElement)

' Specify the ID attribute for the new node.
XElement.SetAttribute("id", "2009")

' Build the sub-elements that belong within a <book> element.

Dim XChildElement As XmlElement

' Create the author sub-element.
XChildElement = XDoc.CreateElement("author")

'Add the text (the value) to this element,
'and append it to the new <book> element:
XChildElement.AppendChild(XDoc.CreateTextNode("Salifs, Sally")

'Append this to the new book element.
XElement.AppendChild(XChildElement)

' Repeat the above to create all the rest of the sub-element

XChildElement = XDoc.CreateElement("title")
XChildElement.AppendChild(XDoc.CreateTextNode("Seems to Me!"))
XElement.AppendChild(XChildElement)

XChildElement = XDoc.CreateElement("genre")
XChildElement.AppendChild(XDoc.CreateTextNode("Pointless Self
XElement.AppendChild(XChildElement)

XChildElement = XDoc.CreateElement("price")
XChildElement.AppendChild(XDoc.CreateTextNode("$4.95"))
XElement.AppendChild(XChildElement)

XChildElement = XDoc.CreateElement("publish_date")
XChildElement.AppendChild(XDoc.CreateTextNode("2003-10-03"))
XElement.AppendChild(XChildElement)

XChildElement = XDoc.CreateElement("description")
XChildElement.AppendChild(XDoc.CreateTextNode("This book is i
XElement.AppendChild(XChildElement)

XDoc.Save(Console.Out)
End
End Sub
```


A Recursive Walk through the Nodes

Going down through a tree structure such as an XML document, you may find that you have to examine each node, see if it has any children, or children of the children, and so on down as far as necessary. Sounds like a job for recursion.

In this next example, you walk through the sample XML document, and can, therefore, manipulate each element or attribute as you wish (using some of the methods described earlier in this chapter, for example). If you require fine-grain management of an XML document, this is one way to do it. You could even use conditional logic, such as: "If this document includes no date stamp anywhere, add one."

Add a TextBox to your form. Using the Properties window, change the TextBox's MultiLine property to True and its ScrollBars property to Vertical. Then type Listing 17.6 into your code window.

LISTING 17.6: WALKING THROUGH AN XML DOCUMENT

```
Dim s As String
    Dim cr As String = ControlChars.CrLf
    Dim quot As String = ControlChars.Quote

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

        Dim Xdoc As New XmlDocument
        Xdoc.Load("c:\books.xml")

        ShowElements(Xdoc) 'pass the entire document to the recursive

        TextBox1.Text = s 'display the whole, finished string
        TextBox1.SelectionLength = 0 'turn off the selection

    End Sub

    Sub ShowElements(ByVal XNode As XmlNode)

        Static c As Integer
        c += 1 'increment counter each time through the recursion

        If XNode.Name = "book" Then _
            'major node
                s &= "Major
```

```
Parent Node _____ ' ' & cr
          s &= c & ". Name of this Node: " & XNode.Name & cr

      ElseIf XNode.Name <> "#text" Then _
' show the name of the other children except text
          s &= c & ". " & XNode.Name & cr
      End If

      Select Case XNode.NodeType
      Case XmlNodeType.Element
          ' Any attributes inside this element?
          If XNode.Attributes.Count > 0 Then 'yes
              s &= ".....Attributes:"

              Dim XAttribute As XmlAttribute

              ' show each attribute
              For Each XAttribute In XNode.Attributes
                  s &= XAttribute.Name & _
" = " & quot & XAttribute.Value & quot & cr & cr
              Next
              End If
          Case XmlNodeType.Text 'show only the contents of a text r
              s &= c & ".          " & XNode.Value & cr
          Case Else
              ' ignore other types of elements
      End Select

      ' Keep calling this same ShowElements Sub
      ' until you run out of child elements:
      Dim XChild As XmlNode = XNode.FirstChild
      Do Until XChild Is Nothing
          ShowElements(XChild) 'call yourself
          XChild = XChild.NextSibling 'move further
      Loop
      End Sub
```

Execute this code and you see the entire XML document atomized into all its components, as shown in Figure 17.1. This is an excellent way to manage XML documents on the lowest level.

This low-level access to XML permits you to write programming that adjusts the content (you can freely edit, delete, add, or replace components) or displays the contents of an XML document in any fashion you choose, for reports. In this report, each <text> element's name (#text) is not displayed, but the value of these text elements is shown (via the Select Case structure). All the other child elements are listed as name/value pairs, with the value tabbed over a bit to make the values easy to distinguish from the element names.

LISTING 17.7: PERSISTING AND RETRIEVING STORED DATASETS

```
Imports System.IO

Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    'Save data as XML

    Dim fStream As Stream
    fStream = New FileStream("C:\XMLData", FileMode.OpenOrCreate)
    XmlSchema11.WriteXml(fStream)

    fStream.Close()

End Sub

Private Sub Button2_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button2.Click
    'load the XML data

    Dim fStream As Stream
    fStream = New FileStream("C:\XMLData", FileMode.Open)
    XmlSchema11.Clear()
    XmlSchema11.ReadXml(fStream)

    fStream.Close()

End Sub
```

Now test all this by running the program and typing in some data, as shown in Figure 17.2:

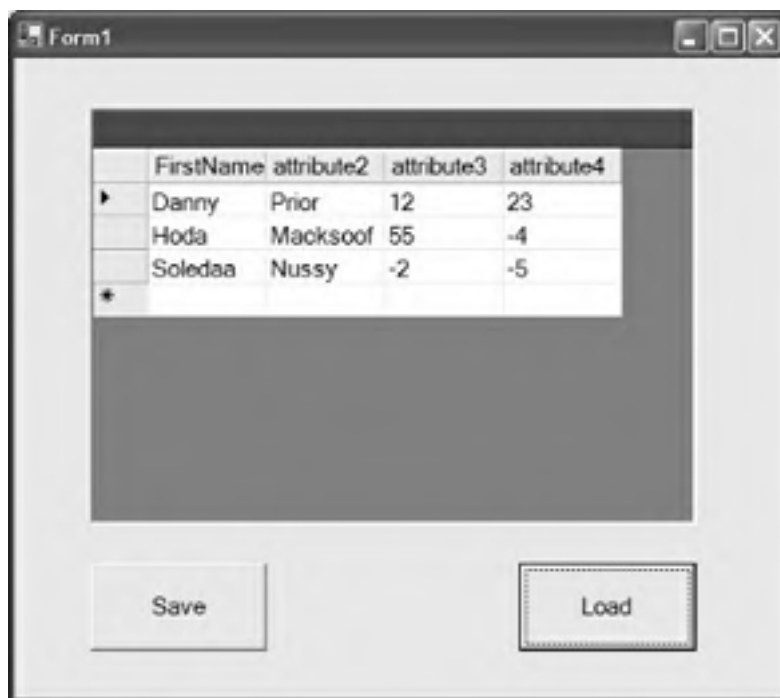


FIGURE 17.2 Saving and loading DataSets stored as XML is quite elegantly efficient in VB.NET.

[Team Fly](#)

 Previous

Next 

```
Dim fs As New FileStream('c:\test.txt", FileMode.Create,
Dim XMLf As New Soap.SoopFormatter
XMLf.Serialize(fs, Arr)
fs.Close()
'read it back
Dim XMLf1 As New Soap.SoopFormatter
Dim fs1 As New FileStream("c:\test.txt", FileMode.Open, Fi
ArrNew = XMLf1.Deserialize(fs1)
fs1.Close()
Console.WriteLine("The new array has been read back:")
Console.WriteLine(ArrNew(0))
Console.WriteLine(ArrNew(1))
Console.WriteLine(ArrNew(2))
```

End Sub

Technically, you don't need to Imports the binary version here of the serialization namespace, but what the heck. Adding a few extra, unused namespaces to a project doesn't do any harm. It doesn't make your executable any larger, and in my experience doesn't result in name collisions. Microsoft has taken care in its .NET assemblies to ensure unique member names as much as possible.

The resulting XML file contains even more than the usual verbosity characteristic of XML:

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
'xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns::SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:clr="http://schemas.microsoft.com/soap/encoding clr/1.0" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<SOAP-ENC:Array SOAP-ENC:arrayType="xsd:string[3]">
<item id="ref-2">This test</item>
<item id="ref-3">continues until</item>
<item id="ref-4">it finishes.</item>
```

```
</SOAP-ENC:Array>  
</SOAP-ENV:Body>  
</SOAP-ENV:Envelope>
```

You can use the `BinaryFormatter` technique in many cases when you are working within the .NET Framework—it's both faster and more efficient as store (it creates more compact files). But when you need to send data across the Internet, or communicate outside the .NET platform—SOAP-style is likely your formatting of choice.

***TIP** The `Serializable` attribute is not inheritable, so if you derive a new class from a serializable class, you must explicitly mark that new class as `<Serializable(>` as well, as illustrated by the following example.*

Mixing and Matching Types

Serialization permits you to easily mix and match various kinds of data in the same file. Once you've started feeding data into a stream, you can keep that stream open and pipe more data through it, even data of a different type or organization. In other words, you can fill your file or other data target with all kinds of data, and data collections, as long as you remember to read them back in the correct order. That is, if you store an `Int32`, a `Char` array, and then an object, you must read these same data types back in the same order that they were sent through the pipe: `Int32`, array, object.

If you thought the XML file in the previous example was a bit verbose, considering the simplicity and brevity of the structure and data, wait until you see the bloat-o-matic file created in the next example, Listing 17.9. This example shows how to save a structure followed by an `ArrayList`. (See the previous section if you have not added a `Soap` namespace to your project.)

LISTING 17.9: THIS CODE ILLUSTRATES XML VERBOSITY

```
Imports System.IO  
Imports System.Runtime.Serialization.Formatters  
  
Private Sub Form1_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles MyBase.Load  
  
    Dim m As New MeatPot  
  
    m.Name = "Mr. Porko"  
    m.Size = 44422  
    m.Price = 1412.99  
  
    Dim MyArray As New ArrayList  
  
    MyArray.Add("Sliders")  
    MyArray.Add("Grinders")  
    MyArray.Add("PoBoys")
```



```
Dim fs As New FileStream('c:\test.txt', FileMode.Create, Fil

Dim XMLf As New Soap.SoopFormatter

'hetero serialization:
XMLf.Serialize(fs, m)
XMLf.Serialize(fs, MyArray)
fs.Close()

End Sub
End Class

<Serializable(> Public Structure MeatPot
    Dim Name As String
    Dim Size As Integer
    Dim Price As Decimal
End Structure
```

Press F5 and use Windows Explorer to look at `Test.txt`. You'll see the big SOAP file full of XML explanations of the structures and data that were saved. I won't reproduce the entire thing, with its schemas upon schemas, but here's the meat of the file illustrating how your data was translated into the usual XML nested tag pair storage system:

```
<a1:MeatPot id="ref-1"
'<Name id="ref-3">Mr. Porco<Name>
<Size>44422</Size>
<Price>1412.99</Price>
'<a1:MeatPot>
<a1:ArrayList id="ref-1"
xmlns:a1="http://schemas.microsoft.com/clr/ns/System.Collections">
<_items href="#ref-2"/>
<_size>3<_size>
<_version>3<_version>
'<a1:ArrayList>
<SOAP-ENC:Array id="ref-2" SOAP-ENC:arrayType="xsd:anyType[16]">
<item id="ref-3" xsi:type="SOAP-ENC:string">Sliders<item>
<item id="ref-4" xsi:type="SOAP-ENC:string">Grinders<item>
<item id="ref-5" xsi:type="SOAP-ENC:string">PoBoys<item>
<SOAP-ENC:Array>
```


READING MIXED DATA

To load the XML from the file created in the previous example, it's your responsibility to deserialize the data types in the same order that you serialized them. In other words, you have to pay attention to the data types and their position in the stream.

The SoapFormatter's deserialize method returns only object variables. This means that you are also responsible for casting each of them into the correct data type. You can use the CType command to accomplish this casting. In Listing 17.10, the incoming data packages are cast into a Meat-Pot structure, followed by an ArrayList (this structure and ArrayList were saved to the hard drive in Listing 17.9).

LISTING 17.10: DESERIALIZING AND CASTING MIXED DATA

```
Private Sub Form1_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles MyBase.Load  
  
    Dim m1 As New MeatPot  
  
    Dim ar As New ArrayList  
  
    Dim fs As New FileStream(''c:\test.txt", FileMode.Open, FileAccess.Read) As FileStream  
    Dim XMLf As New Soap.SoapFormatter  
  
    m1 = CType(XMLf.Deserialize(fs), MeatPot)  
    ar = CType(XMLf.Deserialize(fs), ArrayList)  
  
    fs.Close()  
  
    Console.WriteLine(m1.Price)  
    Console.WriteLine(ar(2))  
  
End Sub
```

Here I used a single stream (fs) to deserialize both the MeatPot structure and the ArrayList. One stream can handle multiple serializations or deserializations.

.NET also includes a similar XML serializer, and it differs somewhat from the SOAP version. XML serialization permits you to specify the XML namespace, the name of tags, and whether a property is to be serialized as an attribute or element. However, object identity and assembly information are both lost, and it works only with public classes, fields, and properties. You can use this method when the consuming application doesn't need self-description (when it already knows the syntax governing the data).

This approach is useful when you want a quick way to communicate data, and to consume it under your control into your objects and databases. Listing 17.11 illustrates this technique.

[Team Fly](#)

 Previous

Next 

It's different now. You must use a `System.Random` object in .NET. The good news is that the `Random` object has useful capabilities that were not available via the old `Rnd` and `Randomize` functions. Try this example:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    Dim i As Integer
    For i = 1 To 100
        Console.Write(rand(i) & ' ' ")
    Next
End Sub
```

This function returns random numbers between 1 and 12:

```
Function rand(ByVal MySeed As Integer) As Integer
    Dim obj As New System.Random(MySeed)
    Return obj.Next(1, 12)
End Function
```

WARNING Although the arguments say 1, 12 in the line `Return obj.Next(1, 12)`, you will not get any 12s in your results. The numbers provided by the `System.Random` function in this case will range only from 1 to 11. I'm hoping that this error will be fixed at some point. However, even the latest version of VB.NET (2003) still gives you results only ranging from 1 to 11. Perhaps the same people who believe that arrays should start with a "zero element" also think that an upper bound of 12 should provide only numbers between 1 and 11. Can this truly odd behavior be an intentional feature of the `Random.Next` method? It's just too tedious to hear mystical techie justifications for these kinds of silly boundary rules, such as this 12, that turn out—surprise!—not to be the true boundaries after all.

On a happier note, Listing 2.14 is an example that illustrates how you can use the `NOW` command to seed your random generator.

LISTING 2.14: SEEDING THE RANDOM GENERATOR

```
Private Sub Button1_Click_1(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click

    Dim sro As New coin()
    Dim x As Integer
    Dim i As Integer

    For i = 1 To 100
        sro.toss()

        Dim n As String

        x = sro.coinvalue
        If x = 1 Then
            n = "tails"
        End If
    Next
End Sub
```

LISTING 17.11: AN EXAMPLE OF STREAMLINED WITH THE XMLSERIALIZER

```
Imports System.IO
Imports System.XML.Serialization

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

    Dim Xser As New XmlSerializer(GetType(YourClassName))

    Dim c As New YourClassName("Doris", "Duke")

    Dim fs As New FileStream("c:\test", FileMode.Create)
    Xser.Serialize(fs, c)

    fs.Close()

    'read it back

    Dim fs1 As New FileStream("c:\test", FileMode.Open)
    Dim c1 As YourClassName = CType(Xser.Deserialize(fs1), YourC
    fs1.Close()

    Console.WriteLine(c1.LastName)
End Sub

End Class

Public Class YourClassName

    Public FirstName As String
    Public LastName As String

    'the class must have a parameterless constructor
    Sub New()
    End Sub

    Sub New(ByVal FN As String, ByVal LN As String)
        Me.FirstName = FN
        Me.LastName = LN
    End Sub

End Class

End Class
```

The resulting file is largely free of XML's typical verbosity:

```
<?xml version='1.0'?>
<YourClassName xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <FirstName>Doris</FirstName>
  <LastName>Duke</LastName>
</YourClassName>
```

You can specify tag names, attributes, and so on when using the XMLSerializer. You add your specifications within the class itself, like this, fiddling with the properties of the class:

```
<XmlRootAttribute("FamousLady", Namespace:="http://www.yourplaceormine.com",
  IsNullable:=True)> Public Class YourClassName
  <XmlAttributeAttribute("HerFirstName")> Public FirstName As String
  <XmlIgnore()> Public LastName As String
```

These modifications result in a rather different XML file—one that has a new root name and a new namespace, and one that forces the FirstName property to be serialized as an attribute rather than an element and refuses to store the LastName property at all:

```
<?xml version="1.0"?>
<FamousLady xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  HerFirstName="Doris"
  xmlns="http://www.yourplaceormine.com" />
```

Deserialization Trapping

Perhaps you agree with critics who view "self-description" and "discovery" as merely honorable wishes rather than realized, practical programming behaviors.

Whatever else you may say, XML is not artificial intelligence. It doesn't permit you to write an application that loads an unknown XML file and "understands" what to do with the structures and data therein.

If you receive this from a chemical company:

```
<Borax>5</Borax>
```

your application cannot know what it means. Some human programmer must translate this into a usable piece of information. Does it mean five boxes of Boraxo soap? Five tons of raw borax? Borax is now \$5 a pound? Or 5 cents? To a computer, this "self-describing" data is merely another stream of bytes. And to an application that hasn't been programmed to recognize this particular stream of bytes, it's meaningless.

While deserializing an incoming XML stream, the XMLSerializer object throws exceptions when it cannot process something in the stream. The UnknownElement and UnknownAttribute events are

useful as a way of creating a log or otherwise flagging (for human intervention) that some "self-described" data isn't self-described enough.

If you don't write code to handle this situation, the serializer will simply ignore the incoming items it doesn't know what to do with. So you'd better use these events in your code. Let's use a previous example to create a problem in the XML. Using Notepad, modify the `c:\test` file so it looks like this (changes in boldface):

```
<?xml version="1.0"?>
<YourClassName xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Booboo>Doris</Booboo>
  <LastName>Duke</LastName>
</YourClassName>
```

Now consume this XML using this source code, which is filling an object that expects a First-Name element (property), not a Booboo:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MyBase.Load
  Dim Xser As New XmlSerializer(GetType(YourClassName))
  Dim fs1 As New FileStream("c:\testx", FileMode.Open)
  Dim c1 As YourClassName = CType(Xser.Deserialize(fs1), YourClassName)
  fs1.Close()
  Console.WriteLine(c1.FirstName)
  Console.WriteLine(c1.LastName)
End Sub
```

When you run this, the `FirstName` element was ignored by the `deserialize` method because it had that unrecognized tag `Booboo`. To deal with this problem—to alert a human that intervention is required—add an event to your code, like this:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MyBase.Load
  Dim Xser As New XmlSerializer(GetType(YourClassName))
  AddHandler Xser.UnknownElement, AddressOf deser_UnknownElement
  Dim fs1 As New FileStream("c:\testx", FileMode.Open)
  Dim c1 As YourClassName = CType(Xser.Deserialize(fs1), YourClassName)
  fs1.Close()
```

The interface of the application is based on regular Windows controls, and we've added quite a bit of code to make sure that the application can be used without the mouse. To activate the search button (the button with the question mark on the main form), users can press F3. We won't discuss the part of the code that handles the keystrokes, but you can open the project in Visual Studio and view the complete code, which is well documented.

The Application's Architecture

This is a connected application and it updates the database as soon as a row is edited, inserted, or deleted. When the user selects a row on the auxiliary form, the selected row is stored in the `Products DataTable` of the `DSProducts DataSet`. The `Products DataTable` is associated with the `DAProduct DataAdapter`, which is based on a `SELECT` statement that retrieves a product row by its ID:

```
SELECT      ProductID, ProductName, SupplierID, CategoryID,
            QuantityPerUnit, UnitPrice, UnitsInStock,
            UnitsOnOrder, ReorderLevel, Discontinued
FROM        Products
WHERE       (ProductID = @ID)
```

Once the user has selected a product on the auxiliary form, the product's ID is known and we can use it as an argument to the preceding `SELECT` statement to retrieve the values of the selected row. These values are then displayed on the controls of the main form. With the exception of the two `ComboBox` controls, no other control on the main form is bound.

The `DSProducts DataSet` holds two more tables, the `Categories` and `Suppliers` tables. The `DACategories` and `DASuppliers DataAdapters`, which fill the tables with the categories and suppliers, respectively, are based on the `GetAllCategories` and `GetAllSuppliers` stored procedures, whose listings are shown next:

```
CREATE PROCEDURE GetAllCategories
AS
SELECT CategoryID, CategoryName
FROM Categories
```

and

```
CREATE PROCEDURE GetAllSuppliers
AS
SELECT SupplierID, CompanyName
FROM Suppliers
```

To test the application, you must attach the two stored procedures to the database. The two `DataAdapters` populate the corresponding tables in the `DataSet` when the application's main form is loaded. Because the form is initially empty, the two `ComboBox` controls shouldn't display any category or supplier name, so we set the `SelectedIndex` property of the two controls to -1. Listing 18.1 shows the main form's Load event handler:

LISTING 18:1: POPULATING THE DSPRODUCTS DATASET AT STARTUP

```
Private Sub Form1_Load(ByVal sender As System.Object, _  
                        ByVal e As System.EventArgs) _  
                        Handles MyBase.Load  
    Products1.Clear()  
    DACategories.Fill(Products1, "Categories")  
    DAsuppliers.Fill(Products1, "Suppliers")  
    cmbSupplier.SelectedIndex = -1  
    cmbCategory.SelectedIndex = -1  
End Sub
```

The reason we load the categories and suppliers when the program starts is that these two tables will be used as lookup tools in selecting the current product's category and supplier, so we need to display their names on the two ComboBox controls. To bind the two controls to the corresponding DataTables, we've set a few properties, as shown next. The following table shows the settings of the data-binding properties of the `cmbCategory` ComboBox control:

Property	Setting
DataSource	Products1.Categories
DisplayName	CategoryName
ValueMember	CategoryID

The equivalent settings for the `cmbSupplier` ComboBox control are shown next:

Property	Setting
DataSource	Products1.Suppliers
DisplayName	CompanyName
ValueMember	SupplierID

The DataSource property indicates where the data will be read from—in our case, from the Categories DataTable of the Products DataSet. The DisplayName property is the column we want to display on the control, and the ValueMember property is the column that identifies each row. The DisplayName property is set to a column that makes sense to the user, such as a name, and need not be unique. The ValueMember property is set to the key column of the table.

As soon as we fill the Categories and Products DataTables, the corresponding ComboBox controls are also populated—no need to write any code to populate a data bound control, just set a few properties. The currently selected item's ValueMember property is the ID of the product's category or supplier. In other words, users can select a category or supplier by name on the control. Our code will read the control's ValueMember, which returns the ID of the selected item.

The project's DataSet contains two relations, one between the Products and Suppliers DataTables and a second one between the Products and Categories DataTables. These relations are enforced, and the user can't specify an invalid category or supplier for a product. Allowing the user to select a category and a


```
        Else
            n = "'heads"
        End If

        n = n & " "

        debug.Write(n)

    Next i

End Sub

End Class

Class coin

    Private m_coinValue As Integer = 0

    Private Shared s_rndGenerator As New System.Random(Now.Millisecond)

    Public ReadOnly Property coinValue() As Integer
        Get
            Return m_coinValue
        End Get
    End Property

    Public Sub toss()
        m_coinValue = s_rndGenerator.Next(1, 3)
    End Sub

End Class
```

As nearly always in VB.NET, there is more than one way to do things. The next example uses the `system.random` object's `Next` and `NextDouble` methods. The seed is automatically taken from the computer's date/time (no need to supply a seed as was done in the previous example, or to use the old VB6 `Randomize` function). The `Next` method returns a 32-bit integer; the `NextDouble` method returns a double-precision floating point number ranging from 0 up to (but not including) 1.

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    Dim r As New System.Random()
    Dim x As Integer
    Dim i As Integer
```

supplier by name is much friendlier than requesting a numeric ID, as you would have to do with the DataGrid control. Moreover, we won't have to validate the CategoryID and SupplierID columns (users can't assign invalid data to these two columns). A drawback of our interface is that it doesn't allow the insertion of new categories or suppliers, but this can be easily fixed by adding a button next to each ComboBox control, which will open a similar auxiliary form for the Categories or the Suppliers table.

The selection of the desired product takes place on an auxiliary form and the selection can be based on several criteria, but only one at a time. Depending on the user's selection, the program downloads a few columns of the Products table and displays them on the ListView control at the bottom of the form. The selected columns are the product's ID, name, and price. The code behind the search form doesn't download all the columns you see on the main form, because it doesn't have to. It downloads enough information to help the user make a selection. Once the user has selected a single product, we download all the columns of a specific row for editing purposes. Northwind is a small database and you could download all the columns of the selected rows from the Products table. In a large database, however, the table with the products contains many columns, and many of them are used internally by the application. You should never download to the client more data than necessary.

The auxiliary form contains the `AuxTables` DataSet, which stores the Categories and Suppliers tables, which are populated by the `DACategories` and `DASuppliers` DataAdapters, respectively. The two DataAdapters select all the rows of the corresponding table and populate their DataTables in the DataSet.

To retrieve the products that match the user-supplied criteria, the code creates a SELECT statement and executes it against the database through the `DASelectedProducts` DataAdapter. This DataAdapter executes its `SelectCommand` and populates the Products table in the `AuxTables` DataSet with the selected rows.

The Application's Code

When the user clicks the button with the question mark, the following code displays the auxiliary form for selecting a product:

```
Private Sub btnnSearch_MouseUp(ByVal sender As Object, _  
                               ByVal e As System.Windows.Forms.MouseEventArgs) _  
                               Handles btnnSearch.MouseUp  
    ShowExtForm()  
End Sub
```

The `ShowExtForm()` subroutine creates a new instance of `Form2` and displays it modally. The auxiliary form exposes the ID of the selected product as property, the `ProdID` property, which the main form reads and then uses to display the selected product. The code of the `ShowExtForm()` sub-routine is shown in Listing 18.2.

LISTING 18.2: SHOWING THE SEARCH FORM

```
Private Sub ShowExtForm()
```

```
Dim searchFRM As New Form2  
If searchFRM.ShowDialog() = DialogResult.OK Then  
    Dim prodID As Integer
```

[Team Ely](#)

 Previous

Next 

```
        prodID = searchFRM.ProdID
        If prodID > -1 Then
            ShowProduct(prodID)
        End If
    End If
End Sub
```

The ShowProduct() subroutine retrieves the selected row and then calls the PopulateControls() subroutine to display the values of this row. Downloading all categories and suppliers to the client is a convenient technique, but it may lead to following condition. A user may add a new category and then assign this category to a product (an existing or new product). When another user selects this product, an exception will be thrown, because the DataSet won't be able to enforce the relation between the product and its category (the newly inserted category will not be at the client). Our code must detect this condition and reload first the rows of the Categories table and then the selected row of the Products table. In our code, when we detect that a related row in either of the Categories or Suppliers table is missing, we reload both tables. The ShowProduct() subroutine is shown in Listing 18.3.

LISTING 18.3: RETRIEVING AND DISPLAYING A PRODUCT BY ITS ID

```
Private Sub ShowProduct(ByVal productID As Integer)
    ClearFields()
    DAProduct.SelectCommand.Parameters('@ID').Value = productID
    Products1.Products.Clear()
    Try
        DAProduct.Fill(Products1, "Products")
    Catch exc As ConstraintException
        DACategories.Fill(Products1, "Categories")
        DAsuppliers.Fill(Products1, "Suppliers")
        Products1.Products.Clear()
        DAProduct.Fill(Products1, "Products")
    End Try
    PopulateControls()
    txtProductName.Focus()
End Sub
```

The PopulateControls() subroutine displays the selected row's values on the form's controls and its code is straightforward. In addition, it locks all the controls on the main form. You can look up the code of the PopulateControls() subroutine in the project's code.

When the Add or Edit buttons are clicked, the program unlocks the fields on the form by calling the UnlockFields() subroutine. When a new row is added, in addition to unlocking the fields on the form the code also clears them, and it also clears the Products DataSet. The Edit and Add buttons prepare the application for the editing of the current row and the insertion of a new row, respectively; their code is shown in Listing 18.4.

LISTING 18.4: THE EDIT AND ADD BUTTON'S CODE

```
Private Sub btnEdit_Click(ByVal sender As System.Object, _
                        ByVal e As System.EventArgs) _
                        Handles btnEdit.Click
    If Products1.Products.Rows.Count = 0 Then
        MsgBox("Please select a product to edit!")
        Exit Sub
    End If
    UnlockFields()
    HideButtons()
End Sub

Private Sub btnAdd_Click(ByVal sender As System.Object, _
                        ByVal e As System.EventArgs) _
                        Handles btnAdd.Click
    Products1.Products.Clear()
    ClearFields()
    UnlockFields()
    HideButtons()
End Sub
```

The two event handlers are similar and quite trivial. The HideButtons() subroutine hides the usual editing buttons and displays the OK and Cancel buttons in their place. The OK button must handle an edit operation as well as the insertion of a row. If the Products Data Table contains no rows, it means that the user is adding a new row and the code must insert a new row into the Data-Table. Otherwise, the OK button was clicked to end an edit operation, in which case we simply update the values of the existing row in the Products DataTable. The OK button's Click event handler is shown in Listing 18.5:

LISTING 18.5: ENDING AN EDIT OR INSERT OPERATION

```
Private Sub btnOK_Click(ByVal sender As System.Object, _
                        ByVal e As System.EventArgs) _
                        Handles btnOK.Click
    If txtProductName.Text.Trim = " " Then
        MsgBox("Product Name must have a value!")
        Exit Sub
    End If
    If txtPrice.Text.Trim = "" Then
        MsgBox("The product must have a price!")
        Exit Sub
    End If
    If Not IsNumeric(txtPrice.Text.Trim) Then
        MsgBox("Invalid price!")
        Exit Sub
    End If
End Sub
```

```
End If
If CDec(txtPrice.Text) < 0 Then
    MsgBox(''The price can't be negative!')
    Exit Sub
End If
Dim newRow As Products.ProductsRow
If Products1.Products.Rows.Count > 0 Then
    newRow = Products1.Products.Rows(0)
Else
    'if not, it s a new row
    newRow = Products1.Products.NewRow
End If
' Get the new values from the controls on the form
' and place them into the columns of the new/edited row
ReadFieldValues(newRow)
If Products1.Products.Rows.Count = 0 Then
    Products1.Products.Rows.Add(newRow)
End If
' Attempt to submit changes to database
Try
    DAProduct.Update(Products1, "Products")
Catch exc As Exception
    ' update operation failed
    MsgBox(exc.Message)
    ' If this is an edited row, reject changes
    If Products1.Products.Rows(0).RowState = _
        DataRowState.Modified Then
        Products1.Products.RejectChanges()
    Else
        ' and if it's a new row, remove it
        Products1.Products.Rows.RemoveAt(0)
        ClearFields()
    End If
    Exit Sub
End Try
' update operation succeeded, continue with viewing/editing
LockFields()
ShowButtons()
bbtnSearch.Focus()
End Sub
```

The code starts by validating the data. If the supplied data doesn't violate any of the database's constraints, it calls the `ReadFieldValues()` to copy the values from the controls into a typed `DataRow` object. For new products, the code creates a new `DataRow` object with the following statements:

```
Dim newRow As Products.ProductsRow
newRow = Products1.Products.NewRow
```


When the user edits an existing row, this is the only row in the Products table of the DataSet and we simply retrieve it from its table with the following statement:

```
newRow = Products1.Products.Rows(0)
```

The ReadFieldValues() subroutine copies the values from the form's controls into the newRow object's columns. Then the code adds this row to the Products table and calls the DAProducts DataAdapter's Update method to submit the new (or edited) row to the database. If the Update method fails, the structured exception handler catches it and handles it accordingly. If the row is a new one, it's removed from the DataSet. If it's an existing row that was edited, the exception handler rejects the changes and restores the DataRow object to its state before the editing process. Restoring a single row to its original version and asking the user to edit it again is quite reasonable. This approach, however, can't be used with a disconnected application, because dozens of rows may fail to update the underlying tables.

The Cancel button on the form calls the PopulateControls() subroutine to re-display the selected product. Notice that we don't have to undo any changes, because the DataRow object that represents the selected product hasn't been modified yet. Only the values of the controls on the form have been changed, and these changes are lost when the controls are repopulated.

THE SEARCH FORM'S CODE

The auxiliary form contains a DataAdapter, the DSelectedProducts DataAdapter, which we configure from within our code with the appropriate SELECT statement, depending on the criteria specified by the user. Then we use the DataAdapter to fill the AuxTables DataSet, which contains the Products DataTable (this is where we store the selected products), as well as two more tables, the Categories and Suppliers tables. The two tables are populated through the DSuppliers and DCategories DataAdapters, which select the IDs and names of all suppliers and all categories respectively. The Categories and Suppliers DataTables are populated when the form is loaded and they're bound to the cmbSupplier and cmbCategory ComboBox controls of the auxiliary form.

When a new category is selected on the cmbCategories ComboBox, the program executes a SELECT statement that retrieves all the products in the selected category, populates the Selected-Products DataTable of the AuxTables Dataset, and calls the ShowProducts() subroutine to display the selected rows on a ListView control. If the user selects a supplier, the code we execute is identical, with the exception of the SELECT statement, which now retrieves the selected supplier's products. When the user presses the Enter key while the top TextBox control has the focus, the code executes a third SELECT statement, which selects products by name. Listing 18.6 shows the code behind the SelectedIndexChanged event handler of the cmbCategory ComboBox control.

LISTING 18.6: RETRIEVING AND DISPLAYING THE PRODUCTS OF A CATEGORY

```
Private Sub cmbCategory_SelectedIndexChanged( _  
    ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) _  
    Handles cmbCategory.SelectedIndexChanged
```

```
txtProductName.Text = ""  
cmbSupplier.Text = ""
```

[Team Ely](#)

 Previous

Next 

```
AuxTables1.Products.Clear()  
' Use a parameterized query to select products by category  
DASelectedProducts.SelectCommand.CommandText = _  
    'SELECT ProductID, ProductName, UnitPrice " & _  
    "FROM Products WHERE CategoryID=@category"  
DASelectedProducts.SelectCommand.Parameters.Clear()  
DASelectedProducts.SelectCommand.Parameters.Add( _  
    "@category" , SqlDbType.Int)  
DASelectedProducts.SelectCommand._  
    Parameters("@category").Value = cmbCategory.SelectedValue  
DASelectedProducts.Fill(AuxTables1, "Products")  
ShowProducts()  
End Sub
```

The ShowProducts() subroutine is straightforward and we won't show its listing here. You can open the project with Visual Studio and examine the code. When the OK button on the auxiliary form is clicked, the program stores the ID of the selected product to the ProdID variable, which is a public field of the form. The code on the main form can access this field and retrieve its value when the user closes the auxiliary form to return to the main form of the application. The code behind the OK button of the auxiliary form is shown in Listing 18.7.

LISTING 18.7: SELECTING A PRODUCT ON THE SEARCH FORM

```
Private Sub btnOK_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) _  
    Handles btnOK.Click  
    If lstProducts.SelectedIndices.Count > 0 Then  
        ProdID = lstProducts.SelectedItem.Tag  
        Me.DialogResult = DialogResult.OK  
    Else  
        Me.DialogResult = DialogResult.Cancel  
        Exit Sub  
    End If  
End Sub
```

The same code is executed from within the DoubleClick event handler of the ListBox control at the bottom of the auxiliary form.

Run the NWProducts application to check out its operation and try to operate the application's interface with the keyboard only (the shortcut key for the search button on the main form is F3). You can run two instances of the same application on your desktop and see how the NWProducts application handles concurrency, as well as update errors. Start an instance of the application and then add a new category to the database from within Enterprise Manager or the Query Analyzer. The new category won't appear in the list of categories on the running application's form. Then start a new instance of the application and add a new product that belongs to this category. Switch to the



FIGURE 18.3 The NWOrders application

Users can enter items by supplying their ID in the first TextBox control and move to the next box by pressing the Enter key (the Tab key will also work). The boxes with the selected product's name and price are skipped and the user is taken to the Quantity box, where the suggested quantity is one. Pressing Enter one more time takes the user to the Discount box, and with a third Enter the new detail line is added to the grid. The focus is also moved to the ID box, where users can enter the ID of the next item in the order. This is also how most barcode readers work, so you'll be able to deploy an application like this at a point of sales.

If the product's ID isn't known, users can search for a product by name. They can enter part of the product's name in the second TextBox control near the top of the form and press the Enter key. The search will most likely return more than a single item. The matching product names will be displayed on a ListBox control, as shown in Figure 18.4, and the user can move to the desired one with the arrow keys and select it by pressing Enter (or double-click the name of the desired product in the ListBox control). The ListBox control with the product names will disappear as soon as an item is selected.

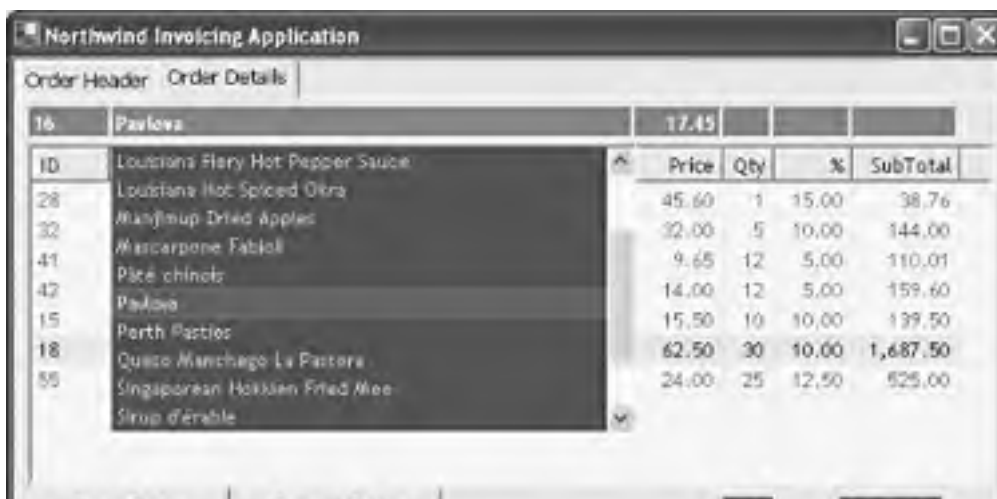




FIGURE 18.4 Selecting a product by name

As the user changes the current selection on the ListBox control with the product name, the current product's ID and price appear on the TextBox controls at the top of the form. You can edit the application's interface to display even more properties on the form.

When an order is completed, it's submitted to the database. Each order contains two sections: its header and its details. The order's header contains the ID of the customer who placed the order and the ID of the employee who made the sale. The NWOrders application allows you to specify the customer and employee by selecting them from the appropriate list on the Order Header tab, as shown in Figure 18.5. After specifying the order's header, users can switch to the Order Details tab by pressing Alt+D. On the Order Details tab, users can specify the order's detail lines and commit or cancel the order with one of the two buttons at the bottom of the form.



FIGURE 18.5 Preparing the order's header

The Application's Architecture

When the form is loaded, the rows of the Customers and Employees tables are downloaded to the client and stored to the `DSCustomers` DataSet. The DataSet's DataTables are populated with the `DACustomers` and `DAEmployees` DataAdapters. The `DACustomers` DataAdapter populates the `DSCustomers` DataSet with the following SELECT statement:

```
SELECT      CustomerID, CompanyName,
            ContactName, ContactTitle,
            Address, City, Region, PostalCode, Country
FROM        Customers
ORDER BY   CompanyName
```

The `DACustomers` DataAdapter need not update the Customers table; clear the Generate Insert, Delete, and Update Statements options on the Advanced tab of the DataSet generation wizard. The `DAEmployees` DataAdapter has the same configuration, except for the SELECT statement, which is shown next:

```
SELECT      EmployeeID, LastName, FirstName
FROM        Employees
```

ORDER BY LastName, FirstName

The order's detail lines are not stored to a DataSet. All the information about the detail lines is stored on the ListView control. The NWOrders application consists of three tiers. The presentation tier retrieves all the fields of the order from the controls on the form and uses them to create a custom object and pass it to the middle tier. The middle tier commits the order to the database in the

[Team Fly](#)

 Previous

Next 

context of a transaction. The third tier is the data tier, which consists of the database and the stored procedures we've written for this application.

The program uses two custom classes, the `ProductPrice` and `OrderedProduct` classes. The `ProductPrice` class is where we'll store product information: the product's name, ID, and price. We use this class to pass information from the middle tier to the presentation tier every time the program requests the price of a product. The `OrderedProduct` class is where we'll store the order's detail lines. When we request a product by its ID, the middle tier will return to our application an object of the `ProductPrice` type with information about the specified product. This—the product's name and price—is the information we need to sell the product. When the order is ready, we'll pass to the middle tier an `ArrayList` of `OrderedProduct` objects, as well as the order's header, and the middle tier's code will submit the order to the database. The `OrderedProduct` class stores the ordered product's ID, its price, the ordered quantity, and the discount.

Let's consider the operations we need to perform from the presentation tier. First, we need to be able to retrieve a product by its ID and display a few of its columns on the form. This operation will be implemented by the `GetProductByID` method, which accepts a product ID as argument and returns a `ProductPrice` object with the product's name and price. We also need to search products by name. This operation will be implemented with the `GetProductsByName` method, which accepts as argument a string and returns an array of `ProductPrice` objects. They're the products whose names match the string passed as argument.

To submit the order to the database, we create an instance of the `OrderedProduct` class, populate it with the order's header and its details, and pass it to the `AddOrder` method, which commits it to the database. If the order is inserted successfully, the `AddOrder` method returns the ID of the new order (a number assigned to the order by the database). If not, it returns the value `-1`.

THE ORDER CLASS CLASS

The `OrderClass` class is the middle tier component of the `NWOrders` application. It contains two nested classes and a few methods. The two classes are the `ProductPrice` class, which represents a product for the purposes of appending it to the order, and the `OrderedProduct` class, which represents a detail line for the purposes of submitting it to the database. The two classes are similar, but the `ProductPrice` class doesn't contain a quantity and the `OrderedProduct` class doesn't contain the product name. When we sell, we assume that the customer has selected the products and all products have a label with a bar code (or at least a label with a unique ID and the price). The cashier doesn't need any more information beyond the product's name and price, so the `ProductPrice` class exposes three members: the product's ID, its name, and its price. It also exposes a custom `ToString` method, which returns the product's name. You'll see shortly how this member is used.

The OrderedProduct class represents a detail line of the invoice and contains the fields that will be included in the order's details table: the ID of a product, its price, a quantity, and a discount. The OrderedProduct class exposes the following members: ProductID, ProductPrice, ProductQTY, and ProductDiscount. Both classes belong to the OrderClass class and their declaration is shown next:

```
Public Class OrderClass
    Public Class ProductPrice
        Public ProductName As String
        Public ProductID As String
```

```
For i = 1 To 10
    Debug.Write(r.Next & ' ' ,")
Next
For i = 1 To 10
    Debug.Write(r.NextDouble & " , ")
Next
End Sub
```

The following is a sample of what you see in the output window when you run this example:

```
519421314, 2100190320, 2103377645, 526310073, 1382420323, 408464378, 985605837,
265367659, 665654900, 1873826712
0.263233027543515, 0.344213118471304, 0.0800133510865333, 0.902158257040269, _
0.735719954937566, 0.283918539194352, 0.946819610403301, 0.27740475408612, _
0.970956700374818, 0.803866669909966
```

Any program that uses this technique can be guaranteed to get unique results each time it's executed. It's impossible to execute that program twice at the *same* time and date, just as you cannot kiss yourself on the face. (Even Mick Jagger can't.)

Of course, if you choose, you can still employ the older VB6 version of the randomizing function:

```
Dim r As Double
    r = Rnd
    MsgBox(r)
```

It's not necessary in this case to invoke the `Microsoft.VisualBasic` namespace. Why not? Pure whimsy. Sometimes they decide to put things into a special package called a *wrapper*. The `Rnd` function is in a wrapper, as another attempt to provide backward compatibility between VB.NET and earlier VB code (such as using `MsgBox` rather than having to type in the whole tedious `MessageBox.Show` every time you want to test a variable during debugging).

If you've used the `Rnd` function before, you might recall that it will provide *identical* lists of random numbers by default (which can be quite useful when attempting to, for example, debug a game). If you want to get identical lists in VB.NET, you can seed the `Random` object, like this:

```
Dim r As New System.Random(14)
```

Filling an Array

The `Random.NextBytes` method automatically fills an array of bytes with random numbers between 0 and 255. Here's how:

```
Dim r As New System.Random()
Dim i
Dim a(52) As Byte 'create the byte array
r.NextBytes(a) ' fill the array
```



```
Public ProductPrice As Decimal
Public Overrides Function ToString() As String
    Return ProductName
End Function
End Class

Public Class OrderedProduct
    Public ProductID As String
    Public ProductPrice As Decimal
    Public ProductQTY As Integer
    Public ProductDiscount As Decimal
End Class
```

In addition to the two classes that represent the entities we need in order to specify an order, the OrderClass module also contains the following methods:

GetProductByID This method accepts as argument a product ID and returns a ProductPrice object that represents the selected product.

GetProductsByName This method accepts as argument a string and returns an array of Product-Price objects, which represent the products whose names contain the specified string.

AddOrder This method accepts a number of arguments that represent an order and commits it to the database. These arguments are the IDs of a customer and an employee (the order's header), as well as an ArrayList of OrderedProduct objects (the order's details).

The implementation of the three methods of the OrderClass class is shown in Listing 18.8.

LISTING 18.8: THE IMPLEMENTATION OF THE ORDERCLASS

```
Public Class OrderClass
    Public Class ProductPrice
        Public ProductName As String
        Public ProductID As String
        Public ProductPrice As Decimal
        Public Overrides Function ToString() As String
            Return ProductName
        End Function
    End Class

    Public Class OrderedProduct
        Public ProductID As String
        Public ProductPrice As Decimal
        Public ProductQTY As Integer
        Public ProductDiscount As Decimal
    End Class

    Public Shared Function GetProductsByName(
        ByVal ProductName As String) As ProductPrice()
```

```
Dim CMD As New SqlClient.SqlCommand
CMD.CommandText = 'GetProductsByName'
CMD.CommandType = CommandType.StoredProcedure
CMD.Parameters.Add(New SqlClient.SqlParameter( _
    "@productName", ProductName))
Dim CN As New SqlClient.SqlConnection
CN.ConnectionString = _
    "data source = powertoolkit;" & _
    "initial catalog=Northwind;" & _
    "integrated security=SSPI;" & _
    "persist security info=False;"
CN.Open()
CMD.Connection = CN
Dim DR As SqlClient.SqlDataReader = CMD.ExecuteReader()
Dim P(999) As ProductPrice
Dim prod As New ProductPrice
Dim i As Integer
While DR.Read
    prod = New ProductPrice
    prod.ProductID = DR.Item("ProductID")
    prod.ProductName = DR.Item("ProductName")
    prod.ProductPrice = DR.Item("UnitPrice")
    P(i) = prod
    i=i + 1
End While
ReDim Preserve P(i - 1)
If i > 0 Then
Return P
Else
    Return Nothing
End If
End Function

Public Shared Function GetProductByID( _
    ByVal ProductID As String) As ProductPrice
Dim CMD As New SqlClient.SqlCommand
CMD.CommandText=GetProductByID
CMD.CommandType=CommandType.StoredProcedure
CMD.Parameters.Add(New SqlClient.SqlParameter( _
    "@productID", CInt(ProductID)))
Dim CN As New SqlClient.SqlConnection
CN.ConnectionString =_
    "data source=powertoolkit;" & _
    "initial catalog=Northwind;" & _
    "integrated security=SSPI;" & _
    "persist security info=False;"
CMD.Connection=CN
CN.Open()
```

```
Try
    Dim DR As SqlClient.SqlDataReader=CMD.ExecuteReader()
    Dim P As ProductPrice
    If DR.Read Then
        P=New ProductPrice
        P.ProductID=DR.Item(''ProductID")
        P.ProductName=DR.Item("ProductName")
        P.ProductPrice=DR.Item("UnitPrice")
    End If
    Return P
Catch ex As Exception
    Return Nothing
End Try
End Function

Public Shared Function AddOrder(ByVal customerID As String, _
                                ByVal empId As Integer, _
                                ByVal Order As ArrayList) As Boolean
    Dim CMD As New SqlClient.SqlCommand
    CMD.CommandText="AddHeader"
    CMD.CommandType=CommandType.StoredProcedure
    CMD.Parameters.Add(New SqlClient.SqlParameter( _
        "@customerID", customerID))
    CMD.Parameters.Add(New SqlClient.SqlParameter( _
        "@employeeID", empId))
    Dim CN As New SqlClient.SqlConnection
    CN.ConnectionString=_
        "data source=powertoolkit;" & _
        "initial catalog=Northwind;" & _
        "integrated security=SSPI; & _
        "persist security info=False;"
    CMD.Connection=CN
    Dim TRN As SqlClient.SqlTransaction
    CN.Open()
    TRN=CN.BeginTransaction
    CMD.Transaction=TRN
    Try

        Dim OrderID As Integer
        OrderID=CMD.ExecuteScalar()
        CMD=New SqlClient.SqlCommand
        CMD.Connection=CN
        CMD.Transaction=TRN
        CMD.CommandText="AddDetailLine"
        CMD.CommandType=CommandType.StoredProcedure
        CMD.Parameters.Add(New SqlClient.SqlParameter( _
            "@OrderID" , Data.SqlDbType.Int))
        CMD.Parameters.Add(New SqlClient.SqlParameter( _
```

```
        '@ProductID', Data.SqlDbType.Int))
CMD.Parameters.Add(New SqlClient.SqlParameter( _
        "@Quantity", Data.SqlDbType.Int))
CMD.Parameters.Add(New SqlClient.SqlParameter( _
        "@Price", Data.SqlDbType.Money))
CMD.Parameters.Add(New SqlClient.SqlParameter( _
        "@Discount", Data.SqlDbType.Real))
Dim Item As OrderedProduct
For Each Item In Order
    CMD.Parameters("@OrderID").Value=OrderID
    CMD.Parameters("@ProductID").Value=Item.ProductID
    CMD.Parameters("@Quantity").Value=Item.ProductQTY
    CMD.Parameters("@Price").Value=Item.ProductPrice
    CMD.Parameters("@Discount").Value=Item.ProductDiscour
    CMD.ExecuteNonQuery()
Next
Catch exc As Exception
    TRN.Rollback()
    CN.Close()
    Return False
End Try
TRN.Commit()
CN.Close()
Return True
End Function
End Class
```

To retrieve products either by ID or by name, the methods of the OrderClass component execute a stored procedure against the database and return the matching products. The GetProductByID method returns a single ProductPrice object, while the GetProductsByName method returns an array of ProductPrice objects. The result of the query is read through a DataReader and our middle tier's code is responsible for extracting the values from the DataReader and populating the instance(s) of the ProductPrice objects (one for each selected product row). The two methods use the GetProduct-ByID and GetProductsByName stored procedures, whose implementations are shown in Listings 18.9 and 18.10. The two stored procedures must also be appended to the database.

LISTING 18.9: THE GETPRODUCTBYID STORED PROCEDURE

```
CREATE PROCEDURE GetProductByID
@productID int
AS
SELECT    ProductID, ProductName, UnitPrice
FROM      Products
WHERE     ProductID=@productID
```

LISTING 18.10: THE GETPRODUCTSBYNAME STORED PROCEDURE

```
CREATE PROCEDURE GetProductsByName
@productName varchar(100)
AS
SELECT      ProductID, ProductName, UnitPrice
FROM        Products
WHERE       ProductName LIKE '%' + @productName + '%'
```

The AddOrder method executes the AddHeader and AddDetailLine stored procedures. The first stored procedure is executed once for each order and inserts a single row into the Orders table. The second stored procedure is executed once for each detail line. The two stored procedures, whose listings are shown in Listing 18.11 and 18.12, must be appended to the database. They're two simple stored procedures that insert the values passed as arguments into the corresponding table. The AddOrder method of the middle tier component initiates a transaction and then executes the AddHeader stored procedure. Then it calls the AddDetailLine stored procedure as many times as necessary and finally commits the transaction, if everything went well. If an error occurs in the process, the order is rejected as a whole. To test the AddOrder method, create a new order that contains one or more rows with negative prices. The client application doesn't validate the price, so you can enter any value. You can validate the price before submitting the order to the database from within your code, or even lock the price field. If the application is going to be used at a POS, you must certainly lock this field, because we don't want operators to edit the prices.

LISTING 18.11: THE ADDHEADER STORED PROCEDURE

```
CREATE PROCEDURE AddHeader
@CustomerID  varchar(5),
@EmployeeID  int
AS
DECLARE      @orderID int
INSERT      Orders (CustomerID, EmployeeID, OrderDate)
VALUES      (@customerID, @employeeID, getDate())
SELECT      SCOPE_IDENTITY()
```

LISTING 18.12: THE ADDDETAILLINE STORED PROCEDURE

```
CREATE PROCEDURE AddDetailLine
@OrderID      int,
@ProductID    int,
@Quantity     smallint,
@price        money,
@discount     real
AS
```



```
INSERT [Order Details]
      (OrderID, ProductID, Quantity, UnitPrice, Discount)
VALUES
      (@orderID, @productID, @quantity, @price, @discount)
```

The Application's Code

The selection of a new product takes place in the KeyUp event handler of the txtID control (the TextBox where the product's ID is shown). This event handler examines the key that was pressed. If the user pressed the Enter key, it means that the user has entered a product ID and the code should retrieve the corresponding row from the database and return it to the application. This action takes place in the middle tier, with the GetProductByID method, which returns an instance of the ProductPrice class. If no matching row is found, the method returns a Nothing value, in which case the application turns the TextBox controls at the top of the form to red for a short moment (see the code of the AlarmUser() subroutine, which we will not discuss here; you may find it excessive, but it's an interesting technique to attract the user's attention in a non-invasive manner).

The fields of the selected product, which are properties of the ProductPrice object returned by the GetProductByID method, are displayed on the various TextBox controls at the top of the form. The program suggests a quantity of 1 and waits for the user to confirm the suggested quantity, or enter a different quantity, and press Enter.

If the user presses one of the up or down arrow keys, the focus is moved to the ListView control with the invoice's detail lines. The selected item is the one selected most recently on the control (the last time the user made a selection from this list). As the user moves through the order's detail lines in the grid, the values of the currently selected line are displayed on the TextBox controls at the top of the form. To signal his intention to edit the current row, the user must press Enter. The KeyUp event handler of the txtID control is shown in Listing 18.13.

LISTING 18.13: SELECTING A PRODUCT BY ITS ID

```
Private Sub txtID_KeyUp(ByVal sender As Object, _
                        ByVal e As System.Windows.Forms.KeyEventArgs)
    Handles txtID.KeyUp
    If e.KeyCode = Keys.Enter Then
        If txtID.Text.Trim = "" Then
            e.Handled = True
            Exit Sub
        End If
        Dim Product As OrderClass.ProductPrice
        Product = Invoice.GetProductByID(txtID.Text.Trim)
        If Product Is Nothing Then
            AlarmUser()
            Beep()
            txtID.SelectAll()
        Else
```

```
        txtDescription.Text = Product.ProductName
        txtPrice.Text = Product.ProductPrice
        txtQty.Text = '1'
        txtDiscount.Text = "0"
        txtQty.Focus()
    End If
End If

If e.KeyCode = Keys.Down Or e.KeyCode = Keys.Up Then
    If lvOrderGrid.Items.Count = 0 Then Exit Sub
    If lvOrderGrid.SelectedItems.Count = 0 Then
        lvOrderGrid.Items _
            (lvOrderGrid.Items.Count - 1).Selected = True
        lvOrderGrid.Items _
            (lvOrderGrid.Items.Count - 1).Focused = True
    End If
    lvOrderGrid.EnsureVisible(lvOrderGrid.SelectedIndex)
    lvOrderGrid.Focus()
End If
End Sub
```

The user may also enter some text in the txtProductName box and then press Enter, to select a product by name (because the ID of a product is not known at the time, or because the barcode reader can't read the product's label). The program displays all matching product names in a ListBox control below the txtDescription box, as shown in Figure 18.4, and moves the focus to the ListBox control. The user can then select the desired product with the arrow keys and press Enter again to display its fields on the controls at the top of the form, where they can be edited. The KeyUp event handler of the txtDescription control is shown in Listing 18.14.

LISTING 18.14: SELECTING A PRODUCT BY NAME

```
Private Sub txtDescription_KeyUp(ByVal sender As Object, _
    ByVal e As System.Windows.Forms.KeyEventArgs) _
    Handles txtDescription.KeyUp
    If e.KeyCode = Keys.Enter Then
        Dim Products() As OrderClass.ProductPrice
        Products = Invoice.GetProductsByName(txtDescription.Text.Trim)
        If Products Is Nothing Then
            Beep()
        Else
            lstProducts.Items.Clear()
            Dim i As Integer
            For i = 0 To Products.GetUpperBound(0)
                lstProducts.Items.Add(Products(i))
            Next
            If lstProducts.Items.Count = 1 Then
```

```
        lstProducts.SelectedIndex = 0
        lstProducts_SelectedIndexChanged(Me, Nothing)
        lstProducts_KeyUp(Me, e)
        Exit Sub
    End If
    If lstProducts.Items.Count < 10 Then
        lstProducts.Height = (lstProducts.ItemHeight + 1) * _
            lstProducts.Items.Count + 3
    Else
        lstProducts.Height = 11 * lstProducts.ItemHeight + 3
    End If
    lstProducts.Visible = True
    lstProducts.SelectedIndex = 0
    lstProducts.Focus()
End If
End Sub
```

The program limits the height of the ListBox control to 10 items. If the control contains fewer items, the height of the control is smaller. While the ListBox control is visible, the user can't switch to another control. They can only select a product by pressing Enter, or hide the ListBox control without selecting a product by pressing Escape. When the Escape key is pressed, the program moves the focus to the txtID control, where the user can select another product by specifying its ID. These actions take place from within the ListBox control's KeyUp event handler.

Another interesting part of the code is the KeyUp event handler of the ListView control. While this control has the focus, the user can perform the following actions:

- ◆ Select a row to edit by pressing Enter. The current row's fields are displayed on the TextBox controls and the focus is moved to the txtQty control. There the user can revise the quantity—or can press Enter to move to the next control and revise the discount—and press Enter once again to commit the changes to the grid control.
- ◆ Delete the current row by pressing the Delete key.
- ◆ Return to the txtID control on the form to enter a new row by pressing Escape.

These actions are handled from within the ListView control's KeyUp event handler, which is shown in Listing 18.15:

LISTING 18.15: HANDLING USER ACTIONS ON THE LISTVIEW CONTROL

```
Private Sub ListView1_KeyUp(ByVal sender As Object, _
    ByVal e As System.Windows.Forms.KeyEventArgs) _
    Handles ListView1.KeyUp
    If e.KeyCode = Keys.Delete Then
        If ListView1.SelectedItems.Count > 0 Then
            Dim selIndex As Integer = _
```

```
                ListView1.SelectedIndices(0)
                ListView1.SelectedItems(0).Remove()
                If selIndex = ListView1.Items.Count Then
                    selIndex = selIndex - 1
                End If
                If selIndex >= 0 Then
                    ListView1.Items(selIndex).Selected = True
                End If
                ShowTotals()
            End If
            If e.KeyCode = Keys.Escape Then
                ClearFields()
                txtID.Focus()
            End If
            If e.KeyCode = Keys.Enter Then
                If ListView1.SelectedItems.Count > 0 Then
                    ShowSelectedRow()
                    SetEditColors()
                    txtQty.SelectAll()
                    EditedID = ListView1.SelectedIndices(0)
                    txtQty.Focus()
                End If
            End If
        End Sub
```

A detail line is committed to the grid when the user presses Enter in the txtDiscount TextBox control. When this happens, the code validates all the fields and then adds a new item to the ListView control. If the txtID box isn't empty, then a product has been selected and we must only make sure that the user has specified a positive quantity. If not, the execution of the event handler is aborted.

If it's a new line, the program creates a new item for the ListView control and appends it to the control. If it's an existing line that was selected for editing, the program replaces the original item with the new one. Notice how the code handles the discount. If it's a number less than 1, it accepts it as is. If it's larger than one, it divides it by 100. To specify a discount of 32%, you can enter a value like 0.32, or 32. Both values will be translated into 0.32 when the discount is inserted into the DataRow object. The KeyUp event handler of the txtDiscount TextBox Control accepts a new detail line and inserts it into the ListView control (or updates an existing line on the same control). It is shown in Listing 18.16.

LISTING 18.16: ADDING A NEW DETAIL LINE TO THE GRID

```
Private Sub txtDiscount_KeyUp(ByVal sender As Object, _
    ByVal e As System.Windows.Forms.KeyEventArgs) _
    Handles txtDiscount.KeyUp
    If e.KeyCode = Keys.Enter Then
        If txtID.Text.Trim = "" Or _
```

```
        Val(txtQty.Text) = 0 Then Exit Sub
    txtSubTotal.Text = _
        FormatNumber(CalculateLineTotal(), 2)
    Dim LI As New ListViewItem()
    Dim newItemIndex As Integer
    newItemIndex = ListView1.Items.Count
    LI.Text = txtID.Text
    LI.SubItems.Add(txtDescription.Text)
    LI.SubItems.Add(txtPrice.Text)
    LI.SubItems.Add(txtQty.Text)
    If CDec(txtDiscount.Text) >= 1 Then
        txtDiscount.Text = CDec(txtDiscount.Text) / 100
    End If
    LI.SubItems.Add(FormatNumber(txtDiscount.Text, 2))
    LI.SubItems.Add(txtSubTotal.Text)
    If EditedID = -1 Then
        ListView1.Items.Add(LI)
        ListView1.EnsureVisible(newItemIndex)
    Else
        ListView1.Items(EditedID).Text = LI.Text
        ListView1.Items(EditedID).SubItems(0).Text = _
            LI.SubItems(0).Text
        ListView1.Items(EditedID).SubItems(1).Text = _
            LI.SubItems(1).Text
        ListView1.Items(EditedID).SubItems(2).Text = _
            LI.SubItems(2).Text
        ListView1.Items(EditedID).SubItems(3).Text = _
            LI.SubItems(3).Text
        ListView1.Items(EditedID).SubItems(4).Text = _
            LI.SubItems(4).Text
        ListView1.Items(EditedID).SubItems(5).Text = _
            LI.SubItems(5).Text
        EditedID = -1
    End If
    ShowTotals()
    ClearFields()
    ResetEditColors()
    txtID.Focus()
End If
End Sub
```

There are additional pieces of code in the application that we didn't present here. You can download the application and examine its code. It's a little lengthy, but the interface is very intuitive and can be used as is with a bar-code reader. Just program the reader to emit three Enter keystrokes after the product's ID and the new row will be automatically committed to the grid with the default quantity of 1. You can use this project as your starting point for building an application to issue invoices

and add the necessary features to it. Such features may include a standard discount for all the lines, or a discount that's determined by the customer to whom the invoice is issued (the customer ID selected on the first tab of the application's interface). The application allows you to enter multiple rows with the same product ID. You can change this behavior by searching all the items on the control for each new product ID entered by the user. If such a row exists, you can add the new row's quantity to the existing quantity. If no row with the same ID exists on the grid, you can add a new item to the control. Before submitting the order to the database, however, the code reduces multiple detail lines that refer to the same product into a single one, with the appropriate quantity.

COMMITTING AN ORDER TO THE DATABASE

When the user clicks the Save Invoice button to submit the changes to the database, the code eliminates rows that contain the same product and creates a single row with the sum of all quantities for each product. This is necessary because the OrderID and ProductID fields form a unique key in the Order Details table. The same order can't contain two (or more) rows that refer to the same product. Even if this constraint didn't exist in the database, it's always a good idea to clean up each order before committing it to the database. Listing 18.17 shows the Click event handler of the Save Invoice button.

The handler of Listing 18.17 calls the ReduceRows() subroutine to combine rows that refer to the same product. Then it iterates through the rows of the ListView control and creates a new OrderedProduct object for each row (that is, each detail line of the order). These rows are added to the `orderedItems` ArrayList. This array, along with the ID of the selected customer and the ID of the selected employee, is passed to the AddOrder method, which commits the order to the database.

LISTING 18.17: COMMITTING AN ORDER TO THE DATABASE

```
Private Sub btnAccept_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnAccept.Click
    ReduceRows()
    Dim orderedItems As New ArrayList()
    Dim P As OrderClass.OrderedProduct
    Dim i As Integer
    For i = 0 To lvOrderGrid.Items.Count - 1
        P = New OrderClass.OrderedProduct
        P.ProductID = CInt(lvOrderGrid.Items(i).Text)
        P.ProductPrice = CDec(lvOrderGrid.Items(i).SubItems(2).Text)
        P.ProductQTY = CInt(lvOrderGrid.Items(i).SubItems(3).Text)
        P.ProductDiscount = _
            CDec(lvOrderGrid.Items(i).SubItems(4).Text) /
            orderedItems.Add(P)
    Next
    Dim INV As OrderClass
    If INV.AddOrder(lstCustomers.SelectedValue, _
        cmbEmployees.SelectedValue, orderedItems) Th
        Dim reply As DialogResult
        MsgBox("'Ordered saved successfully." & vbCrLf & _
            "Press OK to prepare a new order")
```

lvOrderGrid.Items.Clear()

[Team Fly](#)

 Previous

Next 

```
        ClearFields()
        TabControl1.SelectedIndex = 0
        lstCustomers.Focus()
    Else
        MsgBox("Failed to record order." & vbCrLf & _
            "The current order will be cleared " & _
            "and you must create a new one")
        lvOrderGrid.Items.Clear()
        ClearFields()
        txtID.Focus()
    End If
End Sub
```

The ReduceRows() subroutine eliminates multiple rows that refer to the same product; its implementation is shown in Listing 18.18. The code uses two nested loops to compare all items to one another. The outer loop of the subroutine iterates through all the items in the ListBox control, and the inner loop iterates through the items that are below the current item of the outer loop. In effect, each item is compared with all following items. If the product ID values of two items match, then the second item is removed and its quantity is added to the quantity of the first item.

LISTING 18.18: THE REDUCEROWS SUBROUTINE

```
Private Sub ReduceRows()
    Dim idx As Integer
    While idx < lvOrderGrid.Items.Count
        Dim i As Integer = idx + 1
        While i < lvOrderGrid.Items.Count
            If lvOrderGrid.Items(idx).Text = _
                lvOrderGrid.Items(i).Text Then
                lvOrderGrid.Items(idx).SubItems(3).Text = _
                    (CInt(lvOrderGrid.Items(idx).SubItems(3).Text) + _
                    CInt(lvOrderGrid.Items(i).SubItems(3).Text)).ToString
                lvOrderGrid.Items(idx).SubItems(4).Text = _
                    Math.Max(CDec(lvOrderGrid.Items(idx). _
                    SubItems(4).Text), _
                    CDec(lvOrderGrid.Items(i).SubItems(4).Text)).ToString
                lvOrderGrid.Items(i).Remove()
            Else
                i = i + 1
            End If
        End While
        idx = idx + 1
    End While
    lvOrderGrid.Sorting = SortOrder.Ascending
    lvOrderGrid.Sort()
End Sub
```


After collapsing the duplicate rows, the code commits the new order to the database through a transaction. First, it creates an `ArrayList`, the `OrderedItems` `ArrayList`, with the order's detail lines. Each detail line is represented by an object of the `OrderClass.OrderedProduct` type. The `ArrayList` and a few fields that represent the order's header are passed as arguments to the `AddOrder` method. The following statement calls the `AddOrder` method of the middle tier component:

```
Dim INV As OrderClass
If INV.AddOrder(lstCustomers.SelectedValue, _
                cmbEmployees.SelectedValue, OrderedItems) Then
```

The first two arguments are the IDs of the customer and the employee and the third argument is the `ArrayList` with the order's details. If the `AddOrder` method returns a positive value, the order has been added successfully to the database. A negative value means that the update operation failed.

The `NWOrders` application contains quite a bit of code we're not going to discuss in this chapter. The colors of the controls change to indicate the current operation, and there's code to handle the `Enter` and `Escape` keys in most operations. The totals at the bottom of the form are updated as new detail lines are entered or removed. The interface of the application is intuitive and the application will work as-is with a bar-code reader. If you program the bar-code reader to emit three `Enter` keystrokes after reading the product ID, users can keep entering detail lines without even touching the keyboard. Notice that you can move from one `TextBox` to the other with the `Tab` key as well, but you must press `Enter` at the `Discount TextBox` control to add the current detail line to the `ListView` control.

The same interface can be used with different databases as well. All you really need in order to sell, or take orders, are the product descriptions, their prices, and their discounts. The presentation tier code communicates with the database through three simple methods, and you can easily modify the application's middle tier to make it work with a different database. The middle tier component isolates the structure of the database from the presentation tier; you need only edit the middle tier's code to make the application work with any other database. We were able to reuse the `NWOrders` application with a large production database of books, where the product IDs are the ISBNs, their descriptions are the book titles, and the prices were obtained with a stored procedure that takes into consideration offers and special customer discounts. Actually, in the following section we're going to add a middle tier component to calculate discounts based on past purchases by the same customer.

Adding a Business Rule

In this section we'll add a "true" middle tier component to our application. The new component will calculate the discount of each product with a business rule. This rule will be different for different companies and we should be able to deploy it independently of the invoicing application. The company's management may change the discount policy at any time, and you should be able to deploy this component without recompiling and reinstalling the client application on every workstation.

Let's start with the business rule. Each item's discount is determined by two pieces of information: who's buying and what they're buying. With this information we can implement different discount policies in our code. We can use lookup tables based on customers or products (or both), create groups of customers and assign discounts to each group, and so on. In this example we will implement a component that accepts a customer ID and a product ID and calculates the discount

[Team Fly](#)

 Previous

Next 

based on the sales of the same product to the same customer in the past. Customers who have purchased more items of the same product in the past will get an extra discount. Specifically, we'll offer an additional 1% for every \$100 a customer has spent on the same item, with a maximum discount of 24%.

This rule is implemented with the `GetItemDiscount` method of the `BusinessLayer` class. We're adding a new class to the project so that we can deploy it on an application server. The business rule shouldn't be embedded in the client application, because changes in the company's business rules would require recompiling and redeploying the revised application on every workstation. If the business rule is implemented in a module that all clients can access, then we can change this module at a single location and all clients will see the revised business rule. In the revised project, the `NWOrders-BLayer` project, the discount business rule is implemented with a stored procedure, so it's trivial to revise and deploy the business rule. As long as you don't add new arguments to the stored procedure, you can change it at any time and all clients will calculate discounts with the new rule.

THE BUSINESSLAYER MIDDLE TIER COMPONENT

Our business rule is implemented with the `GetItemDiscount` method of the `BusinessLayer` class. The code of the class is shown in Listing 18.19.

LISTING 18.19: THE BUSINESSLAYER CLASS

```
Public Class BusinessLayer
    Public Shared Function GetItemDiscount( _
        ByVal CustomerID As String, _
        ByVal ProductID As Integer) As Integer
        Dim CMD As New SqlClient.SqlCommand()
        CMD.CommandText = 'GetItemDiscount'
        CMD.CommandType = CommandType.StoredProcedure
        CMD.Parameters.Add(New SqlClient.SqlParameter(_
            "@ProductID", ProductID))
        CMD.Parameters.Add(New SqlClient.SqlParameter(_
            "@CustomerID", CustomerID))
        Dim CN As New SqlClient.SqlConnection()
        CN.ConnectionString = _
            "initial catalog=Northwind; " & _
            "integrated security=SSPI; " & _
            "persist security info=False; "
        CN.Open()
        CMD.Connection = CN
        Dim discount As Integer
        discount = CMD.ExecuteScalar
        ' additional discount processing statements here !
        Return CType(discount, Integer)
    End Function
End Class
```

As you can see, the middle tier component is quite trivial: it executes the `GetItemDiscount` stored procedure passing as argument the ID of a product and the ID of the customer placing the order. The `GetItemDiscount` stored procedure is shown in Listing 18.20; you must attach this stored procedure to the database.

LISTING 18.20: THE `GETITEMDISCOUNT` STORED PROCEDURE

```
CREATE PROCEDURE GetItemDiscount
@CustomerID nchar(5),
@ProductID int
AS
DECLARE @CustomerTotal int
SET @CustomerTotal =
    (SELECT ROUND(SUM(unitprice*quantity*(1-
discount)),0,0)
FROM      [Order Details] INNER JOIN Orders
          ON on Orders.OrderID = [Order Details].OrderID
WHERE     Orders.CustomerID = @CustomerID AND [Order
Details].ProductID = @ProductID)
IF @CustomerTotal IS NULL
    SELECT 12
ELSE
    BEGIN
    IF @CustomerTotal < 1200
        SELECT 12 + @CustomerTotal / 100
    ELSE
        SELECT 24
    END
```

Notice that the `BusinessLayer` component isn't really necessary in our example, because the discounts are actually implemented by a stored procedure. When the discount business rule changes, we can simply change the stored procedure. Our business rule covers most cases, as discounts depend on two pieces of information: who's buying and what they're buying. To implement a more complicated discount policy, you may need a middle tier component that performs most of the calculations, as opposed to delegating the task of calculating each customer's discount to a stored procedure.

The presentation tier of the revised `NWOrders` application is identical to the presentation tier of the original application, with the exception of the statement that displays the discount. To calculate the discount of an item, we call the `GetItemDiscount` method of the `BusinessLayer` component passing the IDs of the customer and the product. The value returned by the `GetItemDiscount` method is displayed on the appropriate `TextBox` control on the form and copied onto the grid when the detail line is finalized:

```
txtDiscount.Text = BLayer.GetItemDiscount(_
lstCustomers.SelectedValue, txtID.Text.Trim).ToString("'##.00")
```

To load the DataSet we used four DataAdapters, one for each table. The DataAdapters were configured with the following SQL statements:

DAProducts

```
SELECT    ProductID, ProductName
FROM      Products
ORDER BY  ProductName
```

DACustomers

```
SELECT    CustomerID, CompanyName
FROM      Customers
```

DAOrders

```
SELECT    OrderID, CustomerID
FROM      Orders
ORDER BY  CustomerID, OrderID
```

DADetails

```
SELECT    OrderID, ProductID, UnitPrice, Quantity, Discount
FROM      [Order Details]
```

Notice that we downloaded all the customers and all products, including products that have never been ordered and customers who have never placed an order. You will have to configure your DataAdapters to select the rows that meet the user-specified criteria and not load unneeded data to the client.

The ListBox control on the left is bound to the Products DataTable of the ProductSales DataSet. We could have populated the ListBox control from within our code with a few simple statements. The control's DisplayName property is set to the ProductName column and the ValueMember property is set to the ProductID column. Users will select products by name, but we should be able to retrieve the selected product's ID quickly and use it to extract from the DataSet the sales data. The data-bound properties of the ListBox control are set as follows:

Property	Setting
DataSource	ProductSales1.Products
DisplayName	ProductName
ValueMember	ProductID

The Application's Code

When the form is loaded, the tables of the ProductSales DataSet are populated through their respective DataAdapters from within the form's Load event handler, shown in Listing 18.21. The last two statements force the selection of the first item in the ListBox control (the name of the first product

[Team Ely](#)

 Previous

Next 

in the list). This list is bound to the Products DataTable, and it's automatically populated as soon as the DataTable is filled.

LISTING 18.21: LOADING THE DATA AT THE CLIENT

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles MyBase.Load
    DAProducts.Fill(ProductSales1, "'Products")
    DACustomers.Fill(ProductSales1, "Customers")
    DAOOrders.Fill(ProductSales1, "Orders")
    DADetails.Fill(ProductSales1, "Order Details")
    ' Force the SelectedIndexChanged event of the ListBox control
    ListBox1.SelectedIndex = -1
    ListBox1.SelectedIndex = 0
End Sub
```

The core of the application is the ListBox control's SelectedIndexChanged event handler, which is shown in Listing 18.22. Every time the user selects another product in the ListBox control, the event handler displays the names of the customers that have ordered the specific product, along with some totals.

LISTING 18.22: DISPLAYING SALES DATA ABOUT THE SELECTED PRODUCT

```
Private Sub ListBox1_SelectedIndexChanged( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles ListBox1.SelectedIndexChanged
    ListView1.Items.Clear()
    Dim productID As Integer
    Dim OrderCount, ItemCount, Revenue As Decimal
    Dim TotalOrders, TotalCount, TotalRevenue As Decimal
    productID = ListBox1.SelectedValue
    Dim DetailRows() As DataRow
    ' Get all detail lines that refer to the selected product
    DetailRows = _
        ProductSales1.Order_Details.Select("ProductID = " & productID)
    Dim DetailRow As ProductSales.Order_DetailsRow
    Dim OrderID As Integer
    Dim LI As ListViewItem
    Dim OrdersTable As New DataTable()
    ' Make a new DataTable with the same schema
    ' as the Orders table
    OrdersTable = ProductSales1.Orders.Clone
    Dim OrderRow As ProductSales.OrdersRow
    Dim OrderRows() As DataRow
```

```
' The following loop extracts all the orders that contain
' the selected product and adds them to the OrdersTable DataTable
For Each DetailRow In DetailRows
    OrderRow = DetailRow.GetParentRow('OrdersOrder_Details")
    OrdersTable.Rows.Add(OrderRow.ItemArray)
Next
' Sort selected orders by Customer ID
OrderRows = OrdersTable.Select(" ", "CustomerID")
Dim currentCustomerID As String = " "
ItemCount = 0 : OrderCount = 0 : Revenue = 0
For Each OrderRow In OrderRows
    Dim CustomerID As String
    Dim CustomerRow As ProductSales.CustomersRow
    CustomerID = OrderRow.Item("CustomerID")
    OrderID = OrderRow.Item("OrderID")
    DetailRow = _
        ProductSales1.Order_Details._
        FindByOrderIDProductID(OrderID, productID)
    If CustomerID <> currentCustomerID And _
        currentCustomerID <> "" Then
        LI = New ListViewItem
        CustomerRow = _
            ProductSales1.Customers.FindByCustomerID(Customer
        LI = MakeListItem(CustomerRow.CompanyName, _
            OrderCount, ItemCount, Revenue)
        If ListView1.Items.Count Mod 2 = 0 Then
            LI.BackColor = Color.Beige
        Else
            LI.BackColor = Color.Gainsboro
        End If
        ListView1.Items.Add(LI)
        TotalOrders = TotalOrders + OrderCount
        TotalCount = TotalCount + ItemCount
        TotalRevenue = TotalRevenue + Revenue
        OrderCount = 0 : ItemCount = 0 : Revenue = 0
    End If
    currentCustomerID = CustomerID
    OrderCount = OrderCount + 1
    ItemCount = ItemCount + DetailRow.Quantity
    Revenue = Revenue + DetailRow.Quantity * _
        DetailRow.UnitPrice * (1 - DetailRow.Discount)
Next
LI = New ListViewItem
LI = MakeListItem("TOTAL", TotalOrders, TotalCount, TotalRevenue)
LI.Font = New Font(LI.Font.Name, LI.Font.Size + 2, FontStyle.Bold)
LI.BackColor = Color.LightGreen
ListView1.Items.Add(LI)
End Sub
```


Add these Imports statements to the very top of your code window:

```
Imports System.IO
Imports System.Runtime.Serialization.Formatters
Imports System.Runtime.Serialization.Formatters.Binary
```

In the following example, you create an object and display three of its properties in the output window. Then the program changes the properties, serializes the object, and streams it to a file. Finally, the object is streamed back to the program and deserialized, and the new properties are displayed in the output window, just to show you that the process actually works. Add this class (Listing 2.15) to your code window.

LISTING 2.15: SERIALIZING AND DESERIALIZING COMPLICATED DATA

```
<Serializable()> Public Class MyObject

    Public a As String = "This thing goes"
    Public b As String = "into a stream, then back."
    Public c As Integer = 120

End Class
```

Then type this into the Form1_Load event:

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

    Dim obj As New MyObject()

    Console.WriteLine(obj.a)
    Console.WriteLine(obj.b)
    Console.WriteLine(obj.c)
    Console.WriteLine()

    obj.a = "Incoming!!"
    obj.b = "I've been serialized."
    obj.c = 44
    Dim fs As New FileStream("c:\test.txt", FileMode.Create, File
    Dim BF As New Binary.BinaryFormatter()

    BF.Serialize(fs, obj)
    fs.close()

    'read it back

    Dim BF1 As New Binary.BinaryFormatter()
    Dim fs1 As New FileStream("c:\test.txt", FileMode.Open, File
```

authors, onto a ListView control. The application doesn't interact with the user, it simply maps a set of related data onto a Windows controls. If you used a DataGrid to display the same information, users would be able to view only one level of data at any one time (only publishers, or only a publisher's titles, or only the authors of a single title).



FIGURE 18.7 Mapping publishers, titles, and authors on a ListView control

The structure of the Relations1 project is similar to that of the Relations project, so we'll discuss briefly its architecture and code. The application contains four DataAdapters that load the following tables to the `DSTitles` client DataSet. The four DataAdapters are configured for selecting rows from their corresponding tables, and not updating these tables, with the following statements:

```
DAPublishers DataAdapter
    SELECT    pub_id, pub_name
    FROM      publishers

DATitles DataAdapter
    SELECT    title_id, title, pub_id, price
    FROM      titles

DAAuthors DataAdapter
    SELECT    au_id, au_lname, au_fname
    FROM      authors

DATitleAuthor DataAdapter
    SELECT    au_id, title_id, au_ord
    FROM      titleauthor
```

The application loads the entire tables to the DataSet, which isn't something you want to do with a production database. You can add a WHERE clause to all of the SELECT statements shown here to limit the number of rows. Since the four tables are related, the same WHERE clause will apply to

[Team Ely](#)

 Previous

Next 

all statements (limit the number of publishers, or retrieve titles of selected authors along with their publishers, and so on).

Once you've created the structure of the `DSTitles` DataSet by configuring the DataAdapter objects, you must establish relations between its tables. The relations are identical to the ones that exist in the database, but you must drop the compound key of the `TitleAuthor` table and create two new keys on the `au_id` and `title_id` fields.

When the Load All Titles button is clicked, the program populates the DataSet and maps its rows to the ListView control with the statements of Listing 18.23:

LISTING 18.23: MAPPING PUBLISHERS, TITLES, AND AUTHORS ON A LISTVIEW CONTROL

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

    DsTitles1.Clear()
    ListView1.Items.Clear()
    DAPublishers.Fill(DsTitles1, "Publishers")
    DAAuthors.Fill(DsTitles1, "Authors")
    DATitles.Fill(DsTitles1, "Titles")
    DATitleAuthor.Fill(DsTitles1, "TitleAuthor")

    Dim LI As ListViewItem
    Dim PubRow As DSTitles.publishersRow
    For Each PubRow In DsTitles1.publishers
        LI = New ListViewItem()
        LI.Text = PubRow.pub_name
        Dim BookRow As DSTitles.titlesRow
        Dim BookRows() As DSTitles.titlesRow
        BookRows = PubRow.GetChildRows("PublishersTitles")
        For Each BookRow In BookRows
            LI.SubItems.Add(BookRow.title)
            Dim TitleAuthorRow As DSTitles.titleauthorRow
            Dim TitleAuthorRows() As DSTitles.titleauthorRow
            TitleAuthorRows = _
                BookRow.GetChildRows("TitlesTitleAuthor")
            For Each TitleAuthorRow In TitleAuthorRows
                Dim AuthorRow As DSTitles.authorsRow
                AuthorRow = _
                    TitleAuthorRow.GetParentRow("AuthorsTitleAuthor")
                LI.SubItems.Add(AuthorRow.au_lname & _
                               ", " & AuthorRow.au_fname)
            End For
            ListView1.Items.Add(LI)
            LI = New ListViewItem()
            LI.Text = " "
            LI.SubItems.Add(" ")
        End For
    End For
```

Next

The regular expression shown in Figure 19.2 matches all eight-character words in the text, one at a time. After specifying the search pattern, click the Find First Match button to locate the first match in the text. After locating the first match in the text, keep clicking the Find Next Match button to locate the next match.

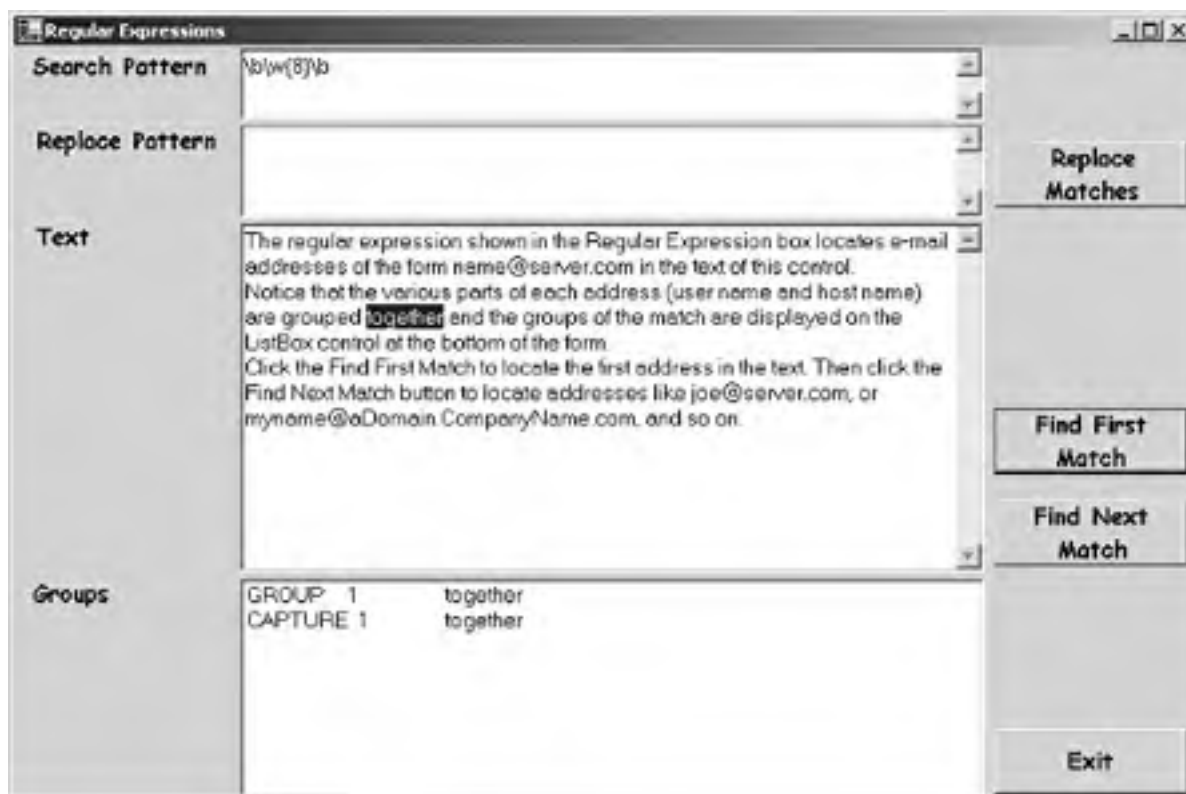


FIGURE 19.2 Matching regular expressions with the RegularExpressions project

To test the projects, and create some of the examples, we've used some large text files, which we downloaded from the Project Gutenberg site (<http://promo.net/pg/>). This site is an on-line library and contains hundreds (perhaps thousands) of books in text format for on-line browsing, or downloading to your computer.

Let's start with a few examples so that you'll understand what regular expressions are and how they're used. Any string can be considered a generalized regular expression. The pattern "Visual" will match all the instances of this word in the text. The search is case-sensitive by default, but you can make it case-insensitive by setting an option (you'll see shortly the objects for working with regular expressions). Of course, this isn't why we're using regular expressions. A regular expression contains one or more metacharacters: characters with special meaning. The most basic metacharacter is the period, which stands for any character. The following expression will locate all two-letter words that end with "f":

`.f`

and the following expression will locate all five-letter strings that begin with "t" and end with "e":

t...e

This expression will not locate words, but strings; it will locate much more than five-letter words. It will also locate passages such as "to get," "the end," and so on. To match words only, use the `\w` metacharacter in the place of the period metacharacter:

t\w\w\we

[Team Fly](#)

 Previous

Next 

WHAT'S REGULAR ABOUT THESE EXPRESSIONS?

Would you expect that the following non-sensical string is called a regular expression? Probably not. (By the way, it's an expression that will match all dates in a text.)

```
\b\d{1,2}\\/\d{1,2}\\/(\\d{2}|\\d{4})\b
```

It's probably one of the last things you'd call regular, but the name stems from the fact that such expressions normalize general search patterns. Anyway, this is their name, and writing a correct regular expression is a real challenge. Constructing the correct regular expression for a specific pattern is like putting together a puzzle.

The `\w` metacharacter indicates a word character; that is, a character other than space and punctuation symbols. Metacharacters are mixed freely with regular characters in a regular expression. Many metacharacters are letters prefixed with the backslash.

The last expression is not perfect either. Although it will not return matches that span multiple words, it will match the "tinue" in "continued." To match complete words, use the `\b` metacharacter, which specifies the beginning or end of a word. The following expression will match all five-letter words that begin with "t" and end with "e":

```
\bt\w\w\we\b
```

This expression will locate words such as "there," "three," "theme," and so on, but not parts of larger words ("continued") or multiple words ("to get," "the end").

To avoid repeating a metacharacter multiple times, as we did in the last couple of examples, you can use a *quantifier*. There are many types of quantifiers; one of the most common is the `{n}` quantifier, which means that the preceding expression must be matched exactly *n* times. The expression that matches five-letter words that begin with "t" and end with "e" can be written as:

```
\bt\w{3}e\b
```

Here's how you read this regular expression: the desired matches must be words (delimited by the `\b` metacharacter) and they should start with the character "t," followed by three characters (any three characters) and ending with the character "e."

NOTE Since we're using the `\b` metacharacter to indicate the beginning and end of words, you'd expect that the `\w` metacharacter is now equivalent to the period. This isn't the case, and here's why: Placing the `\b` metacharacter at the two ends of a regular expression doesn't limit the match to a single word. The first instance of the metacharacter means the beginning of a word and the second instance means the end of a word. Taken together, they don't mean the beginning and end of a single word. If you replace the `\w` metacharacter in the regular expression with the period, you may locate matches that span multiple words. The regular expression `\bt.{9}e\b` will match word sequences such as "through the" and "they may be." The expression `\bt\w{9}e\b`, however, will match only nine-character words.

Our next example, a regular expression that locates all ISBNs in a text, demonstrates the range operator. Instead of "any" character, or a specific character, you can specify a range of characters in a pair of square brackets. The following expression matches any vowel (depending on whether you count "y" as a vowel or not, of course):

```
[aeiou]
```

[Team Fly](#)

 Previous

Next 

There are two more overloaded forms of the `Regex` class's constructor, which are shown next:

```
Dim RX As Regex(pattern)
Dim RX As Regex(pattern, options)
```

where *pattern* is the regular expression to be matched against the text and the *options* argument specifies various search options. The *options* argument determines how the search will be performed (for example, whether the search will be case-sensitive) and is discussed shortly.

Using the `RX` object, you can apply a regular expression against a text and retrieve the corresponding matches. The `Matches` method of the `Regex` class accepts as arguments the text to be searched and the regular expression, and returns the matches in a collection, the `MatchCollection` object. To apply the regular expression stored in the `strSearch` variable to the text on the `TextBox1` control, use the following statements:

```
Dim RX As Regex
Dim allMatches As MatchCollection
Try
    allMatches = RX.Matches(TextBox1.Text, strSearch)
Catch exc As Exception
    MsgBox(exc.Message, MsgBoxStyle.OKOnly, _
        "INVALID REGULAR EXPRESSION")
    Exit Sub
End Try
```

The exception handler will catch any errors in the regular expression itself (i.e., an invalid regular expression). If the specified regular expression is invalid, an exception is thrown and you must catch it in your code. The `allMatches` collection exposes the usual members of a collection, which you can use to iterate through the matches. Each member of the collection is a `Match` object, and we'll examine the members of the `Match` object shortly.

The `Matches` method is overloaded. Its simplest form accepts the two arguments shown in the example: the string to be searched and a regular expression. The other overloaded forms of the method are:

```
Regex.Matches(string)
Regex.Matches(string, pattern, options)
Regex.Matches(string, startIndex)
```

To use the form of the `Match` method that accepts a single argument (the text to be searched), you must instantiate the `Regex` class with a constructor that accepts a regular expression as argument (the *pattern* argument in the overloaded forms shown above).

The *options* argument of the `Regex` constructor lets you adjust the search method by specifying one or more of the options shown in Table 19.1. The available options are members of the `RegexOptions` enumeration, and they're shown next. You can combine multiple options with the OR operator. Some of the explanations refer to topics that are discussed later in the chapter and you may have to return to this table after reading about groups and captures.

[Team Fly](#)

 Previous

Next 

TABLE 19.1: THE REGEXOPTIONS ENUMERATION

MEMBER	DESCRIPTION
Compiled	The regular expression is compiled for faster execution. The compilation process increases the startup time, but subsequent searches with the same regular expression are executed more quickly.
CultureInvariant	Forces the methods of the RegularExpressions class to ignore the current culture. Case-insensitive operations are always culture-sensitive. The methods get the specifics of the current culture from the property Thread.CurrentCulture.
ECMAScript	Enables ECMAScript-compliant behavior for the expression. This flag can be used only in conjunction with the IgnoreCase, Multiline, and Compiled options, otherwise an exception will be thrown.
ExplicitCapture	Specifies that the only valid captures are explicitly named or numbered groups of the form (?<name>...).
IgnoreCase	Specifies that the search is case-insensitive (by default, the search is case-sensitive).
IgnorePatternWhiteSpace	Eliminates the white space from the pattern and enables comments marked with #.
Multiline	Specifies multi-line mode. In this mode the ^ and \$ characters match the beginning and end of every line.
None	Uses the default options.
RightToLeft	Specifies that the search will be performed from the end of the text and proceed to the beginning of the text.
SingleLine	Specifies single-line mode. The only difference between single-line and multi-line mode is that in single-line mode the period matches every character, including the newline character.

You may not need all the matches at once. For example, you may want to process the current match before locating the next one. The `Regex` class allows you to locate one match at a time with the `Match` method, which returns a `Match` object that represents the first match. After that, you can call the `NextMatch` method of the `Match` object returned by the `Match` method to retrieve the next match. This method returns a `Match` object as well, and you can continue searching through the text for the same regular expression by calling this object's `NextMatch` method. It's actually simpler than it sounds and you'll see this technique in the section "Using the `Match` and `NextMatch` Methods" later in this chapter.

You can locate all instances of the word "expression" in your text by specifying this literal as the regular expression. Whether you will locate instances of the same word in different cases depends on the setting of the IgnoreCase option. To locate only the sentences that begin with the word "Expression", specify the regular expression ^Expression and turn on the RegEx object's Multiline option.

[Team Fly](#)

 Previous

Next 

```
obj = BF1.Deserialize(fs1)
fs1.Close()

Console.WriteLine(obj.a)
Console.WriteLine(obj.b)
Console.WriteLine(obj.c)
```

End Sub

What does this look like on the hard drive? Load it into Notepad and here's what you get:

```
'           ^  CTrans, Version=1.0.1202.12130,
  Culture=neutral, PublicKeyToken
=null'      Trans.MyObject~ 'a'b'c' ' ^ ~
  Incoming!!'  I  ve been serialized.,
```

This is "binary" serialization, but, as you can see, some text survives the grinder. To serialize in a purely text format, use the XML (SOAP) serialization covered in Chapter 17. XML serialization ignores any private fields (binary serialization saves both private and public fields).

Mixing Types into the Same Stream

This next example illustrates how you can shove disparate items into a serialization/deserialization process. This example sends a structure, followed by an arraylist into the same stream. (See the previous example for the Imports statements required.)

Add the structure and Form_Load code in Listing 2.16 to your code window.

LISTING 2.16: MIXING DISPARATE DATA STRUCTURES DURING SERIALIATION

```
<Serializable()> Public Structure Hat
    Dim Name As String
    Dim Size As Integer
    Dim Price As Decimal
End Structure
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load

    Dim d As New Hat
    d.Name = "Bowler"
    d.Size = 9
    d.Price = 44.98

    Dim MyArray As New ArrayList

    MyArray.Add( "Sandbag" )
    MyArray.Add( "Bigbag" )
    MyArray.Add( "Paperbag" )
```

Notice that the `Options` property of the `Regex` object is read-only. The options must be specified in the `Regex` object's constructor, as follows:

```
RXOptions = RegexOptions.Compiled Or _  
            RegexOptions.Multiline Or _  
            RegexOptions.IgnoreCase  
RX = New Regex(txtPattern.Text, RXOptions)
```

When you call the `Matches` property of the `RX` object, you will retrieve only the sentences that begin with "Expression". To catch all instances of the word, even those that start with a lowercase character, we usually turn on the `IgnoreCase` option, as shown in the previous sample.

The `Match` object contains information about the current match, and its basic members are described in Table 19.2.

TABLE 19.2: THE MOST IMPORTANT MEMBERS OF THE MATCH CLASS

PROPERTY	DESCRIPTION
<code>Index</code> Property	Returns the starting position of the first character of the match in the text
<code>Length</code> Property	Returns the length of the match
<code>Value</code> Property	Returns the matched string in the text
<code>Success</code> Property	Returns a Boolean value indicating whether the match is successful
<code>NextMatch</code> Method	Retrieves the next match in the text, following the current match

The `Match` object exposes other members as well, which you can look up in the documentation. The members you'll be using most often in your code are shown in Table 19.2.

Using the `Matches` Method

The statements in Listing 19.1 retrieve all the matches of a regular expression in a collection, the `allMatches` collection, and then iterate through the items of the collection and print the matched text, its starting location in the string, and its length.

LISTING 19.1: ITERATING THROUGH THE MATCHES COLLECTION

```
Dim myText As String = TextBox1.Text  
Dim RX As System.Text.RegularExpressions.Regex  
Dim searchPattern As String = "\be\w{3}\b"  
Dim match As System.Text.RegularExpressions.Match  
Dim matchValue As String  
Dim matchStart, matchLength As Integer  
Dim allMatches As System.Text.RegularExpressions.MatchCollection  
allMatches = RX.Matches(myText, searchPattern, _  
                    System.Text.RegularExpressions.RegexOptions.IgnoreCase)
```

For Each match In allMatches

[Team Fly](#)

 Previous

Next 

```
matchValue = match.ToString
machStart = match.Index
machLength = match.Length
Console.WriteLine(' MATCH: " & matchValue & _
                  " at location: " & machStart & _
                  ", length: " & machLength)
```

Next

The text to be searched is the `myText` variable and the regular expression is stored in the `searchPattern` variable. The expression `\be\w{3}\b` locates all four-letter words that begin with the character "e" ("exit," "else," and so on). Notice the second argument passed to the `Matches` method, which specifies that case should be ignored. This means that the regular expression will match "exit," "Exit," and "EXIT." If you omit this argument, then the search will be case-sensitive. The code segments shown in this section fully qualify the member names. You should import the `System.Text.RegularExpressions` namespace to shorten the names of the members of the `Regex` class and save yourself some typing.

Using the Match and NextMatch Methods

You can also match a regular expression against some text and retrieve the matches one at a time with a loop like the one shown in Listing 19.2. The `Match` method is called initially to retrieve the first match. The code enters a loop that keeps calling the `NextMatch` method of the `currentMatch` object, which represents the current match. If the method locates another match, the `Success` property of the same object is set to `True` and the loop repeats. With each iteration, the code prints the current match on the Output window. When there are no more matches in the text, the `While` loop terminates because the `Success` property of the `currentMatch` object is set to `False` and the number of matches is printed.

LISTING 19.2: LOCATING ONE MATCH AT A TIME

```
Dim RX As System.Text.RegularExpressions.Regex
Dim searchPattern As String = "\be\w{3}\b"
RX = New System.Text.RegularExpressions.Regex(searchPattern, _
      System.Text.RegularExpressions.RegexOptions.IgnoreCase)
Dim matches As Integer
Dim currentMatch As System.Text.RegularExpressions.Match = _
      RE.Match(TextBox1.Text)
While currentMatch.Success
    matches += 1
    Console.WriteLine(" MATCH: " & currentMatch.Value & _
                      " at location: " & currentMatch.Index.ToString & _
                      ", length: " & currentMatch.Length.ToString)
    currentMatch = currentMatch.NextMatch
End While
Console.WriteLine(" Found " & matches.ToString & " matches ")
```


The text being searched shouldn't be edited between calls to the NextMatch method. The FindFirst method of the RegEx class locates all the matches at once and the NextMatch method simply returns the subsequent matches. As a result, the NextMatch will report incorrect matches if you edit the text. Let's say you're searching for all three-letter words in the following sentence (the matches are in boldface in the example):

A brown **fox** jumped over **the** lazy **dog**.

Let's say you locate the first three-letter in the text and then you edit the sentence by adding the word "high" after the verb. The following three matches will be located as shown here:

A brown **fox** jumped high **over** the **lazy** dog.

The matches reported by the NextMatch method are at the locations of the original matches in the text. If you need to process the matches in the text as you go along, use two copies of the string you're searching, or the Replace method of the RegEx class.

The Split Method

The Split method of the RegEx object is similar to the Split method of the String class, but it splits the specified input string at the positions defined by a regular expression. The regular expression can be specified in the RegEx object's constructor, or as an argument to the Split method, as shown in the two overloaded forms of the method:

```
RegEx.Split(text)  
RegEx.Split(text, pattern)
```

The Split method returns an array of strings, which are the parts of the original string. For example, you can split a text into segments based on paragraph numbers (such as 1.1, or 5.3.2.4) or section headers ("chapter," "section," "paragraph," and so on). Splitting a text with such general criteria is impossible with the simpler Split method of the String class. Whereas the Split method of the String class uses specific delimiters (periods, space, tab, etc.) to split a string, the Split method of the RegEx object uses more general expressions (for example, the string "Section" or "Chapter" followed by a number, or a series of numbers separated by periods).

The Replace Method

This method replaces all the matches of a regular expression with another pattern. There are many overloaded forms of the Replace method, the simpler one being the following:

```
RegEx.Replace(text, replacement)
```

where the first argument is the original string (the string where all the replacements will take place) and the second argument is the replacement string. The Replace method doesn't modify the text directly. Instead, it returns the modified string. The regular expression that will be used to locate the matches is specified in the constructor of the RegEx object.

The `replacement` argument can be a literal, but it can also be a regular expression. This method is a very flexible and powerful tool and you'll see some interesting examples later in this chapter, when we'll discuss how to group the parts of a match.

Another, even more powerful form of the `Replace` method is the following:

```
Regex.Replace(text, matchEvaluator)
```

[Team Fly](#)

 Previous

Next 

The `matchEvaluator` argument is a delegate (a user-specified procedure) that is called at each match to return the replacement string. The declaration of the delegate is shown next:

```
Public Delegate Function _  
    MatchEvaluator(ByVal match As Match) As String
```

Every time a match is found, the `MatchEvaluator` delegate is called; it returns the replacement string for the match passed as argument. You can create very elaborate search and replacement procedures with this form of the `Replace` method. To use it, you must provide the code of the `MatchEvaluator` function. In the function's body you can access the current match through the `match` argument, then create a string that will be used by the `Replace` method to replace the match. The replacement string is the function's return value. The use of the `MatchEvaluator` delegate is demonstrated in the discussion of the `MatchEvaluator` project, later in this chapter.

To experiment with replacement operations with regular expressions, use the `RegExEditor` or the `RegularExpressions` project. You will see how the `Replacement` method is used in the discussion of the code of these two projects. A typical example is the formatting of phone numbers. A document may contain phone numbers with the area code in parentheses or not, with hyphens or periods between the groups of digits, and so on. Using the `Replace` method of the `RegEx` class, you can apply a uniform formatting to all phone numbers in the text—a format such as `(nnn) nnn.nnnn`, where `ns` are the appropriate digits.

The replacement string can be a literal, but in most cases it's either a part of the match or a transformed part of the match (you'll have to provide a `MatchEvaluator` delegate to transform the match). To specify a part of the match, you must use one of the symbols shown in Table 19.3 below.

TABLE 19.3: COMMON REPLACEMENT METACHARACTERS

METACHARACTER	DESCRIPTION
<code>\$\$</code>	Substitutes the "\$" literal
<code>\$&</code>	Substitutes a copy of the entire match
<code>\$`</code>	Substitutes the part of the input string before the match
<code>\$'</code>	Substitutes the part of the input string after the match
<code>\$+</code>	Substitutes the most recently matched group
<code>\$_</code>	Substitutes the entire input string

Let's say the text is the following sentence:

```
Match and replace words
```

If you search for the word "and" and then replace the match with `$'` you'll get the sentence:

```
Match replace words replace words
```

If you apply the replacement pattern \$ _ to the same text, the result will be the following sentence:

Match Match and replace words replace words

[Team Fly](#)

 Previous

Next 

THE MATCHEVALUATOR PROJECT

The MatchEvaluator project demonstrates how to use the Replace method with a MatchEvaluator delegate to perform elaborate replacement operations. As you already know, it is possible to perform search and replace operations using regular expressions. A regular expression allows you to locate substrings based on general patterns, rather than literals, and use parts of these expressions to build elaborate replacement strings. However, the RegularExpressions class doesn't provide any string or date manipulation functions you may need to perform even more elaborate replacements. You have seen how to locate short dates in the text (dates in the form mm/dd/yy or mm/dd/yyyy). However, what if you want to replace these instances with long dates that include day and month names? Visual Basic provides tools to format days in all possible ways, but how can we access this functionality from within the RegEx.Replace method?

The solution is to write a custom function that accepts a Match object as argument and returns the replacement string. The Replace method can call this function for every match and retrieve the replacement string. In the function's body you're free to use any of the Visual Basic functions, or the functionality of any of the Framework's classes. One of the overloaded forms of the Replace method accepts as arguments the original text, the search pattern, and a delegate. The Replace method first calls the delegate, a function that retrieves the actual replacement string, and then performs the replacement.

Let's say you want to replace a string with the same text, but set the first character of each word in uppercase. First you must write a function that accepts a Match object as argument and returns a string that will replace the corresponding match in the original text. Here's such a function:

```
Private Function CamelCase(ByVal m As Match) As String
    Dim str As String = m.ToString
    Return Char.ToUpper(str.Chars(0)) + str.Substring(1, str.Length - 1)
End Function
```

This function accepts an argument of the Match type, processes the text of the match, and returns the uppercase of the first character in the string, followed by the rest of the string. To use it with the Replace method, create a variable of the MatchEvaluator type:

```
Dim MatchEval As System.Text.RegularExpressions.MatchEvaluator
```

MatchEval is a delegate, which must be associated with a function (the CamelCase() function in this example):

```
MatchEval = _
    New System.Text.RegularExpressions.MatchEvaluator( _
        AddressOf CamelCase)
```

Now you can call the Replace method passing the MatchEval variable in the place of the replacement string:

```
Dim result As String  
result = Regex.Replace(txt, pattern, MatchEval)
```

Here's how this form of the Replace method works: It locates all the matches of the specified pattern in the `txt` string. For each match, it calls the function specified by the `MatchEval` delegate. The function returns a string, which the Replace method uses as the replacement string for the current

match. If the pattern is the regular expression `\w+`, the Replace method will locate all words in the text and then will replace each word in the text with the same word in different case.

Figure 19.3 shows the main form of the MatchEvaluator project. The upper TextBox contains the original text and the lower TextBox displays the result of the replace operation. You can specify the search pattern in the Search Pattern box, select the type of replacement you want to perform by checking the appropriate radio button, and then click the Replace button. The search pattern locates dates and the Replace method replaces the short dates with the same dates in the long date format.



FIGURE 19.3 Performing elaborate replacement with a MatchEvaluator delegate

Listing 19.3 shows the definitions of the delegates implemented in the MatchEvaluator project.

LISTING 19.3: THE FUNCTIONS THAT WILL BE USED AS DELEGATES IN THE MATCHEVALUATOR PROJECT

```
Private Function CamelCase(ByVal m As Match) As String
    Dim str As String = m.ToString
    If str.Length > 0 Then
        Return Char.ToUpper(str.Chars(0)) + _
            str.Substring(1, str.Length - 1)
    Else
        Return str
    End If
End Function

Private Function UpperCase(ByVal match As Match) As String
    Return match.ToString.ToUpper
End Function

Private Function LowerCase(ByVal match As Match) As String
```

```
Return match.ToString.ToLower  
End Function
```

[Team Fly](#)

 Previous

Next 


```
Private Function LongDate(ByVal match As Match) As String
    Dim mDate As Date
    Try
        mDate = CType(match.ToString, Date)
    Catch
        Return "*" & match.ToString & "*"
    End Try
    Return CDate(match.ToString).ToLongDateString
End Function
```

The first three functions are trivial; they change the case of the characters in their argument. In the CamelCase() function's code we examine the length of the string that's passed as argument, because certain regular expressions may return a zero length string. If we attempt to extract the first character of a zero-length string, an exception will be thrown. The If statement, in effect, prevents this exception (you could have used a structured exception handler).

The LongDate() function converts its argument to a Date value and returns it, after formatting it as a long date. Notice the use of the structured exception handler: if the match passed to the function isn't a valid date value, the function returns its argument embedded in asterisks, to indicate an error condition. You should probably return the match as it was passed to the function, because if you use this delegate with the wrong regular expression (a regular expression that returns numbers, or words, for example), the replace operation will fill the text with asterisks, because the matches will not be dates.

When the Replace Text button is clicked, the code shown in Listing 19.4 is executed. First, it creates the appropriate MatchEvaluator delegate, depending on which of the radio buttons is selected. Then it passes this delegate to the Replace method, along with the search pattern and the text to be searched. The string returned by the Replace method is assigned to the second TextBox control on the form.

LISTING 19.4: PERFORMING REPLACE OPERATIONS WITH DELEGATES

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

    Dim pattern As String
    Dim MatchEval As System.Text.RegularExpressions.MatchEvaluator

    If rdLower.Checked Then
        MatchEval = New System.Text.RegularExpressions. _
            MatchEvaluator(AddressOf LowerCase)
    End If
    If rdUpper.Checked Then
        MatchEval = New System.Text.RegularExpressions. _
            MatchEvaluator(AddressOf UpperCase)
```

```
End If
If rdCamel.Checked Then
    MatchEval = New System.Text.RegularExpressions. _
        MatchEvaluator(AddressOf CamelCase)
End If
If rdLongDates.Checked Then
    MatchEval = New System.Text.RegularExpressions. _
        MatchEvaluator(AddressOf LongDate)
End If

pattern = txtPattern.Text.Trim
Dim result As String = _
    Regex.Replace(TextBox1.Text, pattern, MatchEval)
TextBox2.Text = result
End Sub
```

To test the other delegates, specify the following regular expression, which matches individual words, and then select one of the options that changes the word's case:

```
\b\w{1,}\b
```

Each match (that is, each word in the text) is passed to the appropriate delegate, which returns the same string in different case (Figure 19.4).

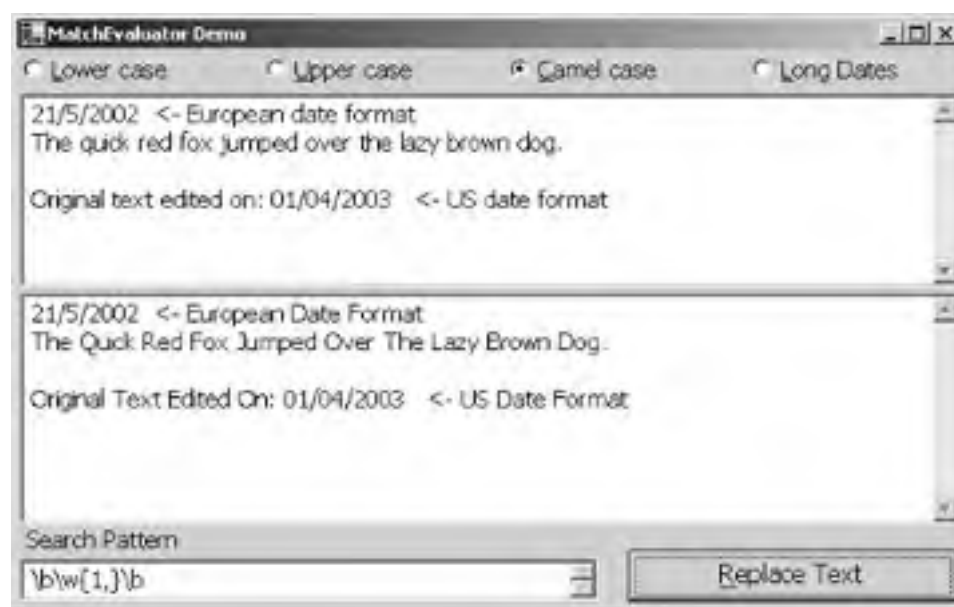


FIGURE 19.4 Changing the case of the text with a custom MatchEvaluator

The MatchEvaluator project demonstrates the use of custom MatchEvaluator delegates in performing elaborate replacement operations with regular expressions, but it's hardly useful on its own. To make the most of the RegEx class's Replace method, you must write a function that accepts a Match object as argument and returns a new string that will replace the current match. The function's return value could be any string, but it's usually derived from the current match. Once this function is in place, you can pass to the Replace method a delegate to this function.

will match "since" (three characters followed by the characters "c" and "e," but it will also match "once" (there's a space in front of the word). To limit the match to words, use the `\w` metacharacter in the place of the period metacharacter. The expression:

```
\w\w\wce
```

(which is equivalent to "`\w{3}ce`") will locate "since" and "twice," but not "once." The problem with this pattern is that it will also match "sources." In the section "Anchors" later in this chapter, you'll learn how to specify word limits too.

To locate all e-mail addresses in a text, you can specify the following pattern: one or more characters, followed by the "@" symbol, followed by one or more characters, a period, and then one or more characters. The following is a simple regular expression for locating e-mail addresses:

```
\w+@\w+\.\w+
```

The expression `\w+` stands for any number of word characters. In effect, the expression `\w+` locates a word (one or more word characters). In the realm of regular expressions, a word is a string delimited by spaces and/or punctuation symbols. This word should be followed by the @ symbol, another word, the period, and another word. The period has special meaning in a regular expression; to locate a period, you must turn off the special meaning of the period by prefixing it with a slash character. This expression will locate simple e-mail addresses of the form `name@server.com`. Mail addresses can be more complicated than this and you will see a more general e-mail matching pattern later in this chapter.

The `\W` metacharacter matches any non-word character. This metacharacter matches the first or last character in a word, as well as the carriage return and line feed characters. In effect, it's equivalent to the following range operator `[^\w]`, which stands for all characters except upper/lowercase characters, digits, and the underscore:

```
[^A-Za-z_0-9]
```

The caret (^) symbol negates the following characters, or range of characters (see the following section for more information on character ranges).

Ranges of Characters

Instead of a specific character, or the "any character" metacharacter, you can specify a range of characters in square brackets. The following expression stands for all vowels:

```
[aeiou]
```

To locate three consecutive vowels in the text, use the expression:

```
[aeiou]{3}
```

If the characters in the range are consecutive, you can use the range operator (the minus symbol) between the first and last character in the range. The following expression locates a numeric digit:

[0-9]

[Team Ely](#)

 Previous

Next 

```
Dim fs As New FileStream( 'c:\test.txt", FileMode.Create, Fi
Dim BF As New Binary.BinaryFormatter

'do the multiple serialization:
BF.Serialize(fs, d) 'store the hat object
BF.Serialize(fs, MyArray) 'store the ArrayList
fs.Close()

End Sub
```

NOTE A structure in VB.NET replaces the traditional VB user-defined type. A structure is similar to, though more flexible than, the user-defined type. Now, to make things even more confusing, the term type has a far different meaning in .NET than it did in previous versions of VB. Type now means objects, fields, enumerations, and various other elements of an assembly—in other words, type includes pretty much everything you might manipulate during design-time. (Note this section's title, "Mixing Types into the Same Stream.") Finally, OOP professors tell us not to use structures at all; instead, you are to use classes, which are similar but offer more features. Besides, OOP professors just like classes in general, and suggest you use them universally, even when dreaming.

Reading Back Mixed Data

There are two requirements when deserializing. First, you must deserialize the items *in the same order* that you serialized them. In this case, you sent the Hat structure in, then the ArrayList. So you pull them back out in the same order.

Second, the Deserialize method returns only object variables, so you must cast each incoming object into the correct type (you can use the CType command). Listing 2.17 deserializes the file created in the previous example (Listing 2.16).

LISTING 2.17: READING DISPARATE DATA STRUCTURES VIA DESERIALIZATION

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

    Dim d1 As New Hat
    Dim ar As New ArrayList

    Dim BF1 As New Binary.BinaryFormatter
    Dim fs1 As New FileStream("c:\test.txt", FileMode.Open, FileA

    d1 = CType(BF1.Deserialize(fs1), Hat)
    ar = CType(BF1.Deserialize(fs1), ArrayList)

    fs1.Close()
```

You can negate a range of characters by prefixing it with the caret (^) symbol. The following expression stands for all characters but vowels:

```
[^aeiou]
```

Notice that the previous expression doesn't stand for consonants—it includes numeric digits and punctuation symbols, in addition to consonants. To search for all characters except numeric digits, use the following expression:

```
[^0-9]
```

White Space and Metacharacters

The `\s` metacharacter matches any white space (spaces, horizontal and vertical tabs, line feeds). The `\S` metacharacter matches any non-white space. Avoid the `\S` metacharacter, as it will match any character in the text. If you want to match specific characters that produce white space (space, tab, line feed, and so on), use one of the following metacharacters. These metacharacters represent common non-printable characters and they're shown in Table 19.4:

TABLE 19.4: WHITE SPACE METACHARACTERS

METACHARACTER	DESCRIPTION
<code>\a</code>	bell (beep)
<code>\n</code>	newline
<code>\r</code>	carriage return
<code>\t</code>	tab
<code>\f</code>	formfeed
<code>\e</code>	escape

Quantifiers

A regular expression may contain one or more quantifiers, which determine how many times the preceding element should be matched. The quantifiers are shown in Table 19.5:

TABLE 19.5: QUANTIFIER METACHARACTERS

METACHARACTERS	DESCRIPTION
<code>?</code>	Matches the preceding element zero or one times
<code>*</code>	Matches the preceding element zero or more times
<code>+</code>	Matches the preceding element one or more times
<code>{num}</code>	Matches the preceding element num times exactly
<code>{min, max}</code>	Matches the preceding element at least min times, but not more

than max times

{min, }

Matches the preceding element at least min times

[Team Fly](#)

 Previous

Next 

The difference between the `?` and `*` quantifiers is that the `?` quantifier doesn't match two or more consecutive instances, while the `*` does. The following expression locates Social Security numbers in the form XXX-XX-XXXX. The groups may be separated by dashes and/or spaces. The following expression will locate an SSN whether it's written 213-46-8915, or 213 - 46 - 8915, or 213 46 8915, or even 213468915.

```
\d{3}[ -]*\d{2}[ -]*\d{4}
```

This regular expression will locate strings that are made up of three groups of digits: the first group must have three digits (leading zeros can't be skipped in Social Security numbers), the second group must have two digits, and the last group must have four digits. These groups can be separated by any number of dashes and/or spaces. The element `[-]` means either a space or the dash. Moreover, this element may appear one or more times. This regular expression will match Social Security numbers even if they're entered as 213--46--8915, or as 213- -46--8915.

GREEDY VERSUS NON-GREEDY PATTERNS

The `*` metacharacter is used to create "greedy" expressions. A greedy expression doesn't stop at the first match, but attempts to find the longest possible match. In other words, the match returned by a greedy expression may contain multiple instances of the character(s) to which the quantifier applies. Let's say you want to locate all the tags in an XML (or HTML) document. Let's also assume that each element of the XML file is stored on a separate line. The following pattern will locate any string starting with the opening bracket, followed by any number of characters and ending with the closing bracket:

```
<.*\>
```

This is a greedy pattern: it will not stop at the first closing bracket it encounters. When you apply it to a document that contains the following line:

```
<ProductID>101</ProductID>
```

it will report the entire line as a single match, because it starts with an opening bracket and ends with a closing one. Between them, there are many characters. One of them happens to be the closing bracket, but a greedy pattern treats it as a regular character, because there's another one later in the line.

To turn this pattern into a "non-greedy" one (a pattern that will search for the minimum, not the maximum, number of characters), write it as:

```
\<.*?\>
```

The question mark is another metacharacter that matches the preceding expression zero or one time, but no more. If you apply the new pattern to the same XML document, it will report two matches for the previous line:

```
<ProductID>
```

and

</ProductID>

[Team Fly](#)

 Previous

Next 

Greedy expressions are quite often desirable. Let's say you're searching for uppercase words with a regular expression such as the following:

```
\b[A-Z]{2,}\b
```

This expression locates words made up of two or more uppercase characters. If the matched words are consecutive, you may wish to locate them all in a single match. If you want to locate `VISUAL BASIC` as a single match, and not as two matches, you need a greedy expression like the following:

```
(\b[A-Z]{2,}\b\s{0,})+
```

This expression locates strings of two or more uppercase characters followed by any number of spaces. Any succession of words in uppercase with white space between them will be captured in a single match.

Anchors

In addition to specifying a pattern and a quantifier, we can also specify where in the text the pattern should appear—the position at which a particular pattern occurs. To specify the location of the pattern in the text you can use one of the anchor metacharacters shown in Table 19.6.

TABLE 19.6: ANCHORING METACHARACTERS

METACHARACTER	MEANING
<code>^</code>	Start of line. The pattern <code>^[0-9]</code> matches lines that begin with a number.
<code>\$</code>	End of line. The pattern <code>[0-9]\$</code> matches lines that end with a number.
<code>\b</code>	Word boundary (matches any character at the beginning of a word).
<code>\B</code>	Non-word boundary (matches any character not at the beginning of a word).

The `^` metacharacter has a whole different meaning when it appears in the range operator (after the opening square bracket). The `^` and `$` metacharacters are among the most useful ones when it comes to replace operations. They allow you to select the entire line and replace it with something else.

The following pattern locates lines that begin with a number followed by a period (such as paragraph numbers or section numbers):

```
^\d?\.
```

The regular expression that matches numbers searches for one or more digits. The following pattern will locate all the words that begin with the character "x":

```
\bx\w*\b
```

The `\b` metacharacter at the beginning and the end of the expression indicates that matches should occur at a word boundary. The next character in the pattern ("x") is the first character of the desired match and it should be followed by any number of word characters.

[Team Fly](#)

 Previous

Next 

Escaping Metacharacters

If you want to treat the metacharacters as regular characters in a pattern, you must "escape" them: place a backslash ("\") in front of a metacharacter to reverse its special meaning (in other words, treat it as a normal character instead of as a metacharacter). To locate all lines that start with a number followed by a period, use the following pattern:

```
^\d?\. 
```

To locate lines that contain URLs, you can use this pattern:

```
http://\//www
```

This regular expression will match only the protocol part of the URL, as shown by the boldfaced section of the following example:

```
http://www .sybex.com
```

To catch the entire URL use the following expression, which locates the same patterns as the previous one, followed by anything that's not space:

```
http://\//www\S*
```

Of course, not all URLs begin with "www," but this short example shows how ugly a regular expression with escaped characters can get. The characters in Table 19.7 have special meaning in regular expressions, and you must escape them if you want to match them as literals. All other characters match themselves.

TABLE 19.7: ESCAPING METACHARACTERS

.	Period	\.		pipe symbol	\
\$	Dollar sign	\\$	slash	\\	
^	Caret	\^	*	Asterisk	*
{	Opening curly bracket	\{	+	plus symbol	\+
[Opening square bracket	\[?	question mark	\?
(Opening parenthesis	\()	closing parenthesis	\)

Notice that the closing square bracket need not be escaped, because its meaning in a regular expression is determined by the matching opening square bracket. To locate an expression in square brackets, use the following expression:

```
\[.*]
```

Alternation

The alternation metacharacter allows you to search for the "either/or" type of matches. The pipe symbol (|) matches one of the characters (or expressions) it separates. The following expression will match either "u" or "ou":

(o | ou)

[Team Ely](#)

 Previous

Next 

The following regular expression will match both "Petroutsos" and "Petrutsos":

```
Petr(ou|u)tsos
```

Use similar expressions to match words with alternate spellings, or frequently misspelled words. To locate the word "color" in the text, use the following regular expression:

```
(color|colour)
```

or:

```
col(o|ou)r
```

To match the words "gray" or "grey," use the following regular expression:

```
gr(a|e)y
```

We've covered a lot of ground so far and you should have a good grasp of regular expressions and the RegEx class. Let's look at a practical example that demonstrates many of the topics discussed so far.

The RegExEditor Project

The RegExEditor application is a simple text editor capable of searching with regular expressions, as well as literals. Figure 19.1 shows the main form of the application, where the text is entered and edited, and the Find & Replace dialog box, where you specify what to search for. RegExEditor is a text editor based on the functionality of the TextBox control and that of the RegularExpressions namespace. In this chapter we're not going to discuss the basic operations of the editor, just the code that implements the features related to regular expressions.

The Find & Replace dialog box of the RegExEditor project can treat the search pattern as a literal (like the Editor project) or as a regular expression. The Use Regular Expressions check box is checked by default. Even so, if the search expression doesn't contain any metacharacters, the program will search the text for a literal. The Replace and Replace All buttons replace the current match, or all matches, in the text with the specified string.

Let's start with the code of the Find button. The code starts by examining the state of the Use Regular Expressions check box. If this control is cleared, the code escapes the search string to cancel the effect of any metacharacters in the string. Then the code checks the state of the Case Sensitive box and sets the `searchOptions` variable. This value is the second argument of the RegularExpressions class's constructor and it determines how the Match method will search the text. This argument is of the RegexOptions type and can be one of the values of the RegexOptions enumeration (or a combination of the enumeration's members). Listing 19.5 shows the code of the Find button's Click event handler:

LISTING 19.5: THE FIND BUTTON'S CODE

```
Private Sub btnFind_Click(ByVal sender As System.Object, _
```

```
ByVal e As System.EventArgs) _  
    Handles btnFind.Click  
Dim searchPattern As String  
If chkRegex.Checked Then  
    searchPattern = searchWord.Text
```



```
Else
    searchPattern = RegEx.Escape(searchWord.Text)
End If
Dim searchOptions As RegexOptions
If Not chkCase.Checked Then
    searchOptions = RegexOptions.Multiline Or _
                    RegexOptions.IgnoreCase
Else
    searchOptions = RegexOptions.Multiline
End If
Try
    RegEx = New RegEx(searchPattern, searchOptions)
Catch exc As Exception
    MsgBox(exc.Message, MsgBoxStyle.OKOnly, _
           "INVALID REGULAR EXPRESSION")
    Exit Sub
End Try
currentMatch = RegEx.Match(EditorForm.txtBox.Text)
If Not currentMatch.Success Then
    MsgBox("Can't find word")
    Exit Sub
End If
```

The Find Next command calls the NextMatch method of the RegEx object to locate the next match of the regular expression in the text (Listing 19.6). The NextMatch method always returns a Match object, even if no match is found. In this case, the Success property of the Match object is False. If a match is found, the corresponding text on the editor's form is highlighted.

LISTING 19.6: THE FINDNEXT BUTTON'S CODE

```
Private Sub btnFindNext_Click(ByVal sender As System.Object, _
                              ByVal e As System.EventArgs) _
    Handles btnFindNext.Click
    currentMatch = RegEx.Match(EditorForm.txtBox.Text, _
                               EditorForm.txtBox.SelectionStart + _
                               EditorForm.txtBox.SelectionLength)
    If Not currentMatch.Success Then
        MsgBox("No more matches")
        Exit Sub
    End If
    Dim selStart As Integer = currentMatch.Index
    EditorForm.txtBox.Select(selStart, currentMatch.Length)
    EditorForm.txtBox.ScrollToCaret()
End Sub
```

The Replace All button's code uses the Replace method of the RegEx object to replace all the matches with a literal (if the Use Regular Expression option is cleared), or with another regular expression (if the Use Regular Expression option is checked). Listing 19.7 gives the code behind the Replace All button:

LISTING 19.7: THE REPLACE ALL BUTTON'S CODE

```
Private Sub btnReplaceAll_Click(ByVal sender As System.Object, _
                                ByVal e As System.EventArgs) _
    Handles btnReplaceAll.Click
    Dim curPos, curSel As Integer
    curPos = EditorForm.txtBox.SelectionStart
    curSel = EditorForm.txtBox.SelectionLength

    If chkRegEx.Checked Then
        Dim searchOptions As RegexOptions
        searchOptions = RegexOptions.Multiline
        If Not chkCase.Checked Then
            searchOptions = searchOptions Or RegexOptions.IgnoreCase
        End If
        RegEx = New System.Text.RegularExpressions.Regex( _
            searchWord.Text, searchOptions)
        EditorForm.txtBox.Text = RegEx.Replace( _
            EditorForm.txtBox.Text, replaceWord.Text)
    Else
        If searchWord.Text <> And replaceWord.Text <> Then
            EditorForm.txtBox.Text = _
                EditorForm.txtBox.Text.Replace( _
                    searchWord.Text, replaceWord.Text)
        End If
    End If
    EditorForm.txtBox.Select(curPos, curSel)
    EditorForm.txtBox.ScrollToCaret()
    btnFindNext.Enabled = False
    btnReplace.Enabled = False
    btnReplaceAll.Enabled = False
End Sub
```

The Replace button replaces the currently selected text on the editor with another literal, or regular expression. To replace a single match, the Replace method is passed the currently selected, not the entire, text of the control (Listing 19.8).

We'll also look at regular expressions that allow you to specify the surrounding characters of a pattern without actually matching anything more than the desired pattern. For example, you can search for dates and match only the year, or month, part of the date.

Grouping and Backreferences

Sometimes we're not interested in just a match, but also in identifying segments of a match. Let's say you're locating e-mail addresses in a text file with the following expression:

```
[\\w\\.\\-]+@[\\w\\.\\-]+\\.\\w+
```

This regular expression locates a string made of word characters, periods, and/or dashes, followed by the ampersand symbol, followed by another string with the same structure, followed by a period and another word made up of lower/uppercase characters. The first string in the match is the user's name, the second string is the domain name, and the last string is the suffix of the domain name (com, org, net, and so on). If you want to capture these substrings as separate entities, enclose them in parentheses, as in the following regular expression:

```
([\\w\\.\\-]+)@([\\w\\.\\-]+)\\.([\\w+)
```

It's the same regular expression as before: it locates the same matches, but it maintains information about each group of the match. A group is a segment of the match that corresponds to a regular subexpression in parentheses. The username is the first group of the last regular expression, the domain name is the second group, and the domain suffix is the third group. Once you group the results of a match, you can access each part of the e-mail address.

Enter the last regular expression in the Search Pattern box at the top of the RegularExpressions application and an e-mail address like `name@server.com` in the Text box on the form. Then click the Find First Match button and you will see the following items in the Groups box near the bottom of the form (ignore the items with the "CAPTURE" prefix):

```
GROUP 1    name@server.com
GROUP 2    name
GROUP 3    server
GROUP 4    com
```

If you search for the non-grouped pattern shown at the beginning of this section, only one group will be matched and the following item will appear in the Groups box:

```
GROUP 1    name@server.com
```

To access the groups in a match, you can use the elements of the Groups collection of the Match object. The following loop iterates through the groups of the object `thisMatch`, which represents a regular expression match:

```
If thisMatch.Groups.Count > 0 Then
    Dim igrp As Integer
    For igrp = 0 To thisMatch.Groups.Count - 1
        Console.WriteLine("GROUP" & igrp.ToString & _
            vbTab & thisMatch.Groups(igrp).Value)
    Next
End If
```

[Team Fly](#)

 Previous

Next 

If you apply the regular expression that matches e-mail addresses and groups the parts of the address to some text that contains e-mail addresses, you will see that the first item in the Groups collection is the entire match and the following items are the groups, as specified by the pairs of parentheses in the regular expression. Figure 19.5 shows the RegularExpressions application matching this regular expression against the text in the large TextBox control. For the time being, ignore the captures listed in Figure 19.5; we'll discuss captures in the following section.

The groups in a match are identified with a number, or a name. To name each group, use the question mark followed by a name in a pair of angle brackets immediately after the group's opening parenthesis. The following regular expression is the same as the one shown in Figure 19.5, but it names the three groups. The names are `User`, `Domain`, and `Suffix`:

```
(?<User>[\w\.-]+)@(?(?<Domain>[\w\.-]+)\.(?<Suffix>\w+)
```



FIGURE 19.5 Using groups in a regular expression

REPLACEMENT OPERATIONS WITH GROUPED MATCHES

If you don't supply your own names, the groups will be named with a number (1 for the first group, 2 for the second, up to 9). But why bother naming the groups? One good reason is that you can refer to them in substitution operations. Instead of replacing a match with a fixed string, you can replace the match with a string that's based on the match itself. Let's look at another example of a fairly advanced replacement operation, this time with the RegExEditor project.

Start the RegExEditor project and enter some text that contains phone numbers, like that shown in Figure 19.6. Then open the Find & Replace dialog box and enter the following search pattern:

```
\({0,1}\(\d{3})\) {0,1}[ -]{0,}\(\d{3}) [ -]{0,}\(\d{4})
```

This pattern locates phone numbers and returns three groups for each match: the area code, the first three digits of the phone number, and the remaining four digits of the phone number. The search pattern will locate all instances of phone numbers with area codes, regardless of how they're formatted. The area code may or may not be enclosed in parentheses, there may or may not be a



FIGURE 19.6 Using regular expressions in replacement operations. The original text (top) and the same text after the replacements (bottom).

space between the area code and the phone number, and there may or may not be a separator between the two groups of the phone number.

The first character is the opening parenthesis, which is a special character and must be prefixed by a backslash: `\`. The opening parenthesis may appear zero or one times, but no more. The first group of the regular expression matches a group of three digits. This is the number's area code; it may be followed by a closing parenthesis. The next character can be one or more spaces and dashes: `[-] { 0 , }`. The following characters of the regular expression, `(\d { 3 })`, attempt to locate another group of three digits, followed by one or more spaces and/or dashes. The last part of the regular expression is a group of four digits.

Then specify the following replacement pattern:

($\$1$) $\$2$.\$ $\$3$

If you click the Replace button, all phone numbers in the text will be formatted as:

(nnn) nnn.nnnn

where n is a digit. The following text:

Home number: (343) 445-3434
Office numbers: 800 777-3434, 435 8883999
Cellular number: 111-4453222

[Team Ely](#)

 Previous

Next 

will become:

```
Home number:      (343) 445.3434
Office numbers:   (800) 777.3434, (435) 888.3999
Cellular number:  (111) 445.3222
```

You can also refer to a group later in the same regular expression by number or by name. The group's identifier must appear in a pair of parentheses and be prefixed by a forward slash. One rather common regular expression with groups is the following, which locates consecutive duplicate words in a text (as in "words in in a text"):

```
\b(\w+) (\1)\b
```

This regular expression looks for a word, which is captured as a group, followed by a space, which is followed by the first group of the match. The subexpression `(\1)` refers to the first grouped match of the regular expression, which is a word. The entire expression will match identical words separated by a space. If you want to allow for any white space between the words (multiple spaces, tabs), you should replace the single space with the `\s` metacharacter followed by a quantifier:

```
\b(\w+)\s+(\1)\b
```

Knowing how to locate consecutive duplicate words in a text is useful, but it would be even more useful to be able to remove the second instance of the same word. If you want to get rid of the repeated word, use the following replacement pattern:

```
$1
```

(an odd replacement pattern, but it specifies that the entire match should be replaced by the first group of the match). Of course, not all duplicate words that may appear in the text are in error ("... a chance that that same evening..." is a part of a correct sentence that contains repeated words).

***TIP** We tested our regular expressions by applying them to large documents and all kinds of special cases surfaced during the tests. Getting the correct regular expression takes a few trials, and we recommend that you test your regular expressions with long documents that are likely to contain a large number of matches. For example, you may write a regular expression to match e-mail addresses that works with the sample document you provide, but misses addresses that contain periods in the username.*

Referring to a previous subexpression in the same match is also known as *backreference*. Backreferences are references to earlier groups of the same match by their number, and you can refer to the last nine groups in the same match as `\1`, `\2`, etc. up to `\9`. Notice that you can't make backreferences by name. Named groups can be used in a replacement pattern, but not in the same regular expression you're using to search the text.

Another example along the same lines is the formatting of dates in a standard format throughout a document. The following regular expression will locate dates in the text:

```
\b(?:<month>\d{1,2})[/|-](?:<day>\d{1,2})[/|-](?:<year>\d{2,4})\b
```

The regular expression isn't going to validate the dates (it will happily locate a date like 13/13/2004, which is clearly an invalid date in the western calendar). However, it will accept dates with backslashes and/or dashes as separators. Each part of the date is a different group of the match, because they're

[Team Fly](#)

 Previous

Next 

enclosed in a pair of parentheses. The groups are named as well, and we'll use their names in the replacement operation. Let's say you want to format dates as ' 'yyyy-mm-dd.' This type of formatting simplifies sorting and comparison operations. To format the dates in the text, you can use the following replacement pattern:

```
${year}-${month}-${day}
```

If you supply the search and replacement patterns shown earlier to the following text:

```
Date1 12/3/2001  
Date2 4-11-2004  
Date3 8/1/04
```

the edited text will become:

```
Date1 2001-12-3  
Date2 2004-4-11  
Date3 04-8-1
```

TIP *By the way, there's no regular expression that will format month and day numbers with leading zeros (at least, none that we're aware of). The RegEx class doesn't expose elementary string manipulation methods, such as a method to convert to upper/lowercase. To apply special formatting to the matches, or convert the matched string to uppercase, you should use the MatchEvaluator delegate, as discussed in the section "The Replace Method," earlier in this chapter.*

One last, and quite practical, example of grouped matching is the parsing of a file with delimited fields, like the following:

```
value1 23.5  
value2      -19.1  
value3      3.34  
value4      2.89
```

Some of the rows in this example are separated by a single space and others by multiple spaces; the last two rows use tabs and spaces to separate the two fields. In all cases, the separator is white space. Use the following regular expression to locate rows that contain data in this space-delimited format and capture the two fields as separate groups:

```
(?<name>[\w\.]*) (\s+(?<value>-?[\d\.]*) )
```

The first field (name) can't have spaces ("value1" is a valid name, but "value 1" isn't). The second field is made up of the (optional) minus symbol, followed by one or more digits, a decimal period, and then a few more digits.

Enter the text and the regular expression shown here in the appropriate boxes of the Regular-Expressions program. Then click the Find First Match button to locate the first match. The RegularExpressions project will report the following groups and captures:

GROUP 1	value1	23.5
CAPTURE1	value1	23.5
GROUP2	23.5	
CAPTURE1	23.5	
GROUP3	value1	

[Team Fly](#)

 Previous

Next 

```
CAPTURE1    value1
GROUP4      23.5
CAPTURE1    23.5
```

The following metacharacters allow you to group the matches. The grouping metacharacters appear in pairs, and the most important grouping pair are the parentheses. Grouping is used in substitutions and we'll talk a little about substitutions with regular expressions. When you group a match, you're "capturing" it (it's saved in a register so that you can use it later in the expression). The capturing and substitution of matches can get quite complicated, so we will only scratch the surface of this topic.

() Captures the matched substring. The captured subexpressions are numbered automatically based on the order of the opening parentheses, starting from one. The first capture, capture element number zero, is the text matched by the whole regular expression pattern.

(?<name>) Captures and names a match. The name argument is a string, which may not contain any punctuation and cannot begin with a number.

(?:) This match will not produce a group.

(?imnsx-imnsx:) Applies or disables the specified options within the subexpression. For example, **(?i-s:)** turns on case insensitivity and disables single-line mode. For the meaning of the options you can turn on or off, see Table 19.1.

Ignores white space in the pattern and allows comments in a multi-line regular expression. Comments begin with the **#** symbol and everything to the right of this symbol to the end of the line is considered a comment. The following is a simple regular expression with embedded comments:

```
(?x:(?<month>\d{1,2})[/|-]
      # match the month part of the year
      # followed by a backslash or dash
(?<day>\d{1,2})[/|-]
      # match the day part of the year
      # followed by a backslash or dash
(?<year>\d{2,4})
      # match the year part of the year
      # followed by a backslash or dash
)
```

Now that you have seen how to split a match into groups, let's look at captures: how to match multiple instances of the same regular expression in a single step.

Regular Expressions with Multiple Captures

The matches we've seen in the examples so far contain a single capture each: that is, they contain a single instance of the pattern we're searching for. It's possible to capture multiple instances of the same pattern in the text in a single sweep. Let's say you're given a file that contains pairs of keys and values and you want to extract either the pairs, or their parts. Here's the sample text:

```
value1 = 34 value2 = 405 value3 = 4534  
value4 = 45 value5 =3334
```

[Team Fly](#)

 Previous

Next 

```
value10 = -4554
value11 = 3904
value12=456 value13=5564
```

Notice that this is a rather unstructured text document. The only requirement is that it contains key/value pairs. You can have multiple pairs on the same line, separated by at least one space.

The following search pattern will locate all key/value pairs in separate groups, each group containing multiple captures:

```
((\w+)\s*=\s*(\S+)\s+)+
```

The regular expression in the outermost parentheses locates strings like the following:

```
key1 = value1
```

We place this expression in parentheses and then specify the + quantifier to locate any run of key/value pairs in the text. If you click the Find First button, the program will match all the key/value pairs at once. Then the following will appear on the Groups ListBox control:

```
GROUP 1 value1 = 34 value2 = 405 value3 = 4534
value4 = 45 value5 =3334
```

```
value10 = -4554
value11 = 3904
value12=456 value13=5564
```

```
CAPTURE 1 value1 = 34 value2 = 405 value3 = 4534
value4 = 45 value5 =3334
```

```
value10 = -4554
value11 = 3904
value12=456 value13=5564
```

```
GROUP 2 value13=5564
```

```
CAPTURE 1 value1 = 34
CAPTURE 2 value2 = 405
CAPTURE 3 value3 = 4534
CAPTURE 4 value4 = 45
CAPTURE 5 value5 =3334
CAPTURE 6 value10 = -4554
CAPTURE 7 value11 = 3904
CAPTURE 8 value12=456
CAPTURE 9 value13=5564
```

```
GROUP 3 value13
```

```
CAPTURE 1 value1
CAPTURE 2 value2
CAPTURE 3 value3
CAPTURE 4 value4
CAPTURE 5 value5
CAPTURE 6 value10
CAPTURE 7 value11
```

```
CAPTURE 8      value12
CAPTURE 9      value13
GROUP 4        5564
CAPTURE 1      34
CAPTURE 2      405
CAPTURE 3      4534
CAPTURE 4      45
CAPTURE 5      3334
CAPTURE 6      -4554
CAPTURE 7      3904
CAPTURE 8      456
CAPTURE 9      5564
```

The first group contains a single capture, which is the entire matched string. The second group contains the pairs of keys and values, as they appear in the matched text. The last two groups contain the names of the keys and the corresponding values. If you test the pattern shown in this example with a large segment of text, you'll realize that the operation will take a while, but this is the most efficient method of parsing text made up of items with the same structure.

The problem with processing text files with regular expressions is that you won't get any indications about possible errors (errors that you would normally catch from within your code if you wrote a parsing routine in VB). You should make sure that the file has the correct structure before processing it with regular expressions. Because the syntax of regular expressions is so cryptic, you should also test your regular expressions with some simple text to make sure that they locate the desired patterns. Try to include in your sample text patterns that are similar to the ones you want to capture but that don't qualify. After you're certain that the regular expression you have built locates the desired patterns in the sample text, use it with a large segment of text.

Lookahead and Lookbehind Assertions

A lookahead assertion is a tool for specifying not a match, but a pattern that will precede the match. For example, we may be interested in extracting the century from a date's year (the numeric value 17 from 1789, for example). We want to grab the first two digits from a four-digit string. This is a positive lookahead assertion and can be expressed with the following regular expression:

```
\d\d(?=89)
```

This regular expression will match the string "17" in "1789."

There are positive and negative lookaheads. A *positive lookahead* specifies the pattern that must follow the match, and it must appear in a pair of parentheses and be prefixed with a question mark and the equals sign. A *negative lookahead* specifies the pattern that must not follow the match, and it must appear in a pair of parentheses prefixed by a question mark and the exclamation mark.

Lookbehind assertions are equivalent to lookahead assertions, but they specify the pattern that must precede the match. A positive lookbehind specifies the pattern that must precede the match, and it must appear in a pair of parentheses and be prefixed by a question mark, the left angle bracket, and the equals sign. The following regular expression is a positive lookbehind that matches the "34" in 1934 and the "99" in "1999."

```
(?<=19)\d\d
```

[Team Fly](#)

 Previous

Next 

To match the year in a date, use the following regular expression:

```
(?<=\d{1,2}\\/\d{1,2}\\/\d{4}
```

The entire expression is located, but the match that corresponds to the subexpression in parentheses is not part of the match reported by the Matches property, or the Match method. In a text with dates, this regular expression will match the years, as shown next (the matches are in bold):

```
From 1/5/1749 to 12/31/1820
```

Finally, a negative lookbehind specifies the pattern that must not follow the match, and it must appear in a pair of parentheses prefixed by a question mark, the left angle bracket, and the exclamation mark. You will notice that negative lookahead and lookbehind assertions are specified with the same characters as their positive counterparts and an exclamation mark, which reverses the sign of the assertion. Lookahead and lookbehind assertions use the same notation, with the exception of the left angle bracket symbol, which points to direction of the preceding pattern in the case of lookbehind assertions.

What if you want to locate all the years in the 18th century, excluding the century part (the "89" in 1789)? The following regular expression does exactly that: it isolates the last two digits from the year in a date from 1/1/1700 to 12/31/1799:

```
(?<=\d{1,2}[/|-]\d{1,2}[/|-]17)\d\d
```

If you apply the previous regular expression to the sample text shown next, the bold matches will be reported:

```
1/3/1789  
4/8/1779  
19/4/1889  
19/4/17 02
```

You can use a replacement regular expression like the following to change the format of the dates in the text. Enter the following text:

```
Born on 5/8/1915 and died on 16/3/2001
```

and perform a search and replace operation with the last regular expression and the following replacement pattern:

```
${year}-${month}-${day}
```

The original string will be transformed as follows:

```
Born on 1915-5-8 and died on 2001-16-3
```

The regular expression

```
(?i:Click)
```

matches "click," "Click," and "CLICK" in the text, even if the IgnoreCase of the RegEx object has been turned off. The case sensitivity is restored to its original setting for the following searches with the same RegEx object.

[Team Fly](#)

 Previous

Next 

The following expression locates the word "ISBN " followed by an ISBN value (10 digits, or 9 digits and "X"):

```
ISBN \d{9}[\dx]
```

If the current instance of the RegEx object has its IgnoreCase option turned off, you can turn it on for a single search by specifying the **I** option to the regular expression:

```
(?i:ISBN \d{9}[\dx])
```

To turn off the case sensitivity, place the minus symbol in front of the *i* option:

```
(?-i:ISBN \d{9}[\dx])
```

You can also turn on or off any of the *i*, *m*, *n*, *s*, and *x* options in a subexpression. The following regular expression is made up of two subexpressions and it locates substrings like "Grade A" or "GRADE C." The word "grade" can be spelled in lower/uppercase, or any combination of lowercase and uppercase characters. The grade itself, however, is an uppercase character: A, B, C, D, E, F, or I. Here's the regular expression that locates the word "grade" without case-sensitivity, followed by a space and an actual grade value (an uppercase character):

```
(?i:grade) (?-i:[ABCDEFI])
```

You can combine subexpressions in a regular expression to locate exactly the string you're interested in. The following regular expression will locate strings like "Grade B," but it will match only the grade (the character "B"):

```
(?<=(?i:grade)) (?-i:[ABCDEFI])
```

This regular expression will return two matches in the following text (the matches are shown in bold in the text):

```
Grade A grade B Grade c grade c
```

If you want to capture specific groups in the match, add a pair of parentheses around the group(s) you want to capture as usual. To capture the grades, use the following regular expression:

```
(?<=(?i:grade) ) (?-i:([ABCDEFI]))
```

Then you can embed the grades in the original string in a pair of brackets with the following replacement string:

```
[$1]
```

The original string will become:

```
Grade [A] grade [B] Grade c grade c
```

To add a plus sign after each grade, use the following regular expression to match the passing grades:

```
(?<=(?i:grade) ) (?-i:([ABCD]))
```

and the following replacement regular expression:

```
$1+
```

[Team Fly](#)

 Previous

Next 

After the replacement takes place, the original string will become:

```
Grade A+ grade B+ Grade c grade c
```

To exploit lookahead and lookbehind assertions, use the following metacharacters:

(?=) Zero-width positive lookahead assertion. Continues match only if the subexpression matches at this position on the right. For example, `\w+(?=\d)` matches a word followed by a digit, without matching the digit. This construct does not backtrack. The expression

```
\b19(?=\d\d\b)
```

matches the "19" in "Born in 1915 and died in 2001." The following expression matches the day and month part of a date:

```
\d{1,2}\\/\d{1,2}\\/(?=\d{4})
```

(?!) Zero-width negative lookahead assertion. Continues match only if the subexpression does not match at this position on the right. For example,

```
\b(?!un)\w+\b
```

matches words that do not begin with "un".

(?<=) Zero-width positive lookbehind assertion. Continues match only if the subexpression matches at this position on the left. For example,

```
(?<=19)99
```

matches instances of 99 that follow 19. This construct does not backtrack.

(?<!) Zero-width negative lookbehind assertion. Continues match only if the subexpression does not match at the position on the left.

(?>) Nonbacktracking subexpression (also known as a greedy subexpression). The subexpression is fully matched once, and then does not participate piecemeal in backtracking. (That is, the subexpression matches only strings that would be matched by the subexpression alone.)

Advanced Replacement Operations

Now that you've learned how to locate multiple captures with a single regular expression, you probably want to know how to replace all the captures in a single operation. Let's say you want to clean up the preceding list of key/value pairs. We'll use the same search pattern to capture all pairs in a single match. Here's the search pattern:

```
(\w+)\s*=\s*(\S+)\s+
```

and here's the sample text:

```
value1 = 34 value2 = 405 value3 = 4534  
value4 = 45 value5 =3334  
value10 = -4554  
value11 = 3904  
value12=456 value13=5564
```

[Team Fly](#)

 Previous

Next 

This page intentionally left blank.

In the Replace Pattern box you can enter a replacement pattern, which is another regular expression, and then click the Replace Matches button to perform the replacements. The following two declarations appear outside any procedure and are used throughout the project's code:

```
Dim RX As Regex
Dim thisMatch As Match
```

RX is a `Regex` object that's used throughout the code and `thisMatch` is a `Match` object that represents the current match. The Find First Match button's code, which is shown in Listing 19.9, locates the first match by calling the `RX` object's `Match` method.

LISTING 19.9: THE FIND FIRST MATCH BUTTON'S CODE

```
Private Sub FindFirst(ByVal sender As System.Object, _
                    ByVal e As System.EventArgs) _
    Handles btnFindFirst.Click
    Try
        RX = New Regex(TextBox1.Text, _
                      RegexOptions.IgnoreCase Or _
                      RegexOptions.IgnorePatternWhitespace Or
                      RegexOptions.Multiline)
    Catch exc As ArgumentException
        MsgBox(exc.Message)
        Exit Sub
    End Try
    thisMatch = RX.Match(TextBox2.Text)
    If thisMatch.Success Then
        ShowGroups(thisMatch)
        btnFindNext.Enabled = True
    End If
End Sub
```

The code behind the Find Next Match button is similar. This time we call the `NextMatch` method of the `RX` object to retrieve the next match. Listing 19.10 shows the code of the Find Next Match button. The current match is stored in the form level variable `thisMatch`, so that we can use it the next time the `FindNext` button is clicked.

LISTING 19.10: THE FIND NEXT MATCH BUTTON'S CODE

```
Private Sub FindNext(ByVal sender As System.Object, _
                    Handles btnFindNext.Click
    If thisMatch Is Nothing Then
        thisMatch = RX.Match(TextBox2.Text)
        btnFindNext.Enabled = True
    Else
        thisMatch = thisMatch.NextMatch
    End If
End Sub
```

```
        If Not thisMatch.Success Then
            MsgBox(''There are no more matches in the text!')
            btnFindNext.Enabled = False
            Exit Sub
        End If
    End If
    ShowGroups(thisMatch)
End Sub
```

The ShowGroups() subroutine accepts a Match object as argument and displays its groups (if any) on the ListBox control at the bottom of the form, as shown in Listing 19.11.

LISTING 19.11: THE SHOWGROUPS SUBROUTINE

```
Sub ShowGroups(ByVal thisMatch As Match)
    TextBox2.Select(thisMatch.Index, thisMatch.Length)
    TextBox2.ScrollToCaret()
    ListBox1.Items.Clear()
    If thisMatch.Groups.Count > 0 Then
        Dim igrp As Integer
        ' The first group of the match (the one with index = 0)
        ' is the match itself, so we skip it.
        For igrp = 0 To thisMatch.Groups.Count - 1
            ListBox1.Items.Add("GROUP" & igrp.ToString & _
                " " & thisMatch.Groups(igrp).Value)
        Next
    End If
End Sub
```

Finally, the Replace Matches button's code calls the Replace method passing the strings of the three TextBox controls as arguments: the text, the search pattern, and the replacement pattern. Because the replacement pattern itself can be a regular expression, the Replace method is a very flexible and powerful tool. Listing 19.12 shows the code behind the Replace Matches button. This event handler replaces all the matches in the text with the specified replacement string.

LISTING 19.12: THE SHOWGROUPS SUBROUTINE

```
Private Sub Replace(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles btnReplace.Click
    Try
        RX = New Regex(txtPattern.Text)
    Catch exc As Exception
        MsgBox(exc.Message)
    Exit Sub
End Sub
```



FIGURE 19.9 Selecting one of the predefined regular expressions by its description

At the time of this writing, the following regular expressions are displayed on the RegExSamples dialog box:

Locate all HTML tags

```
<[>]+>
```

Locate repeated words

```
\b([A-Za-z]+) \1\b
```

Locate e-mail addresses (1)

```
[A-Za-z0-9]{1,}@[A-Za-z0-9\.\.]{1,}
```

Locate e-mail addresses (2)

```
[\w\.-]+@[\w\.-]+(\.[a-zA-Z]+)
```

Locate URLs in HTML

```
(\w+):\/\/\([^\:]+\):(\d*)?([\# ]*)/
```

Locate all 10-letter words in text

```
\b[A-Za-z]{10}\b
```

Locate dollar amounts

```
\$\d{1,}\.\d{2}
```

Locate uppercase words

```
\b[A-Z]{2,}\b
```

Locate valid postal codes

```
[A-Z]{2}(\ |-)(\d{5})?((\ |-){1}\d{3,5})?
```

Dollar amounts

```
\$[0-9]+(\.[0-9]{0-9})?
```

Dates (mm/dd/yy or mm/dd/yyyy)

```
\b\d{1,2}\/\d{1,2}\/(\d{2}|\d{4})\b
```

Of course, you can edit the project's code and add regular expressions that are more specific to your needs.

To search the text, click the grep button (or press Enter). The matches will be displayed on the RichTextBox control at the bottom of the window. The program isolates the lines that contain the matches and displays them on the control. The matches are highlighted in yellow and underlined, in an attempt to emulate an old Unix terminal look. The numbers in front of the lines are not line numbers; they're the number of the matches found so far. If a line contains two matches, the following number is larger by two than the current line's number.

The matching process is quite fast. If the process returns too many matches, displaying them on the RichTextBox control may take more than a few seconds. You can interrupt the process by pressing the Escape key. This keystroke will interrupt the process of displaying the results, not the search.

The code behind the grep button goes through each selected file and calls the GrepFile() subroutine to process the file, as shown in Listing 19.13.

LISTING 19.13: THE GREP BUTTON'S CODE

```
Private Sub btngrep_Click(ByVal sender As System.Object, _  
                          ByVal e As System.EventArgs) _  
    Handles btngrep.Click  
  
    Dim fname As String  
    Abort = False  
    rtfResults.Text = ""  
    For Each fname In lstFiles.Items  
        GrepFile(fname)  
    Next  
End Sub
```

The GrepFile() subroutine finds all the matches in the file whose path is passed to the subroutine as argument and stores them in a MatchCollection object. The RegEx object's Options argument is set to Multiline and (optionally) to Ignore Case. After retrieving the matches, it calls the Show-Matches() subroutine to display the lines where the matches were found on a RichTextBox control. Here's the code of the GrepFile() subroutine (Listing 19.14).

LISTING 19.14: THE GREPFILE() SUBROUTINE

```
Sub GrepFile(ByVal filename As String)  
    Dim TReader As System.IO.StreamReader  
    TReader = New System.IO.StreamReader(filename)  
    Dim txt As String = TReader.ReadToEnd  
    TReader.Close()  
    Dim RX As Regex  
    Dim Options As RegexOptions  
    If chkCase.Checked Then  
        Options = RegexOptions.IgnoreCase Or RegexOptions.Multiline  
    Else  
        Options = RegexOptions.Multiline  
    End If
```

```
Dim allMatches As MatchCollection
Try
    allMatches = RX.Matches(txt, txtPattern.Text, Options)
Catch exc As Exception
    MsgBox(exc.Message, MsgBoxStyle.OKOnly, _
        "INVALID REGULAR EXPRESSION")
    Exit Sub
End Try
ShowMatches(allMatches, filename, txt)
End Sub
```

The results are displayed on a RichTextBox control, so that the matches can be formatted differently than the rest of the text. The code uses the `textFont`, `ulineFont` and `boldFont` variables to represent the three fonts to be used to render the matching lines on the control, and it changes the font by setting the control's `SelectionFont` property.

The `ShowMatches()` subroutine accepts three arguments: the `MatchCollection` that contains all the matches, the name of the file (the code displays the name of the file that was processed), and the original text. Notice that the text is passed by reference, to avoid making a copy of a large block of text.

The subroutine's code extracts each match's starting location in the original text and its length with the `Index` and `Length` properties of the appropriate `Match` object. Then it prints the text in front of the match on the same line in a regular green font, the match in underlined font and yellow color, and the remaining text on the same line in regular green font.

Listing 19.15 shows the `ShowMatches()` subroutine, which is basically an exercise in programming the RichTextBox control. The statements that deal with the matches are straightforward, but most of the code deals with the proper formatting of the matches.

LISTING 19.15: THE SHOWMATCHES() SUBROUTINE

```
Sub ShowMatches(ByVal allMatches As MatchCollection, _
    ByVal fname As String, ByRef text As String)
    Dim textFont As Font = rtfResults.Font
    Dim boldFont As Font = New Font(rtfResults.Font.Name, _
        rtfResults.Font.Size + 2, FontStyle.Bold)
    Dim ulineFont As Font = New Font(rtfResults.Font.Name, _
        rtfResults.Font.Size, FontStyle.Underline)
    Dim aMatch As Match
    Dim matchStart, matchLength As Integer
    Dim lineStart, lineEnd As Integer, txtLine As String
    rtfResults.SelectionFont = boldFont
    rtfResults.AppendText("FILE: " & Path.GetFileName(fname) & _
        vbCrLf & "Found " & allMatches.Count & _
        " matches" & vbCrLf & vbCrLf)
    rtfResults.SelectionFont = textFont
    Dim currentMatch As Integer
```

```
Me.Cursor = Cursors.WaitCursor
rtfResults.ReadOnly = False
For Each aMatch In allMatches
    currentMatch += 1
    matchStart = aMatch.Index
    matchLength = aMatch.Length
    lineStart = text.LastIndexOf(vbCrLf, matchStart)
    If lineStart < 0 Then lineStart = 0
    lineEnd = text.IndexOf(vbCrLf, matchStart + matchLength)
    If lineEnd < 0 Then lineEnd = text.Length
    If multiple matches are found in the same line, display it once
    If txtLine <> text.Substring(lineStart + 2, _
                                lineEnd - lineStart - 2) Then
        txtLine = text.Substring(lineStart + 2, _
                                lineEnd - lineStart - 2)
        rtfResults.SelectionColor = Color.White
        rtfResults.AppendText('# " & currentMatch.ToString & _
                               ">>" & vbCrLf)
        rtfResults.SelectionColor = Color.LightGreen
    Match again all the matches on the same line
    Dim lineMatches As MatchCollection = _
        Regex.Matches(txtLine, txtPattern.Text)
    Dim lineMatch As Match
    Dim nextLineStart As Integer = 0
    For Each lineMatch In lineMatches
        rtfResults.SelectionFont = textFont
        rtfResults.AppendText(txtLine.Substring(nextLineStart, _
                                                lineMatch.Index - nextLineStart))
        nextLineStart = lineMatch.Index + _
            lineMatch.ToString.Length
        rtfResults.SelectionFont = underlineFont
        rtfResults.SelectionColor = Color.Yellow
        rtfResults.AppendText(lineMatch.ToString)
        rtfResults.SelectionFont = textFont
        rtfResults.SelectionColor = Color.LightGreen
    Next
    rtfResults.AppendText(txtLine.Substring(nextLineStart))
    rtfResults.AppendText(vbCrLf)
End If
If Abort Then
    Abort = False
    rtfResults.SelectionFont = boldFont
    rtfResults.AppendText(vbCrLf & "Interrupted!" & vbCrLf)
    rtfResults.SelectionFont = textFont
    Beep()
```

```
        Me.Cursor = Cursors.Default
        Exit Sub
    End If
    Application.DoEvents()
Next rtfResults.AppendText(vbCrLf)
Me.Cursor = Cursors.Default
rtfResults.ReadOnly = True
End Sub
```

To add formatted text on a RichTextBox control, you must first set the SelectionFont and/or SelectionColor properties of the control to the appropriate values (how you want to format the text) and then append the text to be formatted with the AppendText method. The text you append to the control will be formatted according to the settings of the SelectionFont and SelectionColor properties in effect. To return to the default text formatting, set these two properties to their initial values again. Note also that only the matches are formatted differently. All other characters on the line (the characters before and after the match) have the default format. The font for the matches is stored in the `ulineFont` variable and the regular text's font is stored in the `textFont` variable, which are declared at the beginning of the event handler. The default text color is light green.

Notice that the code doesn't display the same line twice if it contains two, or more, matches. Instead, it highlights all the matches on the line. The number shown in front of the match, however, is the number of the first match in the corresponding line. If a line contains two matches, this number will be increased by two, to indicate that the previous line contains two matches.

The Abort variable is declared on the form's level and is set to True when the user clicks the Escape key. As the code iterates through the matches, it examines the value of the Abort variable and, if it's True, it terminates the processing of the matches. The form's KeyPreview property should be set to True. The following statements in the form's KeyUp event handler detect the Escape keystroke (Listing 19.16).

LISTING 19.16: CANCELING THE MATCHING PROCESS

```
Private Sub RegExForm_KeyUp(ByVal sender As Object, _
    ByVal e As System.Windows.Forms.KeyEventArgs) _
    Handles MyBase.KeyUp
    If e.KeyCode = Keys.Escape Then Abort = True
End Sub
```

The rest of the code that prompts the user to select filenames and add them to the ListBox control, or the code that retrieves one of the predefined patterns, is fairly straightforward and we will not discuss it here.

The `PG Graphics` object represents the surface of the `PictureBox` control on the control. This is where we'll draw the functions. The `PG` object is created with the following statements:

```
Dim Pbmp As Bitmap
Pbmp = New Bitmap(PictureBox1.Width, PictureBox1.Height)
PictureBox1.Image = Pbmp
Dim PG As Graphics
PG = Graphics.FromImage(Pbmp)
```

We draw directly on the bitmaps of the two `Graphics` objects to generate persistent graphics. Drawing a complicated plot with many functions takes a few seconds, as the functions must be evaluated every time, and we'd rather not perform all the calculations every time the form that contains the control needs to be refreshed.

The GraphicsPath Object

The new object we're introducing in this section is the `GraphicsPath` object, which represents an arbitrary curve. A `GraphicsPath` object is a collection of simpler graphics entities, such as lines and arcs, and it's treated as a single entity. It's rendered as a single curve and it can be transformed also as a single entity (the transformations you apply to the `GraphicsPath` object are also applied to its constituent elements). Most importantly, you can set a style for the path, which will be applied correctly to all its elements. For example, you can set the path's style to a dashed line and the `GraphicsPath` object will be drawn with the specified style—you don't have to worry about what happens at the joins of the line segments.

Paths are used to draw curves and outlines of shapes, and to define clipping regions. To create a `GraphicsPath` object, declare a variable of this type and then call its `Add` methods to add elements to the path. There's an `Add` method for each type of graphics primitive you can add to a path (`AddCurve`, `AddLine`, `AddPolygon`, and so on). In our code we'll create a separate path for each function; each path is made up of line segments. The plot is a collection of (X, Y) points: (X0, Y0), (X1, Y1), (X2, Y2), ... (Xn, Yn). Instead of turning on the pixels that correspond to these points, we draw line segments between them. The first line segment connects point (X0, Y0) to point (X1, Y1), the second line segment connects point (X1, Y1) to point (X2, Y2), and so on. The result is a solid curve that goes through all the points. This is the definition of the curve. The curve can be drawn in any style, but the path that represents it has no gaps.

The `GraphicsPath` object is the most convenient method of representing a curve as a collection of points on the drawing surface. The coordinates of the path's elements are expressed in world coordinates, which means that you'll have to transform them into screen coordinates before rendering the path. The transformation of the path is not trivial, either, and we discuss it in the section "Transforming the Plot's Curves," later in this chapter.

The Control's Members

All aspects of the appearance of the plots can be specified through properties, and the PlotControl exposes quite a few properties. The basic properties are those that determine the functions to be plotted and how they'll be plotted. The definitions of the functions must be added to a collection. Since each function is plotted in its own style, the control provides four collections, and you must

[Team Fly](#)

 Previous

Next 

add to each collection a member for each function to be plotted. The four collections, which are public ArrayLists, are the following:

Functions The collection with the functions to be plotted. You can use the members of the Math namespace in defining a function.

FunctionColors The collection with the colors in which each function will be rendered.

FunctionStyles The collection with the styles of the lines, in which each function will be rendered. A function's style must be a member of the DashStyle enumeration: Dash, DashDot, DashDotDot, Dot, and Solid (you can't use the Custom style).

FunctionLineWidths The collection with the widths of the lines, in which each function will be rendered.

To set up a function for plotting with the PlotControl, place an instance of the control on the form and then set its properties with statements like the following:

```
PlotControl.Functions.Add('Cos(x*2)*Sin(x*7)')
PlotControl.FunctionStyles.Add(Drawing2D.DashStyle.Solid)
PlotControl.FunctionColors.Add(Color.Red)
PlotControl.FunctionLineWidths.Add(2)
```

The preceding statements prepare the control to print a function with a solid, 2-pixel-thick line in red color. To plot more than one function, use similar statements for each one. Each function must have a definition, a style, a color, and a line thickness.

Once you've specified the functions to be plotted, you must specify the range of X values in which the plot will be generated. To specify this range, set the control's XMin and XMax properties. The control will calculate the corresponding range of Y values and will plot the functions using its defaults for numbering the axes, drawing the grid, and so on. You may skip the automatic calculation of the Y axis range by setting the properties YMin and YMax. If these two properties are not both zero, their values will be used in determining the Y axis range.

In addition to the basic properties of the functions to be plotted, the PlotControl provides a number of properties that affect the appearance of the plot, specifically:

PlotTitle The plot's title, which is printed above the PictureBox with the plot. The text will not be wrapped, so make sure that the title will fit in the available area (use a smaller font for the title, if you have to).

PlotTitleFont, PlotTitleColor The font and color in which the plot's title will be rendered.

XAxisTitle, YaxisTitle The titles of the two axes. The text will not be wrapped, so you should make sure that the text will fit in the available area (use a smaller font for the axis titles, if you have to).

AxisTitleFont, AxisTitleColor The font and color in which the axis titles will be rendered.

ShowMajorGrid, ShowMinorGrid Two Boolean values that determine whether you want to draw a major and a minor grid.

[Team Fly](#)

 Previous

Next 

MajorXTicks, MajorYTicks The number of major ticks along the X and Y axes. Major ticks are shown on the plot with a short, thick line. Major and minor ticks are displayed independently of the grid.

MinorXTicks, MinorYTicks The number of minor ticks between two major ticks. Minor ticks are shown on the plot with a short, thin line.

MajorGridWidth, MinorGridWidth The width of the major and minor grid lines.

MajorGridColor, MinorGridColor The color in which tick marks and grid will be rendered.

AxisNumberFont, AxisNumberColor The font and color in which the axis numbers will be rendered.

The control provides two methods: the Plot method, which generates the plot according to the settings of the properties, and the Clear method, which clears the control.

The implementation of the control's properties is quite trivial and we don't show the corresponding code in this chapter. Instead, we'll focus on the Plot method, which does all the work.

EVALUATING FUNCTIONS AT RUNTIME

Plotting functions would be fairly trivial if Visual Basic provided a technique for evaluating functions at runtime. To evaluate arbitrary math expressions at runtime, we used the MSScript control, which is an ActiveX control that can be used with .NET applications. This control was distributed with VB6 and it's already installed on your computer if you're an old VB6 developer. If not, you can download it from Microsoft, at www.microsoft.com/scripting. We discussed the capabilities of this control briefly in Chapter 16, where we used it to design a simple calculator.

To use the MSScript control in a .NET project, you must reference this control from within your project and place an instance of it on the form that will use it. Since the PlotControl is a custom control, place an instance of the MSScript control on the design surface of the UserControl object. The following function evaluates a math expression for a given value of the independent variable X:

```
Function FunctionEval(ByVal FunctionName As String, _
                    ByVal X As Single) As Single
    Try
        AxScriptControl1.ExecuteStatement('X=" & X)
        FunctionEval = CSng(AxScriptControl1.Eval(FunctionName))
    Catch exc As Exception
        Throw New Exception("Can't evaluate function at X=" & X)
    End Try
End Function
```

The FunctionEval() function accepts two arguments, the expression to be evaluated (a string) and a value of the independent variable, and returns the value of the expression for the specified value of the independent variable. The FunctionEval() function is called for each value of the x variable at which we want to plot the function.

Our code uses the FunctionEval function to evaluate the function in the range that will be plotted. Before plotting the function, we must determine the minimum and maximum values of the Y axis (see Listing 20.1). This takes place from within the following loop, which keeps track of the minimum and maximum values of all the functions that will be plotted.

[Team Ely](#)

 Previous

Next 

LISTING 20.1: CALCULATING THE Y AXIS RANGE FOR ALL FUNCTIONS

```
For t = _Xmin To _Xmax Step (_Xmax - _Xmin) / (PictureBox1.Width - 2)
    Dim ifunction As Integer
    Dim functionName As String
    For ifunction = 0 To Functions.Count - 1
        functionName = Functions(ifunction)
        Try
            val = CSng(FunctionEval(functionName, t))
            Ymax = Math.Max(val, Ymax)
            Ymin = Math.Min(val, Ymin)
        Catch exc As Exception
            MsgBox(''Can't plot this function in " & _
                "the specified range!" & vbCrLf & exc.Message)
            Exit Sub
        End Try
    Next
Next
```

The values calculated by the code of Listing 20.1 can't be used immediately, because they will not produce proper tick marks. If the calculated range is from -0.2002 to 11.294 , you'd probably use a range from -2 to 12 , or -5 to 20 . The control calculates the "best" range of Y values by calling the `AutoRange()` function. By "best range" we mean a minimum and maximum value that doesn't carry unnecessary fractional digits. It's a range that can be easily divided into equal parts, which correspond to the tick marks along the axes. The code of the `AutoRange()` subroutine is shown in Listing 20.2.

LISTING 20.2: THE AUTORANGE() FUNCTION

```
Private Function AutoRange(ByVal minVal As Double, _
                          ByVal maxVal As Double) As PointF
    Dim P As New PointF
    Dim scale As Long = 1
    Dim Diff As Double = maxVal - minVal
    If Diff > 1 Then
        While Diff > 10
            scale = scale * 10
            Diff = Diff / 10
        End While
        P.X = Convert.ToSingle(Math.Floor(minVal / scale)) * scale
        P.Y = Convert.ToSingle(Math.Ceiling(maxVal / scale)) * scale
        Return P
    Else
        While Diff < 1
            scale = scale * 10
            Diff = Diff * 10
        End While
    End If
End Function
```

```
        P.X = Convert.ToSingle(Math.Floor(minVal * scale)) / scale
        P.Y = Convert.ToSingle(Math.Ceiling(maxVal * scale)) / scale
        Return P
    End If
End Function
```

The `AutoRange` function scales the calculated Y range until it falls in the range from 0 to 10. If the range is larger than 10, it keeps dividing it by 10; otherwise, it keeps multiplying it by 10. Then it takes the largest integer that doesn't exceed the minimum value (this value is given by the `Math.Floor()` function) and the smallest integer that exceeds the maximum value (this value is given by the `Math.Ceiling()` function). These values are then scaled back to produce the automatic range.

Let's consider a function whose Y values are in the range from -30.46 to 24.04 . The difference is $24.04 - (-30.46) = 54.50$. This value must be scaled by $1/10$ to produce a range less than 10. The new range is from -3.046 to 2.404 . The largest integer that doesn't exceed the minimum value is -4 and the smallest integer that exceeds the maximum value is 3 . These values are scaled back by 10 and the new range will become -40 to 30 . This is a "convenient" range that can be easily broken into 6 major ticks at these points: -40 , -26 , -12 , 2 , 16 , and 30 . If you had to break the original range into the same number of sections, each section's magnitude would have to be $54.50/6$, or 9.08333 (the numbers along the vertical axis would be odd, to say the least). The method used by the `AutoRange()` subroutine to generate a good range of values for the Y axis is quite simple, but it works in most cases. You can use a more complicated technique, or provide an interface that will allow users to set the range, should the `AutoRange()` subroutine fail to produce an acceptable range.

The code calls the `AutoRange()` function and then sets the values of the variables that determine the vertical range of the plot (they are the `YMin` and `YMax` properties, and their values are stored in the `_YMin` and `_YMax` local variables). The values returned by the `AutoRange` function are taken into consideration only if the `_YMin` and `_YMax` variables haven't been set. If the application has set the plot's Y range, then the calculated values are ignored. Here are the statements that determine the vertical range of the plot:

```
P = AutoRange(Ymin, Ymax)
If _Ymin = _YMax And _YMax = 0 Then
    Ymin = P.X
    Ymax = P.Y
Else
    Ymin = _Ymin
    Ymax = _YMax
End If
```

TRANSFORMING THE PLOT'S CURVES

As you recall, Visual Basic assumes that the origin is always at the top-left corner of the control. When displaying functions, however, the origin should be at the bottom-left corner. Unless you want to view the plots upside-down, you must switch the minimum and maximum values of the Y coordinates. The transformation that relocates the origin to the bottom-left is performed with the Translate method of the world coordinate system. The Translate method accepts two arguments, which are the

displacement in the horizontal and vertical directions. The horizontal displacement is $-_XMin$: if the X axis starts at -10 , the origin is shifted to the right by 10 units. If it starts at 10, the origin is shifted to the left by 10 units. In either case, the origin of the plot will coincide with the origin of the PictureBox control on which it will be plotted. The vertical translation is the negative of the minimum Y value (the smallest Y value will be mapped to the top of the PictureBox control and the largest Y value will be mapped at the top of the control).

In addition to flipping the plot vertically, we must scale it, so that the world coordinates (the plot's X and Y coordinates) will be mapped to the dimensions of the PictureBox control. The PictureBox control's dimensions are fixed (for each plot), so we simply scale the world coordinates so that their end values coincide with the coordinates of the PictureBox. The following statements set up the necessary transformations:

```
' set up the appropriate Scale and Translate transformations
World = New System.Drawing.Drawing2D.Matrix
World.Scale(((PictureBox1.Width - 2) / (_Xmax - _Xmin)), - _
            (PictureBox1.Height - 2) / (Ymax - Ymin))
World.Translate(-_Xmin, -Ymax)
```

(We subtract two pixels from the dimensions of the PictureBox control to make up for the control's border.)

GDI+ supports the basic graphics transformations (translation, scaling, and rotation) through a transformation matrix. Every point on the drawing surface is specified by two coordinates. To apply one or more transforms, GDI+ creates a 3×3 matrix and multiplies it with the original point. The result is another point, which is the transformed point. You'll never have to create the transformation matrix yourself; you simply create a Matrix object and then specify the transformations you want to apply to all graphics elements. The transformations will be performed in the order in which they were specified, and it's important to get their order correct. The preceding statements specify a scaling transformation and then a translation. The transformations are defined with the Scale and Translate method of a new Matrix object, the `World` matrix, which stores the cumulative transformation.

So far we've specified the necessary transformations, but they will not take effect unless we apply them to the objects to be drawn. Each function's plot is a GraphicsPath object, and you'll see in a moment how this object is created. Before drawing the curve specified by the GraphicsPath object, we must apply the World transformation with the object's Transform method. For each function, we'll create a Path object, then we'll calculate the points along the path, and finally we'll apply the World transformation and we'll draw the path. The following statements outline this process:

```
Dim plot As New System.Drawing.Drawing2D.GraphicsPath
' Add points to the path
' Apply transformation
plot.Transform(World)
' And finally draw the path
PG.DrawPath(plotPen, plot)
```

CALCULATING A FUNCTION'S PATH

Let's look now at the code that generates the plots. To plot a function, we must calculate the value of the function for each point along the X axis. To avoid calculating the value of the function at more points than necessary, the program figures out at the very beginning how many pixels are in the PictureBox. In our code we set up a loop that starts at `_XMin` and ends at `_XMax`. The loop's counter is given by the following expression:

```
(_Xmax - _Xmin) / (PictureBox1.Width - 2)
```

This increment makes sure that the function is calculated at the points that correspond to the pixels across the PictureBox control. We calculate the value of the function at each point along the horizontal axis and add a line segment to the path that represents the specific function. The line segment extends from the previous point to the current one. This process is repeated for all functions, and we create a new GraphicsPath object for each function. The GraphicsPath objects are added to an ArrayList collection, the `Plots` collection. We'll use this collection to draw the plots on the PictureBox control. Listing 20.3 shows the statements that create the paths of the functions.

LISTING 20.3: GENERATING THE FUNCTIONS' PATHS

```
Dim plot As System.Drawing.Drawing2D.GraphicsPath
For i = 0 To Functions.Count - 1
    Dim functionName As String
    plot = New System.Drawing.Drawing2D.GraphicsPath
    functionName = Functions(i)
    oldX = _Xmin
    oldY = CSng(FunctionEval(functionName, _Xmin))
    For t = _Xmin To _Xmax Step _
        (_Xmax - _Xmin) / (PictureBox1.Width - 2)
        X = CSng(t)
        Try
            Y = CSng(FunctionEval(functionName, t))
            plot.AddLine(oldX, oldY, X, Y)
            oldX = X
            oldY = Y
        Catch
        End Try
    Next
    plots.Add(plot)
Next
```

To display the plots on the control, we must apply the World transformation matrix to each GraphicsPath object and then render it on the control. This is what the statements of Listing 20.4 do.

LISTING 20.4: DRAWING THE FUNCTIONS

```
Dim plotPen As Pen = New Pen(Color.Red)
Dim iplot As Integer
G.SmoothingMode = SmoothingMode
Try
    For iplot = 0 To plots.Count - 1
        plotPen.DashStyle = FunctionStyles(iplot)
        plotPen.Width = Convert.ToInt32(FunctionLineWidths(iplot))
        plotPen.Color = FunctionColors(iplot)
        plots(iplot).Transform(World)
        G.DrawPath(plotPen, plots(iplot))
    Next
Catch ex As Exception
    Throw ex
End Try
```

Drawing the Grid

Before plotting the functions, the program draws the major and minor grids on the PictureBox control. The settings of the two grids are specified by the developer using the appropriate properties of the control. The two nested loops shown in Listing 20.5 draw the two grids. Regardless of whether the grid lines will be visible or not, the tick marks are always visible. The grid's lines are drawn at the same locations as the tick marks, but they cover the entire width or height of the PictureBox control.

LISTING 20.5: DRAWING THE GRID AND TICK MARKS

```
For iGridLine = 0 To _MajorXTicks
    axisNum = _Xmin + iGridLine * (_Xmax - _Xmin) / _MajorXTicks
    If _ShowMajorGrid Then
        If iGridLine <> 0 And iGridLine <> _MajorXTicks + 1 Then
            G.DrawLine(New Pen(_MajorGridColor, _MajorGridWidth), _
                Convert.ToSingle(MajorXGridSpace * iGridLine), _
                Convert.ToSingle(0.0), _
                Convert.ToSingle(MajorXGridSpace * iGridLine), _
                Convert.ToSingle(PictureBox1.Height))
        End If
        If _ShowMinorGrid Then
            Dim iMinorGridLine As Integer
            For iMinorGridLine = 1 To _MinorXTicks
                G.DrawLine(New Pen(_MinorGridColor, _MinorGridWidth), _
                    Convert.ToSingle(MajorXGridSpace * iGridLine + _
                        MinorXGridSpace * iMinorGridLine), _
                    Convert.ToSingle(0.0), _
                    Convert.ToSingle(MajorXGridSpace * iGridLine + _
                        MinorXGridSpace * iMinorGridLine), _
```

```
Convert.ToSingle(PictureBox1.Height))
Next
Else
    Dim iMinorGridLine As Integer
    For iMinorGridLine = 1 To _MinorXTicks
        G.DrawLine(New Pen(_MinorGridColor, _MinorGridWidth),
            Convert.ToSingle(MajorXGridSpace * iGridLine + _
                MinorXGridSpace * iMinorGridLine), _
            Convert.ToSingle(PictureBox1.Height), _
            Convert.ToSingle(MajorXGridSpace * iGridLine + _
                MinorXGridSpace * iMinorGridLine), _
            Convert.ToSingle(PictureBox1.Height - 10))
    Next
End If
Next
```

NUMBERING THE AXES

Next, the numbers at the major tick marks are drawn. To avoid printing very small or very large values, the code scales the values and uses exponentials, if needed. The value 0.00000123 will be printed as $1.23 * 10^{-6}$, and the value 123000000 will be printed as $12.3 * 10^7$. The following statements determine the values along the Y axis, as well as their format:

```
valStep = (Ymax - Ymin) / _MajorYTicks numFormat = "'0.00"
Dim exp As Integer = 0
If valStep < 0.0001 Then
    While valStep < 1
        exp = exp - 1
        valStep *= 10
    End While
End If
If valStep > 99999 Then
    While valStep > 100
        exp = exp + 1
        valStep /= 10
    End While
End If
If Math.Abs(valStep) > 1 Then numFormat = "0.00"
If Math.Abs(valStep) > 10 Then numFormat = "#,###,##0"
If Math.Abs(valStep) < 0.1 Then numFormat = "0.000"
If Math.Abs(valStep) < 0.01 Then numFormat = "0.0000"
If Math.Abs(valStep) < 0.001 Then numFormat = "0.00000"
If Math.Abs(valStep) < 0.0001 Then numFormat = "0.000000"
```

The following statements (Listing 20.6) print the numbers along the Y axis (these statements appear in the same `For...Next` loop that draws the tick marks along the Y axis):

LISTING 20.6: PRINTING THE AXIS NUMBERS

```
G.DrawString(axisNum.ToString(numFormat), AxisNumberFont, _
    New SolidBrush(_AxisNumberColor), _
    New RectangleF(1, PictureBox1.Top + _
    PictureBox1.Height - _
    Convert.ToSingle(MajorYGridSpace * iGridLine) -
    AxisNumberFont.GetHeight(G) / Convert.ToSingle(2), _
    PictureBox1.Left - 5, AxisNumberFont.GetHeight(G)), fmt)
Next
If exp <> 0 Then
    Dim expVal As String
    expVal = "'x10"
    Dim ReducedFont As New Font(AxisNumberFont.Name, _
        AxisNumberFont.Size - 1, AxisNumberFont.Style)
    Dim expWidth As Integer
    expWidth = Convert.ToInt32(G.MeasureString(expVal, _
        ReducedFont, 999).Width) + 1
    fmt.Alignment = StringAlignment.Near
    G.DrawString(expVal.ToString(), ReducedFont, _
        New SolidBrush(AxisNumberColor), _
        New RectangleF(PictureBox1.Left, _
        PictureBox1.Top - 1.2 * ReducedFont.GetHeight(G)
        expWidth, AxisNumberFont.GetHeight(G)), fmt)
    ReducedFont = New Font(AxisNumberFont.Name, _
        AxisNumberFont.Size - 2, AxisNumberFont.Style)
    G.DrawString(exp.ToString(), ReducedFont, _
        New SolidBrush(AxisNumberColor), _
        New RectangleF(PictureBox1.Left + expWidth, _
        PictureBox1.Top - 1.2 * AxisNumberFont.GetHeight(G),
        expWidth, ReducedFont.GetHeight(G)), fmt)
End If
```

The program prints the numbers on the UserControl object's surface, centered at the corresponding tick marks. Note that the tick marks are drawn on the PictureBox control. The X axis numbers are centered horizontally and the Y axis numbers are centered vertically. In drawing the vertical axis numbers, the program takes into consideration the value of the `exp` variable, which is the exponent that will be used for all numbers along the axis. If its value is different than 0, the code sets up a new Font object, the `ReducedFont` object. This font is identical to the font we use for the axis numbers, but smaller by one point. We use this font to print the exponent above the top left corner of the PictureBox control. The X axis numbers are printed with a similar set of statements.

DRAWING THE TITLES

The last step is the drawing of the plot's titles. Each title is printed in an area of fixed size, which is determined by the placement of the PictureBox control on the UserControl object. The font size used for each axis should be such that the title will fit in the designated area. The plot's title is centered over the width of the UserControl object. The X axis title is centered over the width of the PictureBox control and the Y axis title is rotated 90 degrees counter-clockwise and centered over the height of the PictureBox control.

The code that prints the titles is shown in Listing 20.7. First it prints the Y axis title. It applies a rotation and a translation transformation to the G Graphics object. The rotation is needed to rotate the text. The rotation keeps the top left corner of the object to be rotated fixed, so the title will end up outside the area occupied by the UserControl object. To rectify the displacement introduced by the rotation, we must apply another translation transformation by the following amount:

```
-PictureBox1.Height - PictureBox1.Top
```

After printing the vertical title, the transformation is reset—we don't need any transformation to print the two horizontal titles. Listing 20.7 shows the code that prints the plot's titles on the surface of the UserControl object.

LISTING 20.7: DRAWING THE PLOT'S TITLES

```
fmt.Alignment = StringAlignment.Center

G.RotateTransform(-90)
G.TranslateTransform(-PictureBox1.Height - PictureBox1.Top, 0)

G.DrawString(_YAxisTitle, _AxisTitleFont, _
    New SolidBrush(_AxisTitleColor), _
    New RectangleF(0, 0, PictureBox1.Height, _
        _AxisTitleFont.GetHeight(G)), fmt)
G.ResetTransform()
G.DrawString(_XAxisTitle, _AxisTitleFont, _
    New SolidBrush(_AxisTitleColor), _
    New RectangleF(PictureBox1.Left, _
        PictureBox1.Top + PictureBox1.Height + _
        AxisNumberFont.GetHeight(G) * 1.5, PictureBox1.Width,
        _AxisTitleFont.GetHeight(G)), fmt)
G.DrawString(_PlotTitle, _PlotTitleFont, _
    New SolidBrush(_PlotTitleColor), _
    New RectangleF(Me.Left, 5, Me.Width, _
        _PlotTitleFont.GetHeight(G)), fmt)
```

Open the PlotControl project and build the custom control, then use it with your own projects. There are certainly areas to be improved, but it's a good starting point for a control that plots two-dimensional functions. You can easily add statements to plot data as well: open a file with X/Y values and plot them on the control with different markers, such as squares, circles, and so on.

[Team Fly](#)

 Previous

Next 

One virtue of plotting, of translating a mathematical expression or equation into a visual analog, is that you can see relationships you might not notice when merely reading the numbers. At a glance you often can see a larger, overall view of the entire relationship or structure that for most of us is not so clear in an equation.

However, not all numbers used by mathematicians are those familiar normal (or "real") numbers that we use every day, such as 120 miles or 3 hours. For example, to translate waveforms into audio frequencies, or to analyze an FM radio signal, or, indeed, to generate fractals, it is necessary to use "imaginary" numbers.

REAL AND IMAGINARY NUMBERS

In the 16th century, mathematicians began trying to describe a pretty odd duck, the square root of minus one. When you multiply a number by itself, you square it; $4 * 4$ is 16. And you can also describe this relationship between 4 and 16 the other way: 4 is the square root of 16. It seems obvious that there is a square root for any number. With 4, the square root is 2, and so on. Squares and roots had been used for thousands of years, but finally attention was focused on -1 , which, though paradoxical in some ways, eventually proved quite useful.

What is the square root of -1 ? It's unusual, to say the least. For one thing, it's neither positive nor negative. When you square a positive number, you always get another positive number (a positive multiplied by a positive yields a positive). But when you multiply a negative number by a negative number, you also get a positive. So the thing that you could multiply by itself to get -1 is neither positive nor negative (and of course not zero). In other words, it's not a real number.

Descartes called the square root of minus one an "imaginary" number, and the name stuck. This distinguishes it from "real" numbers such as 2, or the square root of 25, or -12 , or $.55$, which all seem somehow more natural. After all, we can find things around us of which there are two or five, or somebody removed a dozen bagels from the kitchen, or an oil can is half full. We experience these "real" numbers. But where have you ever seen a number that is simultaneously neither positive nor negative?

The square root of -1 is the imaginary unit and is denoted as i (sometimes j). With the introduction of the imaginary unit, it was possible to calculate the square root of any negative number. What's the square root of -4 ? If you attempt to calculate it with Visual Basic as:

```
Console.WriteLine(Math.Sqrt(-4))
```

you'll get a runtime exception. Visual Basic can't calculate square roots of negative numbers because it can handle only real numbers.

Let's start by writing -4 as $-1 * 4$. The square root of this quantity is the square root of -1 (which is the imaginary unit i) times the square root of 4 (which is 2). The square root of -4 , therefore, is $2 * i$, where i is the imaginary unit. We can easily verify the previous result because the square of $2 * i$ must be -4 . Let's calculate the quantity:

$$(2*i) * (2*i) = 4*i*i$$

We know that i is the square root of -1 , and therefore the quantity $i * i$ is -1 . Thus the term $4 * i * i$ is -4 . The imaginary number $2 * i$ is indeed the square root of -4 .

We have seen that there are two kinds of numbers: real ones (such as the speed of a car or the distance to the moon) and imaginary ones (such as the square root of a negative number). It is possible

to combine real and imaginary numbers to produce complex numbers. When you combine real with imaginary numbers, you get points on a grid, as shown in Figure 20.2. Real numbers on this grid describe the horizontal distances, and imaginary numbers describe the vertical distances. The resulting complex numbers are denoted as $4 + i * 7$ or $6 + i * 5$.

***NOTE** This isn't really an addition. The plus sign says that the real and imaginary numbers are tied together to form a complex number. It's like describing a point on a city map by using two coordinates, such as A-13.*

Sometimes we don't even use the symbol of addition to indicate a complex number. Instead, we can write the previous complex numbers as $(4, i * 7)$ or $(6, i * 5)$. You can even drop the i , if you keep in mind that the first number is the real one, and the second number is the imaginary one. One last remark about real and imaginary numbers. We mentioned that there are two kinds of numbers—the familiar real numbers and the imaginary ones. Both, however, are complex numbers. A complex number without an imaginary part is a real number. Similarly, a complex number without a real part is an imaginary number. The real numbers we are all familiar with are therefore a special case of complex numbers.

The two components of a complex number are quite distinct. The number $(4 + i * 7)$ is a complex number and not an addition. The component 4 corresponds to a point on the horizontal axis, and the number 7 corresponds to the vertical axis. It doesn't make any more sense to add the two components of a complex number than it does to add time and distance. The real part of a complex number is the letter in the map metaphor (A), and the imaginary part is the number (13).

It is possible, however, to manipulate complex numbers. For example, you can add two complex numbers by adding their real and imaginary parts separately. At the end of the chapter we'll describe how to add or multiply two complex numbers. If you aren't familiar with complex numbers, just follow the general description of the algorithms and worry later about the details—or don't worry at all, and just use the functions we provide.

What's useful and special about using complex numbers to describe points on the complex plane is that new and important real, experimentally verifiable results can be calculated. Complex numbers are now widely used in every field of mathematics and applied science. They are used in voice processing, image compression, radar, earthquake analysis, and many other applications. Fractals, too, live on the complex plane.

MATH TRANSFORMATIONS

A fractal is the graphical representation of an iterative process (like an ordinary `For . . . Next` loop), which is based on multiplying a complex number by itself. This process is called a transformation because each time we multiply a number by itself, we transform it to another number.

Three things can happen when you repeatedly multiply a number by itself. If it is a real number bigger than 1, it will grow toward an infinitely large number ($2 * 2$ becomes 4; then 4

* 4 becomes 16, $16 * 16$ becomes 256, and so on quickly off to infinity). If our starting number is less than one (a fraction), it will shrink toward an infinitely small number close to zero. For example, multiply $1/2$ by $1/2$ and you get $1/4$, then $1/4$ by $1/4$ becomes $1/16$, and so on down to a very little piece of a pie. However, there is a third possible result. On the fence between the numbers that grow huge and the fractions that shrink is the number one. The number one, and the number one alone, is completely stable and cannot balloon or shrink, cannot move in either direction toward infinity.

Applying a transformation that's based on multiplication of real numbers isn't very interesting, because the result is known a priori. As you can see, numbers even extremely close to one will rapidly diverge toward zero or infinity, depending on which side of one they are. This divergence is the basis for drawing fractal images. You have heard the word *attractors*. For any given transformation, attractors are points on the complex plane that draw other points toward them. Zero and infinity are the attractors in our example above—they "attract" the initial numbers toward themselves like magnets. (The value one is not an attractor, but a "stable" point of the transformation. Nothing changes when we multiply one by itself.)

All fractals are built upon the competing magnetism of two attractors. And between these two attractors there is always a border (in our previous example, this border is the number one). If you start an iteration with the number 1, the result will always be one, since one is the only number that, when squared (or otherwise multiplied by itself), remains unaffected, remains one. However, a fractal such as the Mandelbrot Set is a more complicated border—it is a visual depiction of an iteration involving imaginary and complex numbers, not the number one.

COMPLEX NUMBERS PRODUCE EXCEPTIONALLY RICH SHAPES

Our example above is a simple transformation. We merely squared a number, then squared the result, again and again. But watch what happens when we mix in a complex number. Let's write an iteration that involves squaring, but adds a complex number into the mix. This will not be a radical change. (We will put in a complex number, but we'll use zero for the imaginary component when we start the iterations. Therefore, in this case, the imaginary component will have no effect on the outcome). We're going to use this rather "mild" complex number, with its imaginary part neutered, because we want to approach these ideas one step at a time. Neutering the imaginary number provides us with a simpler, more easily understood example. So, say our starting point is $z = (0, 0)$ and the constant $c = (1, 0)$. (From now on we'll use the notation z , rather than a , to represent the points we generate to create our plots. z is just a convention to remind us that we're no longer using real numbers, but instead are now using complex numbers. Also, the constant c , like a constant in VB, remains unchanged—a static value throughout all the iterations.)

If we repeat the operation $z = z^2 + c$ a few times, we will get the numbers:

$$\begin{aligned}0^2+1 &= 1 \\1^2+1 &= 2 \\2^2+1 &= 5 \\5^2+1 &= 26\end{aligned}$$

As you can see, the results get larger and larger, as before, and in a small number of iterations they escape to infinity, also as before. However, because of the additional constant, the pattern (2...5...26) is becoming a slightly less ordinary sequence than our simpler example, 2...4...16. The constant is slightly disturbing the simple squaring as the transformation heads away from 1.

Let's now go a step further and produce a more intricate result. If we change the constant c from 1 to -1 , we see a very interesting effect: rather than fly off the top of the grid to infinity or sink down to zero, the line flutters between zero and minus one. This transformation results in an oscillation. If $c = -1$, the results are

$$\begin{aligned}(-1)^2 - 1 &= 0 \\ 0 - 1 &= -1 \\ (-1)^2 - 1 &= 0\end{aligned}$$

...and so on.

[Team Fly](#)

 Previous

Next 

This elementary transformation offers another possibility. Some numbers will neither inflate to infinity nor shrink to zero. They oscillate between two values. Let's follow the same transformation with $c = (0 + i * 1)$:

$$\begin{aligned}0^2 + (0 + i) &= i \\i^2 + (0 + i) &= (-1 + i) \\(-1 + i)^2 + (0 + i) &= i \\i^2 + (0 + i) &= (-1 + i)\end{aligned}$$

This is another example of a complex number that oscillates, this time between the values i and $(-1 + i)$. Don't be alarmed if you aren't familiar with complex numbers and can't follow these numeric examples. The important thing to keep in mind is that some numbers will escape to infinity, while others will remain bounded. And this distinction gives rise to the most magnificent computergenerated images.

Finally, what if $c = (0, 0)$ and $z = (0.5, 0)$? If you apply the same transformation, z (which represents our current position on the grid) gets smaller and smaller and practically vanishes after a number of iterations. This transformation has the same two attractors, but the boundary between them is no longer a simple circle (the number one). Figures 20.2 and 20.3 show the trajectories of two points on the complex plane—as you can see, we're not dealing with simple, stable, familiar geometric shapes here.

At the heart of every fractal image is a very simple transformation—a trivial transformation, such as squaring a number over and over again. If this transformation is disturbed with the introduction of an imaginary number, there arises the beautiful and elaborate filigree that we all recognize as fractal images. The word fractal comes from the Latin root that also produced fracture and fraction.

Notice that now our results are not that simple horizontal line between zero and infinity that we got before. z is a complex number, so the boundary between the two attractors (zero and infinity) is no longer a single number like one. It's a shape of indescribable complexity and beauty, as we will shortly see.



FIGURE 20.2 This is the trajectory of a point that's in the process of flying off into infinity. This point, an "escapee," will be displayed as a dot outside the boundary of the Mandelbrot Set.

The Mandelbrot Set

The Mandelbrot Set is the group of numbers that, when plugged into the formula $z = z^2 + c$, remain "bounded" (don't escape to infinity). Technically, this set is the portion of the complex plane that is attracted to zero. A number will either fly off toward infinity (and therefore not be part of the Mandelbrot Set) or it will go toward zero (or, possibly, oscillate). The numbers going toward zero or oscillating, when all painted in, create the classic Mandelbrot shape as shown in Figure 20.4.

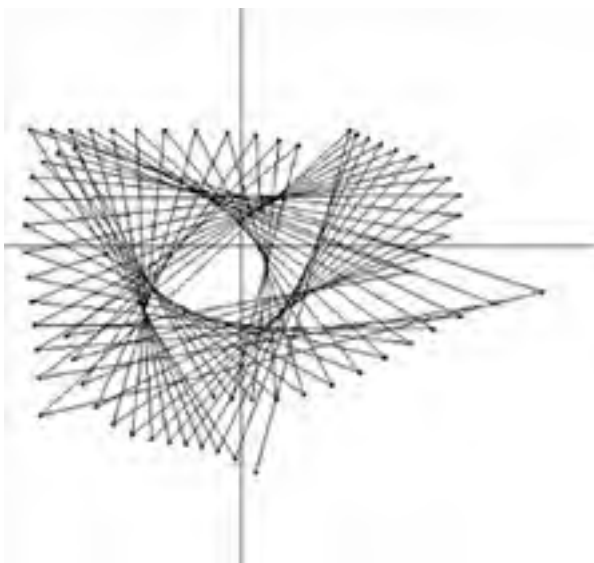


FIGURE 20.3 This trajectory reveals that this point will remain a "prisoner." It cannot puncture the boundaries and will eventually fall to zero. Notice how it approaches zero in an aperture-like, symmetric path.

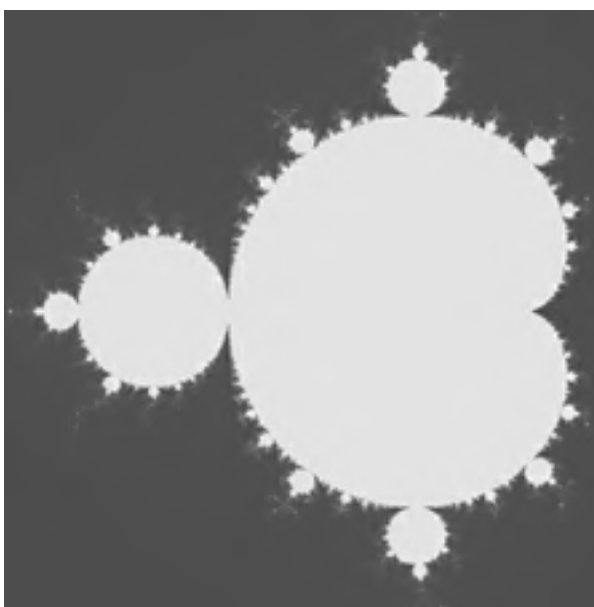


FIGURE 20.4 The classic Mandelbrot Set is revealed here as a somewhat lumpy symmetry.

[Team Fly](#)

 Previous

Next 

To display our fractals on the computer screen, we can think of the complex plane as a grid, with real numbers shown horizontally and imaginary numbers shown vertically. Every pixel on our drawing surface is mapped to a point on the complex plane. To calculate the Mandelbrot Set in VB, we'll first arbitrarily choose a zone within the grid to paint in. For each point in this area, we assign the corresponding complex number to c (corresponding to this particular position on the grid) and, starting with $z = 0$, we repeat the transformation against this value enough times to discover whether or not this particular position will escape to infinity. If it will, we color it white to show that it is outside of the Mandelbrot Set. If it won't escape, we color it black to show that it is part of the set. In other words, we calculate the square of z , add the constant c , and, once again, put the result into z . This process is repeated again and again until we are reasonably confident that we know whether this one is an escapee or not. For a black-and-white fractal, 32 times is enough; for highly detailed fractals, we may check a pixel thousands of times before we're reasonably sure that it remains bounded.

The result is an intricate shape known as the Mandelbrot Set. The irregular "line" between the black and white areas is the boundary between the two attractors (zero and infinity). Points that remain bounded within a fractal set's boundaries are called prisoners, while those that escape to infinity are called escapees.

How do we find out if one of our points succeeds in escaping to infinity? Obviously, we shouldn't have to wait for an overflow condition. It can be mathematically proven that if either the real or the imaginary part of z is larger than 2 or smaller than -2 , it will inevitably escape to infinity. However, if the result remains in the range $(-2,2)$ after a large number of iterations, then the point is bounded. So, we could just run it through the iterations enough times to satisfy ourselves that it is or isn't escaping by going beyond 2 in either direction. However, we're going to use a slightly different approach.

Another way to test for escapees is to compare the result of each iteration against a very large number, such as 100,000. If our result (either its real or its imaginary part) is larger than 100,000, we know that this point escapes to infinity. As we'll soon see, we want to know more about each point than simply whether or not it escapes—so we'll use this compare-to-huge-number technique.

One last detail before we get to the actual VB programming that produces fractals. We mentioned that the operation must be repeated for every point in our grid. Because a computer screen has a limited number of pixels, we need to calculate the values of only those points we can plot on the screen (in other words, the pixels, the dots that are visible to the user). If the area of our grid is from -1 to $+1$ in both axes, and the Picture Box we will use to display the Mandelbrot Set has a resolution of 512×512 pixels, we must find only the points of the grid that correspond to pixels. The increment between two successive points must be $2/511$, and therefore the points we will use are -1 , $-1 + (2/511)$, $-1 + 2 * (2/511)$, and so on, up to $-1 + 511 * (2/511)$. The last point is 1, the other endpoint.

PROGRAMMING THE MANDELBROT SET

Now for the VB programming that opens the door to what many people find a hypnotic and even addictive world. Because fractals are the visual analog of the most complex object in all mathematics, we should be grateful that we amateurs and even nonmathematicians are privileged to see with our computers what even the math geniuses who discovered fractals couldn't visualize.

[Team Fly](#)

 Previous

Next 

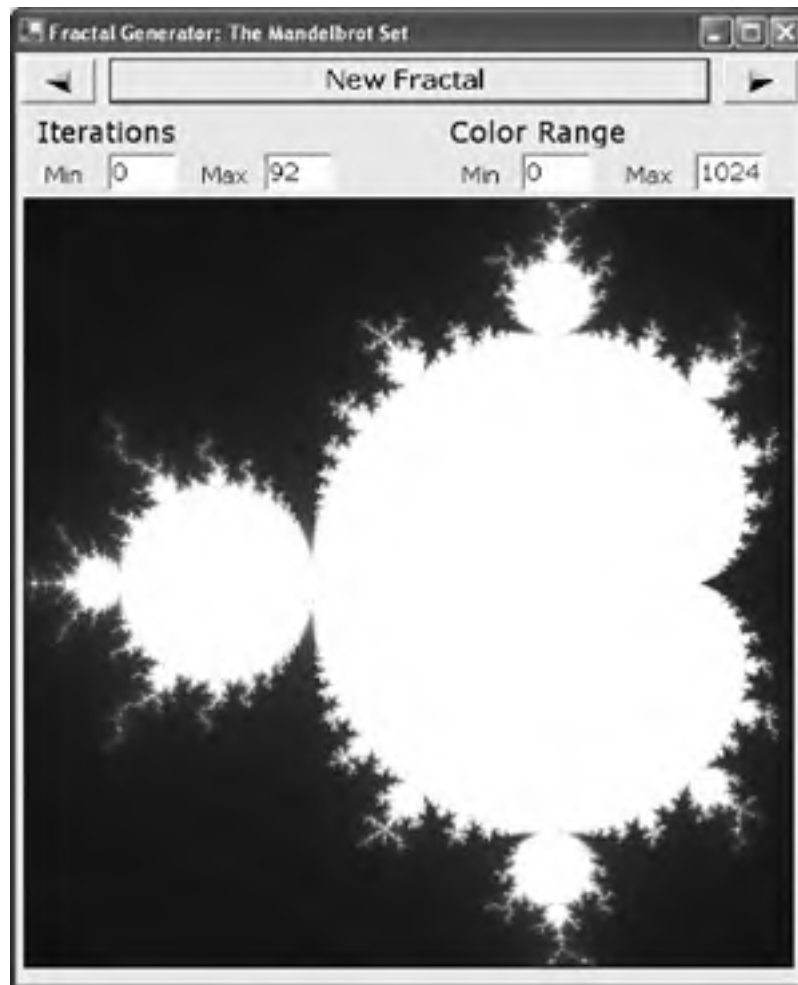


FIGURE 20.5 The Mandelbrot application's interface

The main Form of our first fractal application, Mandelbrot, is shown in Figure 20.5. This application generates colored Mandelbrot Sets. Another type of fractal is the Julia Set, and we'll get to it shortly. The PictureBox control that covers most of the form has dimensions (512×512 pixels).

To draw the Mandelbrot Set, set the parameters on the form and click the New Fractal button. A new fractal will slowly be drawn on your screen. You can stop the calculation (and drawing) of the fractal at any point by pressing the Escape button. After the fractal has been drawn on the screen, you can select an area with the mouse and zoom into it. Just click the New Fractal button again to draw the selected area.

Let's start with the black-and-white Mandelbrot Set. The only parameters to define are the following:

- The area of the complex plane we wish to map (x_{min} , x_{max} , y_{min} , y_{max})
- The resolution of the Picture Box (n_x , n_y)
- The maximum number of iterations required to establish that a given point remains

bounded (`maxiter`)

Thirty-two iterations is a good starting point for the black-and-white Mandelbrot Set. Larger values, such as 500 iterations, yield more accurate sets but, of course, require longer calculation times. So let's plunge in. Listing 20.8 is the programming that results in Figure 20.5.

[Team Fly](#)

 Previous

Next 

SOAP serialization produces a SOAP-compliant envelope that describes its contents and serializes the objects in SOAP-compliant format. SOAP serialized data are suitable for transmission to any system that understands SOAP, and it's implemented by the `SOAPFormatter` class. As you will see later in this book, binary and SOAP serialization are used by remoting to pass objects between two domains (two applications running on the same, or different, computers). SOAP serialization is fire-wall-friendly and is used to remote objects to a server on a different domain. Binary serialization isn't firewall-friendly, but it's faster and more compact. It's used to remote objects in a local area network.

XML serialization is implemented by the `XmlSerializer` class and is a different type of serialization. The `XmlSerializer` class serializes public, read/write properties only. Because it doesn't serialize the private members, or read-only properties, the `XmlSerializer` doesn't quite preserve the state of the object. Another limitation of the `XmlSerializer` is that it doesn't serialize collections, with the exception of arrays and `ArrayLists`. However, it can be customized with the use of attributes (special keywords that prefix the members of a class) and it's as close as we can get to a universal data exchange format. You will see later in this chapter how to retrieve XML from SQL Server and use it to automatically populate objects with a structure that matches the schema of the XML document.

Basic Serialization

To serialize an object, you need to set up a formatter object that determines the type of serialization and a stream object that will accept the result of the serialization. Any stream will do; in this chapter's examples we'll use a stream that represents a disk file. If you only want to look at the serialized stream's bytes, you can use a `MemoryStream`. You can even use a `WebResponse` stream to direct the data to a browser. The formatter is an instance of the `BinaryFormatter` or the `SoapFormatter` class, depending on the type of serialization you want to perform. Both the `BinaryFormatter` and the `SoapFormatter` expose the same methods and the code is identical. The choice of formatter depends on your application's requirements.

***TIP** Not all objects can be serialized in SOAP or binary format. An object can be serialized if its parent class was marked as serializable. Many of the built-in classes are serializable, but not all of them.*

The object to be serialized is usually an instance of a custom class, or a collection of objects. To demonstrate the basic steps in serialization, we'll create an `ArrayList` and populate it with a few serializable objects. The `Rectangle` class is serializable and so is the `Bitmap` class. These are two drastically different types and, as you will see, the serialization mechanism of .NET can handle them both. The `ArrayList` objects is the object to be serialized. We'll serialize it in binary format and store the serialized object into a file, which we'll use later to reconstruct the original `ArrayList`.

First, you must add a reference to the `System.Runtime.Serialization` class to your project. To minimize the typing import the following namespaces to the project:

```
Imports System.Runtime.Serialization.Formatters.Binary
Imports System.IO
```

Then enter the statements of Listing 3.1 in a button's Click event handler. You can find the code samples of this section in the BasicSerialization project, whose main form is shown in Figure 3.1.

[Team Fly](#)

 Previous

Next 

LISTING 20.8: DRAWING THE MAMNDELBORT SET

```
Private Sub btnMandelbrot_Click(ByVal sender As System.Object, _
                                ByVal e As System.EventArgs) _
    Handles btnMandelbrot.Click
    Static IterMin As Integer = 100000
    Dim IterMax As Integer = -100000
    Dim bmap As Bitmap
    bmap = New Bitmap(512, 512, _
        Drawing.Imaging.PixelFormat.Format32bppPArgb)
    PictureBox1.Image = bmap
    Dim pX, pY As Integer
    Dim NX As Integer = 512
    Dim NY As Integer = 512
    Dim MaxIter As Integer
    Dim X, Y As Double
    Dim Iterations As Integer
    Dim minIterations As Integer = CInt(txtMin.Text)
    Dim maxIterations As Integer = CInt(txtMax.Text)
    Dim colorMin As Integer
    Dim colorMax As Integer
    Dim pixelColor As Color

    If IsNumeric(txtColorMin.Text) Then colorMin = _
        CInt(txtColorMin.Text)
    If IsNumeric(txtColorMax.Text) = _
        CInt(txtColorMax.Text)
    If IsNumeric(txtMin.Text) Then minIterations = CInt(txtMin.Text)
    If IsNumeric(txtMax.Text) Then maxIterations = CInt(txtMax.Text)
    If maxIterations <= minIterations Then
        MsgBox("The number of maximum iterations should be larger "
            & "than the number of minimum iterations")
        Exit Sub
    End If
    If colorMax <= colorMin Then
        MsgBox("The first color s value should be smaller " & _
            "than the last color s value")
        Exit Sub
    End If
    For pY = 0 To NY - 1
        Y = YMin + pY * (YMax - YMin) / (NY - 1)
        For pX = 0 To NX - 1
            X = (XMin + pX * (XMax - XMin) / (NY - 1))
            Iterations = Mandelbrot(X, Y, maxIterations)
            Dim clr As Integer
            clr = CInt(colorMin + _
                (colorMax - colorMin) / (maxIterations - minIterations) *
                (Iterations - minIterations))
```

```
        pixelColor = PaintPixel(clr)
        bmap.SetPixel(pX, pY, pixelColor)
    Next
    PictureBox1.Invalidate()
    Application.DoEvents()
    If breaknow Then
        breaknow = False
        Exit Sub
    End If
Next
End Sub
```

Above, X and Y are the grid coordinates of each pixel on the complex plane. In other words, they are the grid coordinates that correspond to the pixels of our display. (Remember that we are calculating the Mandelbrot Set for points on the complex plane that correspond to grid intersections—pixels—on our display.) Since the dimensions of the canvas where the fractal is drawn are 512×512 pixels, we must repeat the transformation 512×512 times. We've set up two nested loops, one through the rows of pixels and another through the pixels of each row. At each iteration, the program calls the Mandelbrot() function, which returns the number of iterations it took for the current point to escape to infinity. For points that persist in remaining bounded, the transformation must be repeated `MaxIterations` times. The current pixel is colored by a color that reflects how long it took the current point to escape to infinity.

The essential calculation that discovers whether or not our pixel is an escapee takes place in the Mandelbrot() function, whose code is shown in Listing 20.9.

LISTING 20.9: THE MANDELBROT() FUNCTION

```
Function Mandelbrot(ByVal Cx As Double, _
                   ByVal Cy As Double, _
                   ByVal maxIter As Integer) As Integer
    Dim iter As Integer
    Dim X2, Y2 As Double
    Dim X, Y As Double
    Dim temp As Double
    While iter < maxIter And (Sqrt(X2 * X2 + Y2 * Y2) < 4)
        temp = X2 - Y2 + Cx
        Y = 2 * X * Y + Cy
        X = temp
        X2 = X * X
        Y2 = Y * Y
        iter = iter + 1
    End While
    Return (iter)
End Function
```

The Mandelbrot() function returns the number of iterations executed. If the function repeated its calculations `MaxIter` times, the point is a prisoner. If the value returned by the function is smaller than `MaxIter`, the point is an escapee and it's colored according to the number of iterations it took for the point to escape. The first five lines of the While loop multiply a complex number by itself and add the constant `c`, as we described earlier in the chapter. (For those who really want to know the gory details, multiplication and addition of complex numbers will be explained at the end of this chapter.) If the point escapes quickly to infinity, we exit the loop before reaching the maximum number of iterations.

In our code we're using a structure to store the coordinates of the current fractal (these statements are not shown here, because they have nothing to do with the fractal calculations; you'll find them in the project's code). We've set up an ArrayList, the `limits` ArrayList, which we populate with instances of the Bounding structure, so that we can move to the previous/next fractal easily. The definition of the Bounding structure is:

```
Structure Bounding
    Dim XMin As Double
    Dim XMax As Double
    Dim YMin As Double
    Dim YMax As Double
End Structure
```

You can look up the project's code to see how the `limits` ArrayList is used in the code. We keep track of the bounding rectangle of each fractal, so that we can return to it should we zoom into an area of no interest. You can expand this structure to keep track of the number of iterations and colors used to produce each fractal.

The Julia Set

We will now temporarily leave the Mandelbrot Set to describe how to generate a different type of fractal image, known as Julia Sets. This is a second major style of fractal. A Julia Set (see Figures 20.6–20.8) is generated by the same process as a Mandelbrot Set, only here the function $z = z^2 + c$ is defined differently. In a Julia Set, the variable `z` takes on every value of the grid, and `c` remains the same throughout the entire process. The constant `c` can have any value, as long as both its parts (the real and the imaginary part) are between `-2` and `2`. In other words, `c` is added to the square of each point in the complex plane, and the same transformation is repeated over and over.

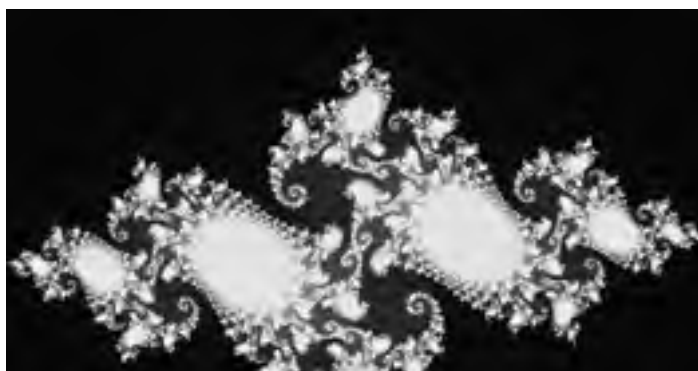




FIGURE 20.6 This is the "classic" Julia Set shape, but Julia Sets beget many different shapes depending on what numbers you feed them.

[Team Fly](#)

 Previous

Next 



FIGURE 20.7 Here's another Julia Set, quite distinct from Figure 20.6.



FIGURE 20.8 And yet another variation of the Julia Set

Unlike the Mandelbrot Set, whose shape is always the same, a Julia Set can be many different designs, depending on the value of the constant c . Different values of c result in quite different images. However, every one of those images can, like any other fractal, be zoomed into. And the zooming will reveal, variously, shapes and designs that resemble (but are not quite identical to) the original. This is why fractal geometry is sometimes referred to as self-similar.

Unfortunately, not all c values yield interesting Julia Sets, and it's not always easy to come up with a good value for the parameter c . The value used most often to produce the "classic" Julia Set is $-0.74543 + i0.11301$. (We'll suggest a few other good c values later in this chapter.)

PROGRAMMING JULIA SETS

The VB code for generating a Julia Set is virtually identical to the code for the Mandelbrot Set (see the Julia application). Actually, the two sample applications of this chapter have an identical user interface, but they produce different types of fractals. The difference is how we apply the transformation to each point of the complex plane. The JuliaSet() function is shown in Listing 20.10.

[Team Fly](#)

 Previous

Next 

LISTING 20.10: THE JULIASET() FUNCTION

```
Private Function JuliaSet(ByVal cx As Double, _
                        ByVal cy As Double, ByVal X As Double, _
                        ByVal Y As Double, ByVal MaxIter As Integer) As Integer
    Dim iter As Integer
    Dim x2 As Double, y2 As Double
    Dim xtemp As Double, ytemp As Double

    iter = 0
    x2 = X + cx
    y2 = Y + cy
    While ((iter < MaxIter) And ((Abs(x2) + Abs(y2)) < 100000))
        xtemp = x2 * x2 - y2 * y2 + cx
        ytemp = 2 * x2 * y2 + cy
        x2 = xtemp
        y2 = ytemp
        iter = iter + 1
    End While
    Return iter
End Function
```

The first two arguments, c_x and c_y , are the real and imaginary parts, respectively, of the complex constant c . Just like the `MandelbrotSet()` function, the `JuliaSet()` function returns the number of repetitions required for the point to escape to infinity. Again, if the point under consideration remained bounded up to the maximum number of iterations, it is considered bounded and we color it black. Otherwise, it's colored according to how fast it escaped.

COLORING FRACTALS

Different colors represent different escape velocities. Points that escape to infinity with different velocities are colored differently. So, although you could assign any colors you want to represent the various speeds of escape or entrapment of each point, there must nonetheless be a consistent pattern of color created within any single Julia or Mandelbrot Set. In other words, red could indicate only the fastest escapees or could mean only the bound points or some other in-between escape velocity. So what red means in general is up to you. But all points colored the same shade of red within a given fractal image share the same (or a similar) velocity.

These mysterious variations of escape velocity (and the resulting patterns of position and color) give fractals their beauty and complexity. We mentioned earlier that once the absolute value of the real or imaginary part of the point under consideration becomes larger than 2, the point will eventually escape. We could have used this well-known fact to end our calculations sooner and make our programs slightly faster, but when drawing colored fractals we are also interested in the velocity of escape, not in the mere fact that a point will eventually escape. So, we permit iterations to go on longer than strictly necessary because this provides us with the information about escape velocity, which we use to determine the color of each point. (By gathering more information about velocity, we can increase the color detail within our fractal.)

[Team Fly](#)

 Previous

Next 

There are many ways you could decide to represent the various escape velocities of these points with colors, but if you think about it, you'll realize it's not easy to come up with a color set for a fractal. Fractals are unpredictable by nature, and there is no way to predict which colors will do best in any particular complex image. Should we color red the points that escape immediately to infinity, or those that take longer? And how many basic colors should we use in the palette? And often of supreme importance to the look of the final picture—how will the palette be organized? Do we want a gradual shift from one shade at the start, to a comparable shade at the end, such as light to dark blue? Or perhaps a rainbow effect with all colors included?

The fractal algorithms won't help us, so we'll follow a different approach. We will attempt to design a color palette that satisfies two basic requirements. First, transitions among colors must be as smooth as possible. Let's say we have a point that escapes to infinity after 100 iterations, and it is colored red. It's likely that some of its neighboring points will escape to infinity after 101 or 99 iterations. Often a better-looking image results if these points are colored with a similar tone of red, and not a tone of green or blue. Second, we don't necessarily want many different colors. The most pleasing effects can result from a palette with a few basic colors, widely spaced apart and separated by gradual transition shades. Such a palette yields the most smoothly colored fractals.

The smoothest palette is one built from a single color, just a gradient of shades of the same color. A palette with 256 tones of gray, from black to white, is the smoothest possible palette. It contains only two basic colors and as many of their transition tones as you can get on a computer. (You can also create other palettes, which contain smooth transitions between, for example, very dark red and white, with all the pinks between.)

In our code we've implemented a palette with 2,048 colors. The first color is black. The next 255 colors are the shades between black and red. In effect, we walk through the color space between black and red. Then we walk the space between the red and yellow colors. The next 256 colors are shades between red and yellow. Then we move to cyan, blue, and so on. If you're familiar with the color cube, you've already understood that we construct our palette by walking through the corners of the color cube, passing through each possible shade between them. Listing 20.11 shows the code of the PaintPixel() function, which accepts as argument the number of iterations and returns the color of the palette that corresponds to this index.

LISTING 20.11: MAPPING ITERATIONS TO COLORS

```
Private Function PaintPixel(ByVal clr As Integer) As Color
    Select Case clr
        Case 0 To 256 - 1
            Return Color.FromArgb(clr, 0, 0)
        Case 256 To 256 * 2 - 1
            Return Color.FromArgb(255, clr - 256, 0)
        Case 256 * 2 To 256 * 3 - 1
            Return Color.FromArgb(255, 255, clr - 256 * 2)
        Case 256 * 3 To 256 * 4 - 1
            Return Color.FromArgb(255, clr - 256 * 3, clr - 256 * 3)
        Case 256 * 4 To 256 * 5 - 1
            Return Color.FromArgb(255, 256 * 5 - clr - 1, 255)
```

[Team Fly](#)

 Previous

Next 

```
Return Color.FromArgb(clr - 256 * 5,
                      clr - 256 * 5, clr - 256 * 5)
Case Else
Return Color.FromArgb(clr Mod 256, clr Mod 256, clr Mod 2
End Select
End Function
```

The Real Magic of Fractals

Up to now, we've been variously limited by ranges of color, organization of the palette, and seeing only the outermost view of the generated fractal. The Mandelbrot and Julia applications allow you to zoom in on a given fractal so that you can see how colors and shapes emerge and vary, the farther into a fractal you move. (Generally, the first, outermost view of a fractal contains less variety of color. Often, the real beauty of a fractal is hidden among the filaments. To really explore a fractal, move repeatedly down through the filaments within.) To zoom into a specific area of a fractal, use the mouse to draw a rectangle that encloses the desired area and then click the Draw Fractal button.

Once you start zooming into a fractal, you must adjust the minimum and maximum number of iterations, as well as the two end colors. The sample applications of this chapter are not the ultimate fractal generators, but they're short Visual Basic applications that can serve as a vehicle for your own exploration of the world of fractals. The two end colors on the application's interface are specified by their index in the 2048 palette. A starting index of 0 and an ending index of 256 correspond to a palette of shades from black to red. A starting index of 512 and an ending index of 1024 yields a palette with 256 shades from yellow to red, followed by 256 shades from red to white. You can edit the code of the PaintPixel() function to create any palette, or add more colors to it.

To control the appearance of each fractal you can adjust the values of the min/max iterations and the min/max colors. The code maps the iterations to colors, and the two ranges shouldn't be very different. For example, if the number of iterations goes from 0 to 64 and the number of colors from 0 to 1024, you won't have a very smoothly colored fractal. You'll have to experiment with the settings of these two parameters a little to get a nice image. Moreover, the settings that will work for one fractal will not work for another one. As you zoom deeper and deeper into a fractal, you'll have to increase the number of iterations, because it's the border of the Mandelbrot Set that contains the most intriguing patterns (points that take a long number of iterations to escape).

Figures 20.9 through 20.14 illustrate the unique nature of fractals. Fractal images do not have "resolution." They never run out of shapes or colors, no matter how deep you zoom into them. Think of the zoom operation as an electronic microscope that probes the deep fractal space. There are limits to how deep you can probe, but these limits are due to the instrument, not the object you are examining. The power of your electronic microscope goes as far deep as double-precision numbers can take it. Keep zooming and you will keep discovering new patterns, until you reach the smallest number Visual Basic can represent. This is where your microscope's resolution, but not the fractal space, ends.

The Mandelbrot Set, just like the Julia Set, exhibits the self-similarity property. Successive magnifications of the same spot in the set look more like different sections of the same image, rather than drastically different magnifications of the same image. Although unreal, they look familiar and remind you of anything from trees to galaxies.

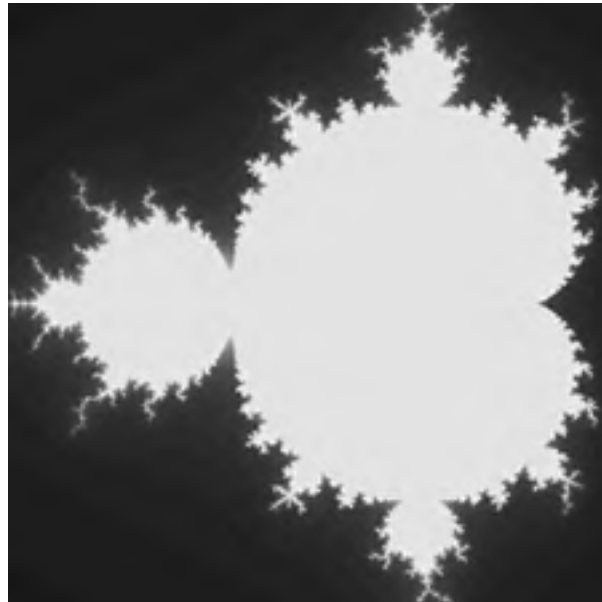


FIGURE 20.9 The Mandelbrot Set in its entirety



FIGURE 20.10 Zooming into the top left area of the fractal of Figure 20.9

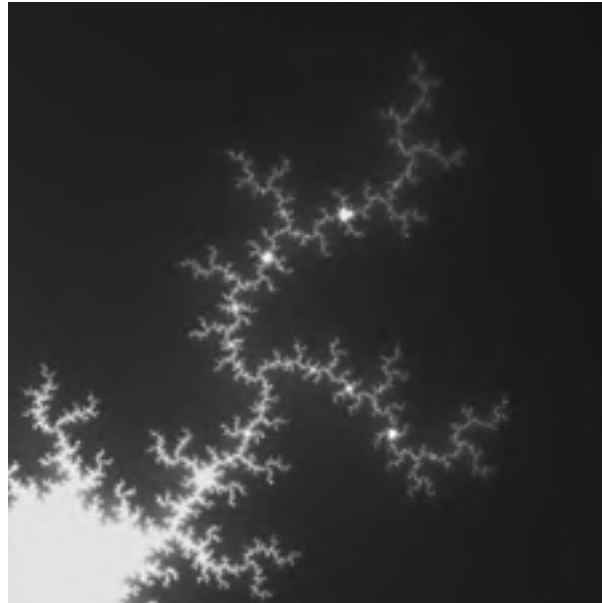


FIGURE 20.11 Exploring the edges of the fractal of Figure 20.10

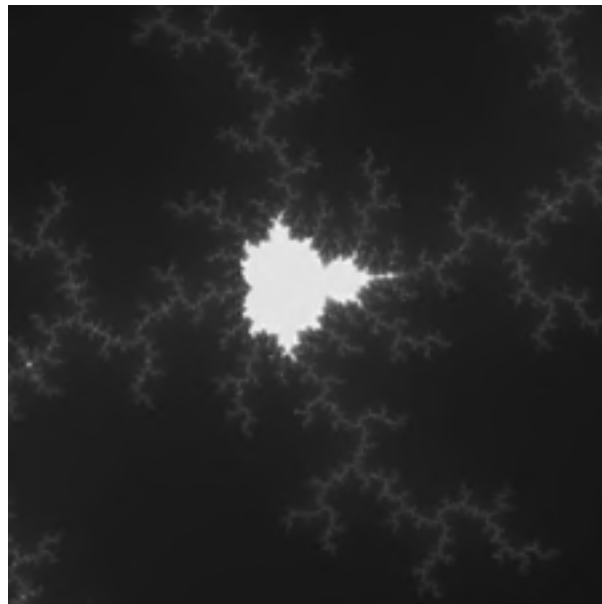


FIGURE 20.12 Discovering a replica of the original Mandelbrot Set in the filaments of the fractal of Figure 20.11

The fractals in the figures of this chapter will be rendered in shades of gray. We will post the same images in color at the book's site, along with the chapter's projects.

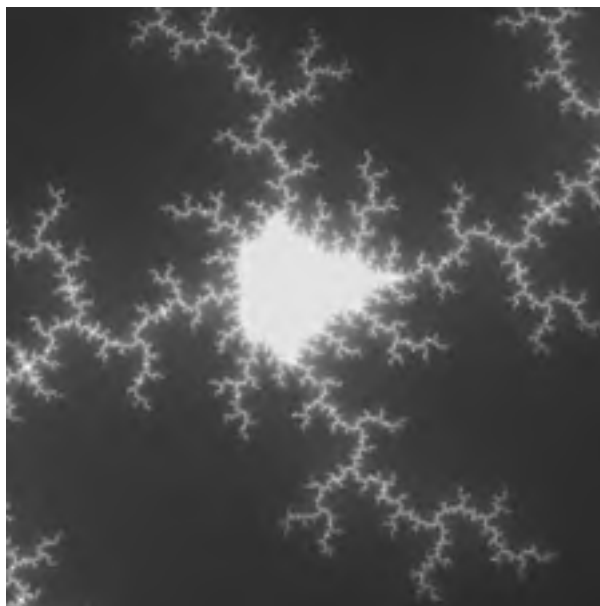


FIGURE 20.13 The same fractal of Figure 20.12 drawn with 64 iterations only

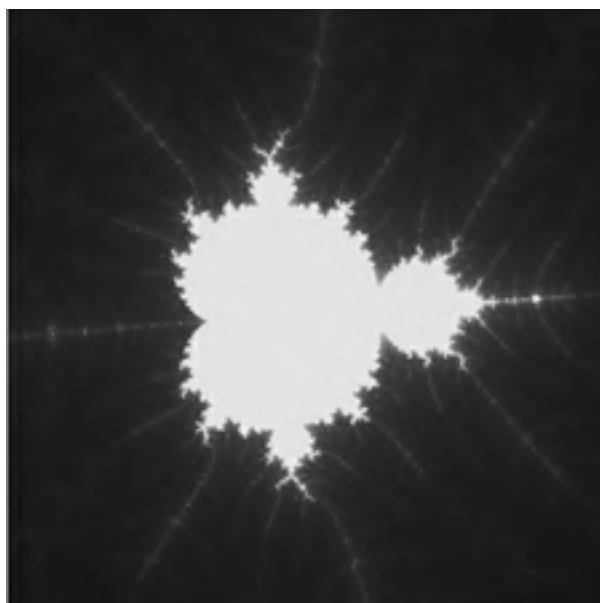


FIGURE 20.14 The same fractal once again, this time computed with 512 iterations

SOME GREAT JULIAS

While the Mandelbrot Set doesn't vary (but you can discover an infinite number of patterns as you zoom into its borders), the Julia Set is based on an initial point on the complex plane (the parameters C_x and C_y). Here is a list of numbers you can try that produce interesting Julia Sets (or you can

LISTING.1: BINARY SERILIALIZATION OF AN ARRAYLIST COLLECTION

```
Private Sub btnSerialize_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) _  
    Handles btnSerialize.Click _  
' UNCOMMENT THE TWO STATEMENTS IN THE PROCEDURE AND COMMENT OUT THE  
' FOLLOWING STATEMENTS TO SERIALIZE THE SAME OBJECTS IN BINARY FORMAT  
    Dim objects As New ArrayList  
    objects.Add(New Rectangle(10, 10, 100, 160))  
    objects.Add(New Bitmap( '..\sample.gif" ))  
    Dim BF As New BinaryFormatter  
'Dim SF As New SoapFormatter  
    Dim Strm As New FileStream( "..\Objects.Bin", FileMode.OpenOrCreate)  
    Strm.SetLength(0)  
    Try  
        BF.Serialize(Strm, objects)  
        'SF.Serialize(Strm, objects)  
    Catch ex As Exception  
        MsgBox( "Serialization failed." & vbCrLf & _  
            ex.Message)  
        Strm.Close()  
    Exit Sub  
    End Try  
    Strm.Close()  
    MsgBox( "Collection serialized successfully ")  
End Sub
```



FIGURE 3.1 The BasicSerialization project demonstrates how to serialize and deserialize built-in objects.

Since i is the square root of -1 , the square of i is -1 . The term $ib * id$, therefore, is reduced to $-bd$. Then we separate the real and imaginary parts, to form the new complex number:

$$a*c + ib*c + ia*d + ib*id = a*c + i(b*c + a*d) -b*d = (a*c - b*d + i(b*c + a*d)), \text{ or } (a*c - b*d, b*c + a*d)$$

Here's an example of complex number multiplication:

$$(3, i7) * (-2, i4) = (3*(-2) - (7*4), 7*(-2) + 3*4) = (-34, -2)$$

The Transformation $z = z^2 + c$

Let's follow the lines of the Mandelbrot function that implements the transformation $z = z^2 + c$. z is a point on the complex plane, defined by two numbers: the real and imaginary part. Let's call these parts x and y . Similarly, c is another complex number whose real part is Cx and imaginary part is Cy . The previous transformation can now be written as

$$z = (x + iy)^2 + (Cx + iCy)$$

The square of a complex number is calculated as follows (it's a direct application of the multiplication formula given earlier):

$$(x + iy)^2 = (x + iy)*(x + iy) = x^2 - y^2 + i2xy$$

The result is another complex number, whose real part is $(x^2 - y^2)$ and imaginary part is $2xy$. It can also be written as $(x^2 - y^2 + i2xy)$.

Then we add the constant C to the result. The new number z becomes:

$$((x^2 - y^2) + i2xy) + (Cx + iCy) = (x^2 - y^2 + Cx) + i(2xy + Cy)$$

The result of the operation $z^2 + c$ is another complex number, whose real part is $(x^2 - y^2 + Cx)$ and imaginary part is $(2xy + Cy)$. This number becomes the new value of the variable z , and the process continues. Look at the Mandelbrot function to see that this is how it implements the basic transformation $z = z^2 + c$. The variables x^2 and y^2 hold the values x^2 and y^2 respectively.

Let's explain this further with a numeric example. We will apply the transformation to the point $(0.2 + i0.5)$ of the complex plane. We start with $z = (0 + i0)$, square it, and add the constant $(0.2 + i0.5)$:

$$z = z^2 + c = (0 + i0)^2 + (0.2 + i0.5) = (0.2 + i0.5)$$

This is the value of z for next transformation:

$$z = z^2 + c = (0.2 + i0.5)^2 + (0.2 + i0.5) = -0.01 + i0.7$$

This value is assigned to z , and the same transformation is repeated over and over again, until either the result exceeds a very large value or the maximum number of iterations is exhausted.

You know the effect. Which stranger do you trust more: the woman in stained sweatpants with oily hair chewing gum? Or the calm, smart woman in a fitted beige silk dress, wearing a single strand of pearls around her clean, strong neck?

Beyond the value of first impressions, users come to rely on visual conventions, the rules of the road. For example, notice the difference between the forms in Figures 21.1 and 21.2:



FIGURE 21.1 BEFORE. Avoid flat, haphazard, dull, childish-looking forms.



FIGURE 21.2 AFTER. Much improved by adding a gradient, a 3D frame, balancing the controls, turning off boldface text, and other adjustments.

See how many differences you can detect between Figures 21.1 and 21.2. There are 14 improvements in Figure 21.2:

- An icon appropriate to the application's purpose (a clock for a scheduling application, for instance) is located in the upper left corner. A large collection of well-designed icons can be found in `\Program Files\Microsoft Visual Studio .NET2003\Common7\Graphics\Icons`. If you don't see this set of icons, you can rerun VS.NET setup and choose to install them.

- ◆ The form has a descriptive title, "Schedule" (the form's Text property).
- ◆ The title "Schedule" is preceded by a space character when typed into the Properties window. This looks better than jamming it up against the icon too closely. Various defaults in VB.NET aren't the best design choice, and this is one of them.
- ◆ A second TextBox is superimposed on the existing TextBox. The inner TextBox is sized slightly smaller than the outer (original) one. This way the outer TextBox provides the framing and the inner TextBox can be adjusted to provide attractive margins. The inner TextBox's BorderStyle property is set to None. The whole purpose of this nesting is to move the text down from the top and over from the left a bit. By default, the text in a TextBox is jammed up against the top and left sides too much (compare the margins of the text *12 November - Maintenance* in the two figures). Figure 21.2 has comfortable, nice-looking margins; Figure 21.1 doesn't.
- ◆ The TextBox's BorderStyle property is changed from FixedSingle to Fixed3D.
- ◆ The Button's FlatStyle property is changed from Flat to Standard.
- ◆ The TextBox's FontName property is changed from Sans Serif to Times New Roman. The sans serif (or Arial) fonts are good for button text, form text, and other "headline" or "title" uses. However, most people prefer a serif (curlicue) font for body text. Books, magazines, newspapers—and TextBoxes too—almost universally employ a serif font such as Times for their smaller body text. It's easier to read because the letters are more distinctive, especially at the smaller font sizes typical of narrative text. Unfortunately, the TextBox defaults to sans.
- ◆ The TextBox font was changed from boldface to regular.
- ◆ The Button font was changed from boldface to regular.
- ◆ The form, TextBox, and button were resized and repositioned to ensure that the borders between them are symmetrical. In other words, the distance between the TextBox's left and right sides—and the form's edges—should be identical. Likewise, the distance between the bottom of the TextBox and the top of the button should match the distance between the bottom of the button and the bottom of the form. You can use the format menu's Align, Make Same Size, and Spacing tools to get things right.
- ◆ The button text has been changed from *Exit* to *Close*. The term *Exit* is reserved for shutting down an application. This is merely one of several forms in this application. Dialog boxes and secondary forms are closed using buttons captioned variously *OK*, *Save*, *Cancel*, or *Close*. But never *Exit*.
- ◆ The text on the button has a subtle etched effect (see Figure 21.3). More on this technique later in this chapter.
- ◆ The button has a gradient background (more on this later). By default, the backgrounds of most controls and the forms are flat.
- ◆ The form background also has a gradient background.



FIGURE 21.3 Etched text looks good on buttons.

[Team Fly](#)

 Previous

Next 

If you don't like the metallic look, at least think about the colors you choose. To some extent, the user's Windows display properties settings determine the look (such as the icons on a form's title bar), but you have control over most elements in your forms.

FontBold Off

Second, FontBold should be False (normal text weight is used, not boldface) in the button captions, menu titles and other text in most applications, including those from Microsoft.

Normal text looks neater and cleaner than boldface text. Reserve boldface for highlighting something—to indicate, for instance, that a particular button has the focus (will activate if you press Enter). More than an occasional touch of boldface is crude and unsightly.

Using a Sans-serif Typeface for Headlines

VB's default Font Property is MS Sans Serif, which is also the Windows default typeface. Of the fonts that ship with Windows, if you want to use a sans face, choose Arial. It's the best choice for most titles, captions, and headlines (but not body text). Arial differs from the default MS Sans Serif primarily in that it improves the appearance of the characters because the midpoint of its uppercase letters is lower than the midpoint of MS Sans Serif's capitals. Arial also improves readability by spacing its letters somewhat more widely apart.

Choosing a Type Size

For most purposes, the default FontSize (8.25) is too small. Normally, for typical screen resolutions, you should adjust body text (captions, labels, TextBox text) to 11, and make headlines and other large text what looks best—something between 20 and 40 points is usually best.

Layering

Another technique that adds dimensionality is layering, a kind of rice-paddy effect whereby you divide your form into logical zones. Take a look at all the controls on your form. If necessary, make lists of which controls go together. Does this label describe this TextBox's function? Does this button cause something to happen to the contents of this ListBox? Ensure that controls that work together are physically located near each other, and even contained together within a GroupBox or Panel. You can also use the new Splitter control, giving the user the ability to adjust the size of the zones on your form. Sometimes you even want to put a Panel within another Panel, to further subdivide the functions of the controls—this is the rice-paddy effect.

Adding Depth

The first and most important graphics technique when designing visually pleasing windows is to add depth, to make them look three-dimensional. Flat, monochromatic screens should never be used in Windows. No design error makes your application look more amateurish than a lifeless, boring, flat surface.

Not only does a sculpted, 3D window look more attractive, it also more efficiently conveys information to the user. One reason is that we are not Cyclops—we have two eyes, so depth, the Z-axis, provides us with additional useful information. Also, the user is supposed to interact with some controls,

[Team Fly](#)

 Previous

Next 

and these controls should have highlights and shadows to make them look physical, to make them look as if clicking on them will in fact do something.

A button looks as if it could be depressed; it seems to protrude from the window (unlike Labels, which seem merely printed on the flat surface of the window). The Windows user understands that a captioned three-dimensional rectangle is a button—just like buttons on appliances in the real world. Likewise, the user understands that the flat printing on Labels is merely informational—it describes the purpose or contents of something, but nothing will happen if you click it. Sculpted controls such as RadioButtons are interactive. Almost everyone knows to press the protruding button labeled POWER on a stereo—few try to press the label POWER itself.

Buttons look best if you leave their FlatStyle property alone. Change FlatStyle and it goes, yes, flat: designed to look like a label until a mouse pointer either moves over it or clicks it (the Flat version turns gray, the PopUp version becomes dimensional—as shown in Figure 21.4). Also unsettling is that any FlatStyle other than the default "Standard" adds a retro dashed line around the button to indicate selected. This is old-style Windows. The current, superior selected style has a dark outline, as shown in the figure. The strange FlatStyle.System option should be avoided at all costs. It causes the button to become narrower, and is intended for people who are forced to create platform-independent user interface designs (that's oxymoronic, if you think about it).



FIGURE 21.4 Important visual clues distinguish various button styles.

Light from the Upper Left

Notice the top two buttons in Figure 21.4. See how Windows draws a button before (left) and after (right) the Button is clicked. In Windows, the light is always assumed to be coming from the upper left corner of the screen. Therefore, a protruding object like a button picks up highlights along its top and left sides. Shadows fall along the right and bottom of the object. However, if the object is supposed to be sinking into the background, the process is reversed.

Interestingly, most light sources in the real world usually follow this same pattern. People reading outdoors often maneuver their position until the sun shines from their left side (and, of course, above) onto the book. People often put lamps to the left of their desk and above it, to the left of a reading chair, and so on. Look at your computer monitor. If you're like most people, your lamp or ceiling light

(the primary light source) is above and to the left. And, in consequence, the top and left of your monitor will pick up the highlights, the bottom and right will be in shadow. As an experiment, turn this book upside down and see what happens to your perception of the shapes in Figure 21.4. And if you work in a cubicle with the light source in some other location, tell your boss that's the reason you've been moaning so much lately, or whimpering or whatever it is that you do when distressed.

Creating Zones

When you start designing a form, you'll decide which controls you're going to use. But don't put them on right away. First determine which controls will be merely decorative, which informational, and which interactive. Then add some zones to visually separate these control categories. Most controls have their own frames, but you want to use GroupBox controls to frame other, related controls (such as a set of CheckBoxes).

Figure 21.5 illustrates excellent user-interface design. Information is displayed effectively and efficiently by using various techniques—particularly by subdividing the main window into logical zones. Also, there's considerable variety.

Notice in Figure 21.5 how many different areas and tools are arranged together to cue the user what to expect. There are tabbed windows, toolbars, menus, lists, slider bars, dropdown lists, menus, and a status bar. There's lots going on here, but there's also a logic to the arrangement.





FIGURE 21.5 Microsoft spends loads of cash refining user-interface design.

First, we create an `ArrayList` collection and add to it a `Rectangle` and a `Bitmap` object. Then we serialize the `ArrayList` by calling the `Serialize` method of the `BinaryFormatter` class. The `Serialize` method accepts two arguments—a `Stream` object where the result of the serialization will be written and the object to be serialized. As you can see, it doesn't get any simpler. The `structure` exception handler is needed in case one of the objects you're trying to serialize is not serializable. Many of the .NET Framework's built-in classes are not serializable and an exception will be thrown if you attempt to serialize them. You will see shortly how to create serializable custom classes, and you must make sure that any of your custom classes you plan to serialize are serializable.

This is the code needed to serialize an object in binary format, regardless of the object's complexity. In effect, all the work is done by a single statement, which calls the formatter's `Serialize` method. Most of the output's bytes are the contents of the bitmap, but you'll be able to figure out that it contains an instance of the `Rectangle` class (the string `"System.Drawing.Rectangle"` appears in the file), as well as an instance of the `Bitmap` class (indicated by the string `"System.Drawing.Bitmap"`). The dimensions of the rectangle are stored in binary format and you can't read them.

To use SOAP serialization, you must first add a reference to the `SoapFormatter` to your project. Open the Add Reference dialog box (menu Project ➤ Add Reference) and select the component `System.Runtime.Serialization.Formatters.Soap`. In the listing of the `BasicSerialization` project, we've commented out the statements that use the `SoapFormatter`. You can uncomment these statements and comment out the statements that perform binary serialization. Here's what the `Objects.bin` file looks like for the SOAP serialized `ArrayList` (the bytes making up the image are stored in Base64 encoding and we're showing only a few dozen bytes). The line breaks were inserted during the typesetting process; normally, there are no breaks in the serialized data.

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:
  xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
  ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:clr="http://schemas.microsoft.com/soap/encoding clr/1.0"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<al:ArrayList id="ref-1"
  xmlns:al="http://schemas.microsoft.com/clr/ns/System.Collections" >
  <_items href="#ref-2"/>
  <_size>2</_size>
  <_version>2</_version>
</al:ArrayList>
<SOAP-ENC:Array id="ref-2" SOAP-ENC:arrayType="xsd:anyType[16]" >
<item xsi:type="a3:Rectangle" xmlns:a3="http://schemas.microsoft.com/clr/nsas
  System.Drawing/System.Drawing%2C%20Version%3D1.0.5000.0%2C%20
  Culture%3Dneutral%2C%20PublicKeyToken%3Db03f5f7f11d50a3a">
<x>10</x>
<y>10</y>
<width>100</width>
<height>160</height>
</item>
<item href="#ref-4"/>
```


Framing

Framing can be done simply using a Panel or GroupBox control, and these controls are good when you want a quick way to subdivide your forms into logical zones. However, they are fairly limited in shape and color.

You may want to adjust the colors or style of your frames. You might want to use etched frames (cut-away framing, which appears carved into the surface of the form, such as a Panel with a Fixed3D BorderStyle or a GroupBox). Or you might prefer embossed frames (protruding, raised surfaces, such as a button control). When you really want to highlight something, consider using multiple frames, as illustrated in Figure 21.6:

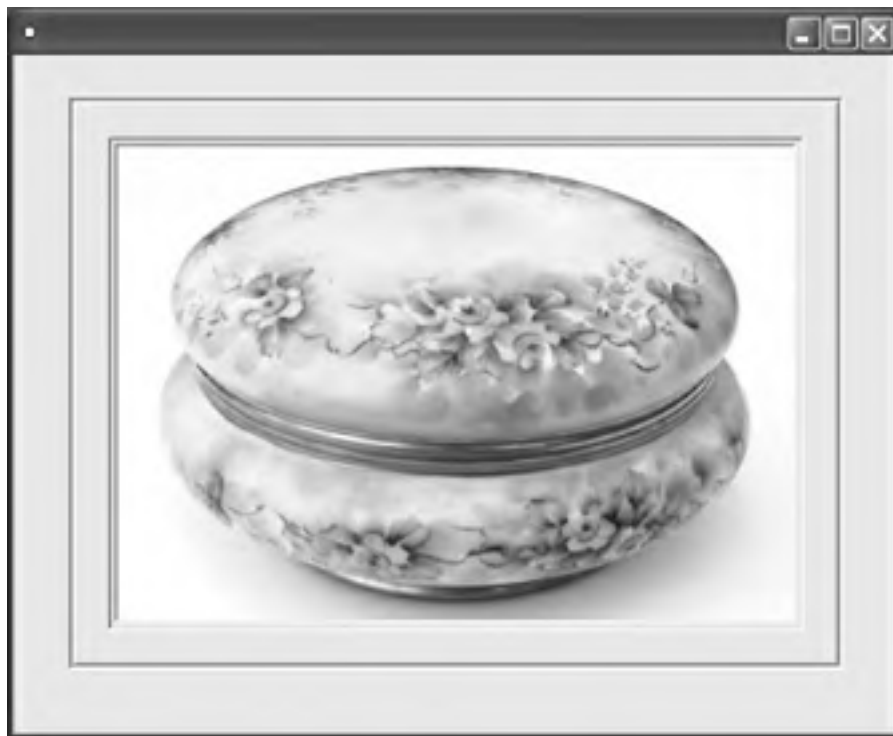


FIGURE 21.6 Add as many frames as you wish, designing layering, matting, and other styles.

Interestingly, the Microsoft designers have found that it's not enough to suggest a frame by using two shades (one for shadow and one for highlight). Two-tone shading is shown in Figure 21.7; four-tone shading (four different grays) is shown in Figure 21.8.





FIGURE 21.7 From a sufficient distance, or at a high resolution, two-tone shading works OK.



FIGURE 21.8 Four-tone shading creates a better effect. Adjust the framing procedure described in Listing 21.2 to suit yourself (it defaults to four-color).

The following code is a multipurpose framing procedure that can create all kinds of frames quickly and easily. Listings 21.1, 21.2, and 21.3 are a bit of typing, so you might want to download them from this book's website.

```
Imports System.Drawing
Imports System.Drawing.Drawing2D
Imports System.Drawing.Imaging
Imports System.Drawing.Text
```

The following variables can be used throughout the form, so they're not declared within any procedure:

```
Public x1, x2, y1, y2 As Integer
Public toprightx, toprighty, bottomrightx, bottomrighty, _
    bottomleftx, bottomlefty As Integer

'frame embossed (out), or etched (inn)
Dim inn As Boolean = False
Dim out As Boolean = True

'TOP pens (drawn along left and top sides)
Dim Lgray As New Pen(Color.FromArgb(113, 111, 100)) ' inner line (dark gray)
Dim DGray As New Pen(Color.FromArgb(172, 168, 153)) 'outer line (light gray)

'BOTTOM pens (drawn along right and bottom sides)
Dim Darkwhite As New Pen(Color.FromArgb(241, 239, 226))
'inner line (medium white)
Dim Lwhite As New Pen(Color.FromArgb(255, 255, 255)) 'outer line (white)
```

Each time you draw a new frame (you can draw as many as you want), you use this `SetPoints` procedure. You pass the name of the control you're framing (parameter `c`), and also the number of pixels out away from that control that you want the frame drawn (parameter `framesize`), as shown in Listing 21.1.

LISTING 21.1: DRAWING SOPHISTICATED FRAMES

```
Private Sub SetPoints(ByVal c As Control, ByVal framesize As Integer)
    ' define coordinates for frame

    Dim n As System.Drawing.Size = c.Size

    x1 = c.Left - framesize
    x2 = c.Top - framesize
    y1 = n.Width + (framesize * 2)
    y2 = n.Height + (framesize * 2)

    toprightx = x1 + y1
    toprighty = x2
    bottomrightx = toprightx
    bottomrighty = x2 + y2
    bottomleftx = x1
    bottomlefty = bottomrighty

End Sub
```

After you've used the `SetPoints` procedure to specify the size of the frame, you then call this `PaintFrame` procedure to actually draw the frame itself. You pass a `direction` parameter, which can be `Out` (which draws the frame outward as if embossed) or `In` (which draws the frame inward, as if etched), as shown in Listing 21.2.

LISTING 21.2: CREATING EMBOSSED AND ETCHED FRAMES

```
Private Sub PaintFrame(ByVal direction As Boolean)

    If direction = False Then

        'draw the frame in (etched style)

        ' Create a Graphics object.
        Dim g As Graphics = Me.CreateGraphics

        ' A lightgray rectangle starting at the upper left
        g.DrawRectangle(Lgray, New Rectangle(x1, x2, y1, y2))
        'a mediumgray rectangle one pixel inward
        g.DrawRectangle(DGray, New Rectangle(x1 - 1, x2 - 1, y1 +

        ' Two white lines from upper to the lower right
```

```
g.DrawLine(Darkwhite, toprightx, toprighty, bottomrightx, bottom
g.DrawLine(Lwhite, toprightx + 1, _
toprighty - 1, bottomrightx + 1, bottomrighty + 1)

' Two white lines from bottom right to bottom left
g.DrawLine(Darkwhite, bottomrightx, _
bottomrighty, bottomleftx, bottomlefty)
g.DrawLine(Lwhite, bottomrightx + 1, _
bottomrighty + 1, bottomleftx - 1, bottomlefty + 1)

' Kill the graphics object
g.Dispose()

Else ' do the frame out (embossed style)
' to create the embossed (outie) effect, you simply
' reverse the colors dark on right/bottom, light on top/l

' Create a Graphics object.
Dim g As Graphics = Me.CreateGraphics

' A lightgray rectangle starting at the upper left
g.DrawRectangle(Darkwhite, New Rectangle(x1, x2, y1, y2))
'a mediumgray rectangle one pixel inward
g.DrawRectangle(Lwhite, New Rectangle(x1 - 1, x2 - 1, y1

' Two white lines from upper to the lower right
g.DrawLine(DGray, toprightx, toprighty, bottomrightx, bot
g.DrawLine(Lgray, toprightx + 1, _
toprighty - 1, bottomrightx + 1, bottomrighty + 1)

' Two white lines from bottom right to bottom left
g.DrawLine(DGray, bottomrightx, bottomrighty, bottomleftx
g.DrawLine(Lgray, bottomrightx + 1, _
bottomrighty + 1, bottomleftx - 1, bottomlefty + 1)

' Kill the graphics object
g.Dispose()

End If

End Sub
```

The `SetPoints` and `PaintFrame` procedures are called within the form's `Paint` event, so that each time the form is repainted, the frame is redrawn. This is necessary because drawn designs don't persist during `Form_Load`, or when the user covers this form with another, resizes it, and so on. The code in Listing 21.3 draws three frames around the `PictureBox` (as shown in Figure 21.9).

LISTING 21.3: CREATING MULTIPLE FRAMES

```
Private Sub Form1_Paint(ByVal sender As Object, _  
    ByVal e As System.Windows.Forms.PaintEventArgs) Handles MyBase.Paint  
  
    'define the frame sizes (passing the name of the control  
    ' to be framed, distance of frame from the control)  
  
        SetPoints(PictureBox1, 3) 'innermost frame  
        PaintFrame(inn)  
  
        SetPoints(PictureBox1, 25)  
        PaintFrame(out)  
  
        SetPoints(PictureBox1, 28)  
        PaintFrame(inn) 'outermost frame  
  
End Sub
```



FIGURE 21.9 A close-up of the three drawn frames, plus the `PictureBox`'s own frame. Note that the second drawn frame is the `out` style—with the highlight on the top.

Fade In, Fade Out

Here's how to do a fade, like the one shown in Figure 21.11:

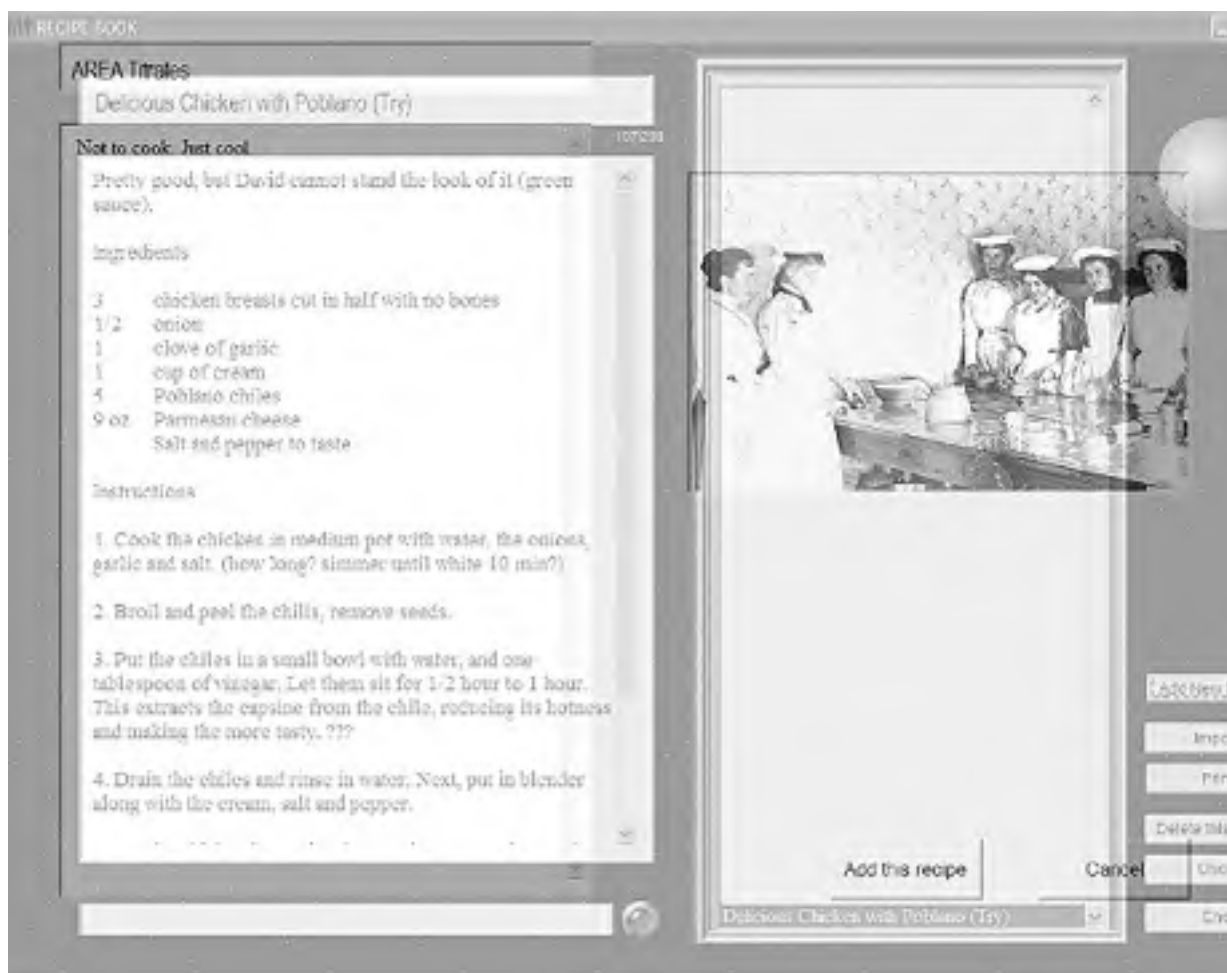


FIGURE 21.11 The main program screen fades out as a data entry screen fades in.

Create a project with two forms. Above the `Form1_Load` procedure in `Form1`, type this global variable that instantiates `Form2` (you've already designed `Form2`, so it's ready to be brought to life):

```
Public f2 As New Form2
```

Also above their form load events, add positioning variables to both `Form1` and `Form2`

```
Private x As Single = 0  
Private y As Single = 1
```

Then, in the `Click` event of the button that triggers the fade transition, use this code to hide the current form and set `Form2`'s opacity to 0 or complete invisibility.

```
Public Sub Button1_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles Button1.Click  
  
    f2.Opacity = 0  
    f2.Show()  
    Timer2.Enabled = True 'fade routine  
  
End Sub
```

[Team Fly](#)

 Previous

Next 

The Opacity property ranges from 0 to 1, rather than from 0 to 100 as you would expect. Apparently 0 to 1 is someone's idea of the proper "scientific" way to express a range of percentages, so there's a performance hit caused by not using integers here.

The actual fade-out that takes Form1 from visible to invisible occurs in Timer2. Note that in VB.NET timers default to `Enabled = False`.

```
Private Sub Timer2_Tick(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Timer2.Tick
    'fade out Form1
    y -= 0.04
    If y <= 0.01 Then Timer2.Enabled = False : y = 1 : Me.Hide() : f2.Show()
    Me.Opacity = y
End Sub
```

Now, in Form2's code window, you detect that the VisibleChanged event has been triggered, and you start the fade-in process to make Form2 visible, as shown in Listing 21.5.

LISTING 21.5: CREATING A FADE TRANSITION

```
Private Sub Form2_VisibleChanged(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles MyBase.VisibleChanged

    Timer1.Enabled = True

End Sub
In Form2's Timer1, this code makes Form2 visible:
Private Sub Timer1_Tick(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Timer1.Tick
    'fade in

    x += 0.04

    If x >= 1 Then Timer1.Enabled = False : x = 0 : Me.Opacity =

    Me.Opacity = x

End Sub
```

And, to reverse the process, you have a Cancel button on Form2 that fades Form2 out and fades Form1 in Listing 21.6.

LISTING 21.6: REVERSING A FADE

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

    Timer2.Enabled = True

End Sub

Private Sub Timer2_Tick(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Timer2.Tick
    'fade out

    y -= 0.04

    If y <= 0.01 Then Timer2.Enabled = False : y = 1 : Me.Hide()
    Me.Opacity = y

End Sub
```

And back in Form1's VisibleChanged event, you trigger yet another timer devoted to fading Form1 in Listing 21.7.

LISTING 21.7: FADING IN

```
Private Sub Form2_VisibleChanged(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles MyBase.VisibleChanged

    Timer1.Enabled = True

End Sub

Private Sub Timer1_Tick(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Timer1.Tick
    'fade in

    x += 0.04
    If x >= 1 Then Timer1.Enabled = False : x = 0 : Me.Opacity =

    Me.Opacity = x

End Sub
```

```
</SOAP-ENC:Array>
<a3:Bitmap id=''ref-4"
xmlns:a3="http://schemas.microsoft.com/clr/nsassem/System.Drawing/
System.Drawing%2C%20Version%3D1.0.5000.0%2C%20Culture%3D
neutral%2C%20PublicKeyToken%3Db03f5f7f11d50a3a" >
<Data href="#ref-5"/>
</a3:Bitmap>
<SOAP-ENC:Array id="ref-5" xsi:type="SOAP-
ENC:base64">R0lGODlhfAHgAvcAAAUGBhQGBggUG
BEWFx8PCSEYECEQGBkbGysQCTEYECUYHC0aGiEhGCEhIS4hFSkhIRQl
JyElJSEpKSExKTEhIS0pHCKhKTEtIRg
```

Now enter the statements of Listing 3.2 in the Deserialize Collection button's Click event handler. These statements read the contents of the Objects.bin file, deserialize them with the Deserialize method of a BinaryFormatter object, and recreate a copy of the original ArrayList, the persistedObjects ArrayList. The program prints the basic properties of the two objects it deserialized and then displays the bitmap on the form.

LISTING 3.2: DESERIALIZING A BINARY FILE INTO AAN ARRAYLIST COLLECTION

```
Dim persistedObjects As New ArrayList
Dim Strm As New FileStream( "..\Objects.Bin", FileMode.Open)
Dim BF As New BinaryFormatter
Try
    persistedObjects = CType(BF.Deserialize(Strm), ArrayList)
Catch ex As Exception
    MsgBox("Deserialization failed." & vbCrLf & _
        ex.Message)
    Strm.Close()
Exit Sub
End Try
Console.WriteLine(persistedObjects(0).GetType)
Dim R As Rectangle = CType(persistedObjects(0), Rectangle)
Console.WriteLine("RECT. WIDTH:" & R.Width.ToString & _
    ", HEIGHT: " & R.Height.ToString) &
Dim IMG As Bitmap = CType(persistedObjects(1), Bitmap)
Console.WriteLine("IMAGE WIDTH: " & IMG.Width.ToString & _
    ", HEIGHT: " & IMG.Height.ToString)
Me.BackgroundImage = IMG
```

The deserialization process, like the serialization process, requires a single statement that calls the Deserialize method. This method accepts as argument the stream with the persisted data and returns a variable of the Object type. You must cast the result returned by the Deserialize method to the appropriate type before using it in your code (unless you don't mind late binding, of course). This object is cast to the ArrayList type and assigned to the persistedObjects variable, which is an ArrayList.

I realize this seems like a bit of trouble, but the effect is so cool that it's quite worth it. Give it a try. You can adjust the granularity of the fades by changing the lines in the timers: `x += 0.04`. A good value for the timers' Interval property is 3, but fiddle with this setting if you wish as well to find a fade speed that pleases you.

Sliding

Sliding one form on top of another (or sliding a control such as a ListBox) is a useful and attractive technique. In Figure 21.12, the marble panel slides up and down, allowing the user to enter or change a password.

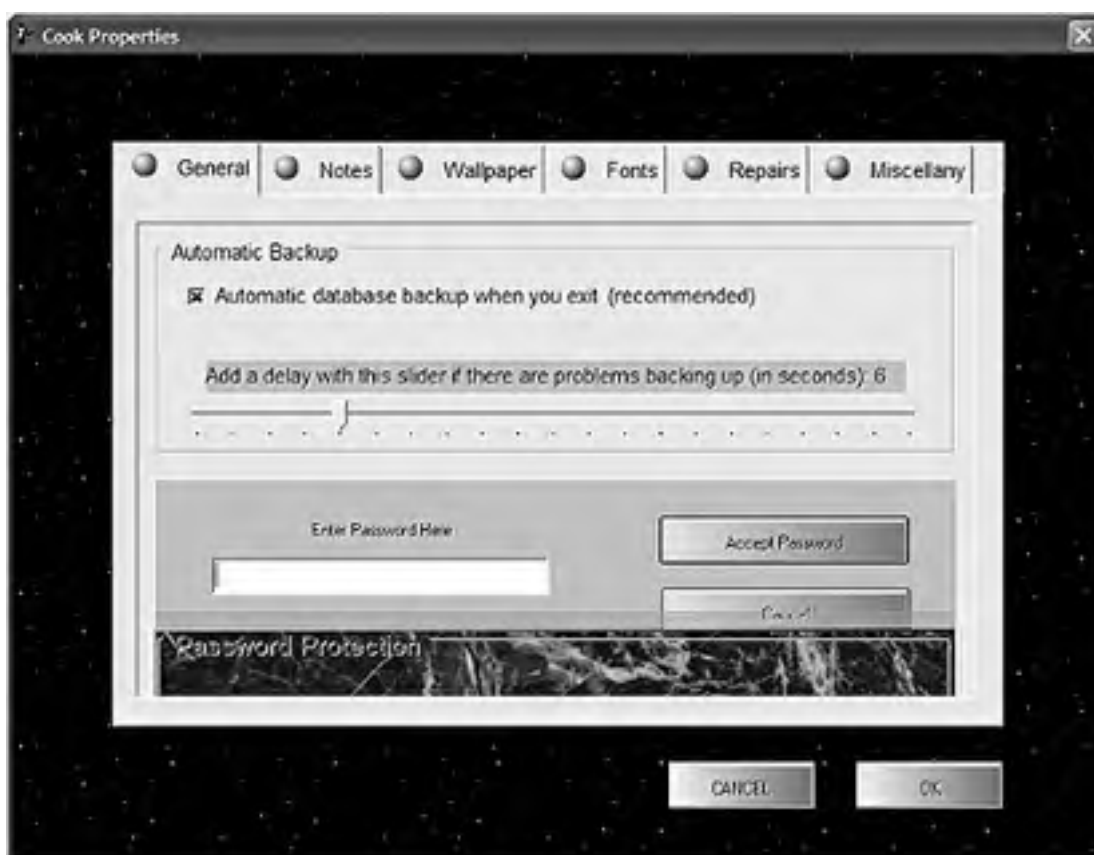


FIGURE 21.12 This marble panel is sliding down to reveal a password entry field.

Sliding simply involves using a timer to continuously adjust a form's or control's Left or Top property. Put a button and panel on a form, then type in the simple code in Listing 21.8.

LISTING 21.8: USING SLIDE TRANSITIONS

```
Dim X As Integer 'remember correct position of Panel1

Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    X = Panel1.Left
    Panel1.Left = -200 'move offscreen
```

```
Timer1.Interval = 1  
End Sub
```

[Team Fly](#)

 Previous

Next 

This page intentionally left blank.

When the design window comes up, there's a highly abbreviated form, probably the smallest one you've ever seen, as shown in Figure 22.2.

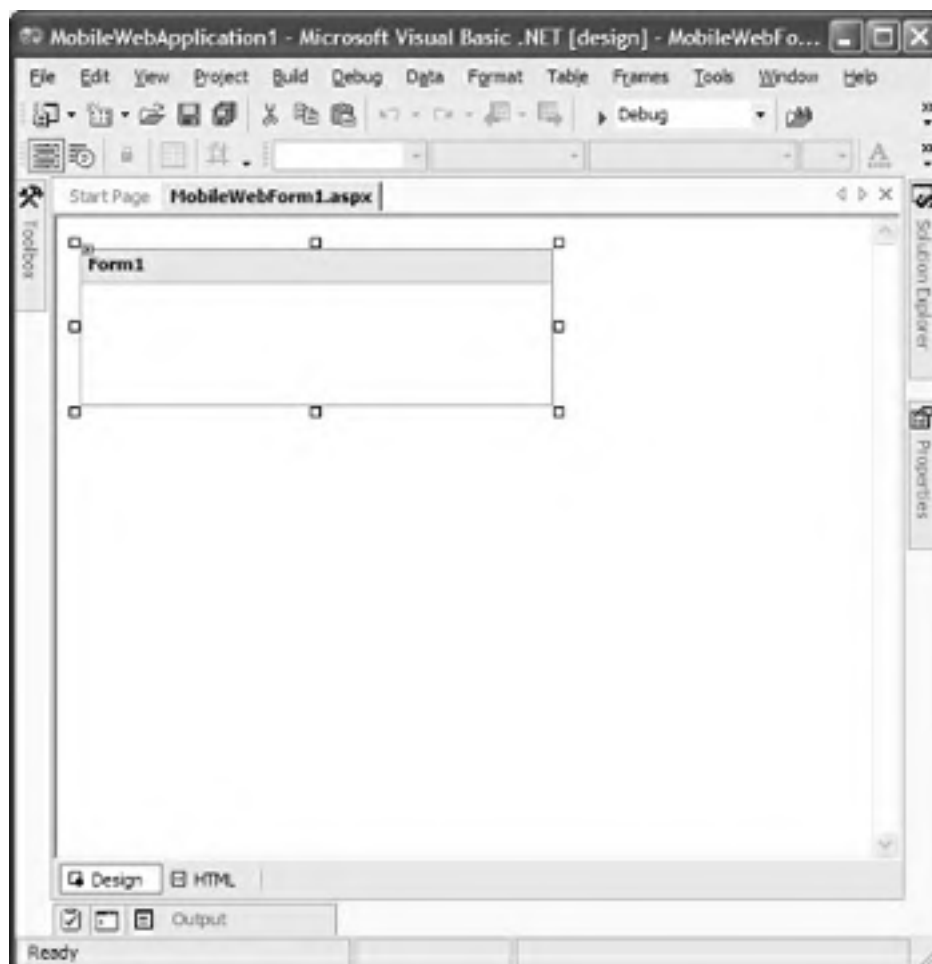


FIGURE 22.2 This tiny form is the space within which you interact with the user.

Understanding the Mobile Form

You get a single form, and most of your applications are likely to employ only one. You can add as many controls as you wish to this one form—and the form will grow taller as necessary to accommodate them. Most mobile devices permit scrolling, so if your form is too large to fit within the visible screen area, the user can simply page down or scroll down to view the rest of it.

You'll find that once you add enough controls to exceed the default form height (generally two controls are the limit), however, when you double-click to add additional controls, they land on the page background, and have to be dragged onto the form.

The CF form isn't necessarily equivalent to a single visible screen on the target device. A CF form is a way to group related controls that may or may not appear on a single page, or may


require additional paging or scrolling to be viewed. In fact, you can specify that you want your form to paginate its output, to ensure that the output doesn't exceed the capacity of whatever device accepts it. The TextView (read-only text display) and various List controls can all grow huge. You may find, however, that some of your application's content is too large for the target device to handle (this can result in an error message from the device during initialization of your page). Try setting the Paginate of the form to True.

Navigating to a Second Form

If instead of pagination you want to subdivide your application into various forms, you can link them simply by adjusting the NavigateURL property of a Link control.

[Team Fly](#)

 Previous

Next 

New Technology, New Behaviors

As you might expect with a new technology, CF has kinks and strange behaviors. As mentioned, double-clicked controls in the Toolbox can land on the design page background, not a form on that page. Then you must drag and drop them onto the form. The Button control is called a Command for some reason. Instead of a Name property, controls have an ID property. Although the ID defaults to Command1, Command2 ... as you'd expect, the default text is Label or Command rather than Label1 or Command1. You use the syntax *Form1.BackColor* rather than *Mc.BackColor*. Deleting a control from a form does not remove any events or source code associated with that control in the code-behind file.

To add controls to the form, you sometimes must double-click them within the Toolbox. Selecting and dragging sometimes won't work, sometimes will. Try adding a TextBox and you'll notice that it looks different from an ordinary TextBox—there's a small input section within a larger box.

Look at the Properties window: There's no Name property, but there is a Title property. This property, and other features of some properties and controls (the calendar, PhoneCall, and so on), derive from an initiative that came out of the Unix camp, their WAP platform, and WML (WebSite Meta Language, another XML derivative). Devices running on WML expect a Title attribute.

You can add additional forms from the Toolbox to this design page, which facilitates creating multi-page applications. However, all the forms look the same as the default form—and the size cannot be adjusted. Nor can the size of many controls, such as the SelectionList, be adjusted. Nor can you reposition the controls horizontally; they go where they go and nowhere else. Nor can the forms be moved around on the design window (it wouldn't have an effect on the target device's display screen anyway).

Just as with ASP.NET applications, there's a code-behind view (see Listing 22.1), which you can access by double-clicking a form. You then see this default code, though the Region is, as usual, hidden unless you expose it.

LISTING 22.1: DEFAULT CODE-BEHIND PROGRAMMING

```
Public Class MobileWebForm1
    Inherits System.Web.UI.MobileControls.MobilePage

    #Region "' Web Form Designer Generated Code '"

    'This call is required by the Web Form Designer.
    <System.Diagnostics.DebuggerStepThrough()> Private Sub Initialize

    End Sub
    Protected WithEvents Form2 As System.Web.UI.MobileControls.Form
    Protected WithEvents Form1 As System.Web.UI.MobileControls.Form

    Private Sub Page_Init(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles MyBase.Init
```

'CODEGEN: This method call is required by the Web Form Design
'Do not modify it using the code editor.'

[Team Ely](#)

 Previous

Next 

```
        InitializeComponent()

    End Sub

#End Region

    Private Sub Page_Load(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles MyBase.Load
        'Put user code to initialize the page here
    End Sub

    Private Sub Form1_Activate(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles Form1.Activate

    End Sub
End Class
```

Note that both `Page_Load` and `Form1_Activate` events are, by default, displayed. There is, however, no `Form_Load` event (though one does exist in the list of available Mobile Web application events for the Form object).

The `Activate` event is similar to the traditional `Load` event. When a page is requested the first time, its first form is activated. However, the `Activate` doesn't trigger on postback. Two other actions trigger `Activate`: when a user uses a link control to navigate to the form, or when the `ActiveForm` property of a page is set in the source code. Use the `Activate` for initialization code, including data-binding to controls on the form and for setting properties such as `BackColor`.

To get a feel for how all this works, try this small program (Listing 22.2). Add a `SelectionList` control to `Form1`, then double-click the Form to get to its code-behind page and type this in:

LISTING 22.2: SEEING A MOBILE APPLICATION IN ACTION

```
Imports System.Web.UI.MobileControls

    Private Sub Page_Load(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles MyBase.Load
        'Put user code to initialize the page here
    End Sub

    Private Sub Form1_Activate(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles Form1.Activate

        Form1.Alignment = Alignment.Center

        Form1.ForeColor = System.Drawing.Color.CadetBlue
        Form1.BackColor = System.Drawing.Color.Bisque
```

```
SelectionList1.Items.Add(' 'ratk")
SelectionList1.Items.Add("atk")
SelectionList1.Items.Add(" tk")
SelectionList1.Items.Add("kratch")

If Not IsPostBack Then
    Form1.Paginate = True
End If
End Sub
```

This Alignment property centers the form within the mobile device's screen (and it's a property from the System.Web.UI.MobileControls namespace, hence the Imports statement. When you test this project, you'll see the controls centered in the browser, but if you want to get a real feel for how it all looks on a cell phone or PDA, reduce your browser's size as shown in Figure 22.3:



FIGURE 22.3 Simulate a mobile device during testing by resizing your browser until it's very, very small, like this.

Further work with the various controls will illustrate for you how the programming for mobile devices is similar to, but just not the same as, regular programming—even regular ASP.NET programming, which it most closely resembles.

Given the limitations of mobile devices, it's not surprising that Microsoft has designed mobile controls—such as the ObjectList—to conserve as much space as possible.

The List Controls

The CF offers your choice of List controls: small, medium, and large. For short lists, use the SelectionList control instead because it doesn't have any provision for pagination. You're encouraged to add fewer than ten items to this list, to permit some cell phone users to choose an item from their keypads (a feature available on some phones). You can also use this list if you want to permit the user to make more than one selection at a time. The SelectionList is a dropdown-style list. The List control is similar to the SelectionList, but all its items are visible (no dropdown); the List control can work with device filters to customize its behavior, and can also paginate. With both the SelectionList and List you can add items during design time by

opening a Properties dialog box (click the Items collection in the Properties window), as shown in Figure 22.4:

[Team Fly](#)

 Previous

Next 

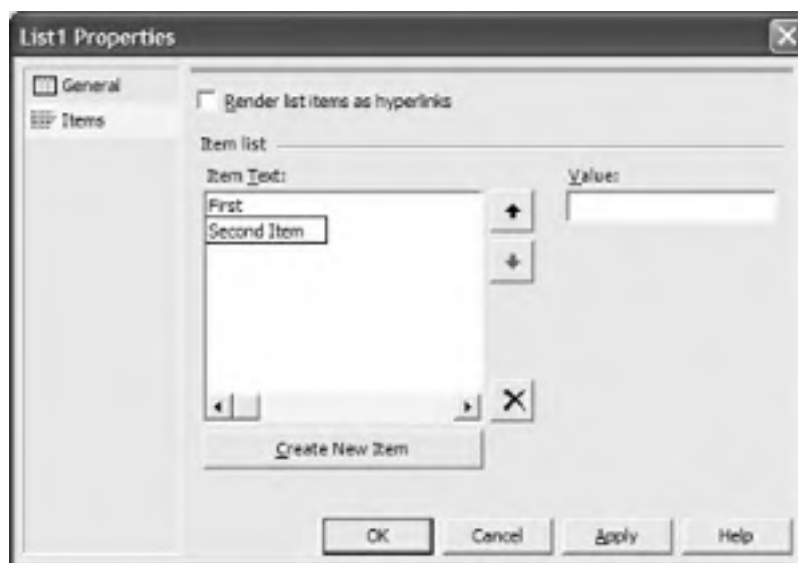


FIGURE 22.4 Use this special editor to create lists.

The ObjectList is the newest, most interesting and most flexible of the three List objects. You can't add items to this list during design time; instead, you must add them programmatically via data binding to a collection such as an array. The ObjectList can use device filters. Listing 22.3 is code that binds an array to a SelectionList during runtime.

LISTING 22.3: BINDING AN ARRAY TO A SELECTIONLIST

```
Private Sub Page_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles MyBase.Load  
  
    If Not IsPostBack Then  
  
        Dim arr(4) As String  
        arr(0) = "'Que Sera Sera, Doris Day"  
        arr(1) = "Seems Like Teen Spirit, Kurt Cobain"  
        arr(2) = "Why Wait for Death?, Keith Richards"  
        arr(3) = "Fever, Peggy Lee"  
        arr(4) = "Let Go of It, Please, Dontatella Versace"  
  
        SelectionList1.DataSource = arr  
        SelectionList1.DataBind()  
  
    End If  
End Sub
```

The results of this code are shown in Figure 22.5.

The ObjectList is even capable of displaying tables, but that might be overkill on the small screens of mobile devices.

Try changing the authorization section in the `Web.config` file in your mobile application to see the effect of a login request. Remove this line:

```
<allow users=''*' /> <!-- Allow all users -->
```

and replace it with this:

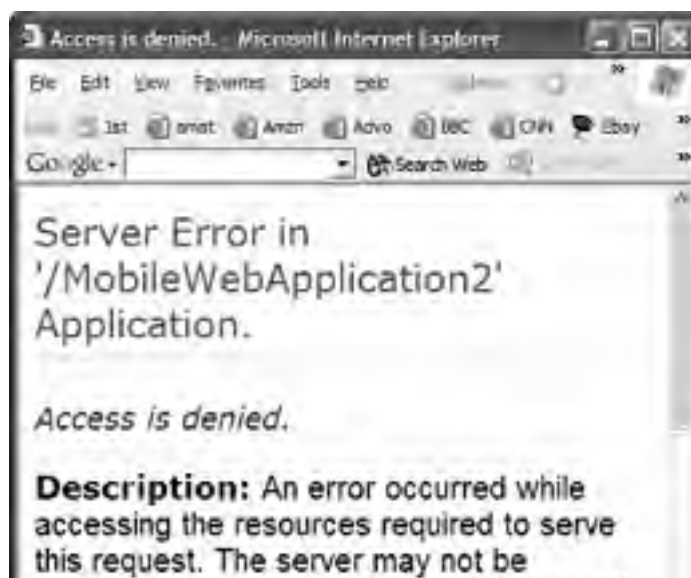
```
<authorization>  
    <allow users="richardm@baxmill.com" />  
    <deny users="*" />  
</authorization>
```

When you press F5 to test your mobile application, you'll be challenged with the logon dialog box shown in Figure 22.6.



FIGURE 22.6 This logon is required when you beef up security for a mobile application.

If you fail to provide the expected logon/password pair, you are denied access to the Web page and the error message shown in Figure 22.7 appears.



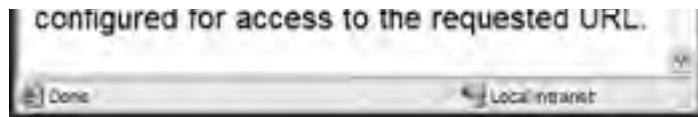


FIGURE 22.7 An error message informs a user that their attempt to log on failed.

[Team Fly](#)

◀ Previous

Next ▶

You can also enable tracing for the current page in a different way. Switch to design view (click the `MobileWebForm1.aspx` tab at the top of the code window). In the Properties window, double-click the `Trace` property to turn it to `True`. There's also a `TraceMode` property, but you should normally leave it to the default `SortByTime`. This shows you the steps in the order that they were carried out by ASP.NET in the process of executing your source code, then building the resulting HTML code to send to the browser. The alternative setting for `TraceMode` is `SortByCategory` and it comes in handy if you want to avoid having your trace messages (your custom `Trace.Write` messages, described shortly) segregated from the other, default trace messages inserted by .NET.

Other options for tracing include *enabled*, which can be selectively turned off, allowing you to skip trace output for pages in your application. Use `Trace=' 'false"` within the page element of the HTML:

```
<%@ Page Language="vb" Trace="false"  
  AutoEventWireup="false" Codebehind="MobileWebForm1.aspx.vb"  
  Inherits="MobileWebApplication2.MobileWebForm1" %>
```

The `RequestLimit` attribute (in the `Web.config` file) governs how many requests that have been made to your application are stored, and it defaults to 10. You can select from among these previous requests and view them. The `LocalOnly` attribute defaults to `True` and specifies whether tracing is enabled for localhost users only (or, if set to `False`, is available for all users).

Custom Tracing

Tracing can also be selective—you can insert your own tracing messages within your code, and they will appear within the trace data, sorted by execution time. For example, if you add these lines to the Page Load event:

```
Private Sub Page_Load(ByVal sender As System.Object, _  
  ByVal e As System.EventArgs) Handles MyBase.Load  
    Dim s As String = "This string"  
    Trace.Write("Beginning Page Load Code HERE. The value of s: " & s)  
End Sub
```

the "Trace information" section of the trace output will include your line, and any variable information you requested, as shown in this result:

```
Trace Information  
Category Message From First(s) From Last(s)  
aspx.page Begin Init  
aspx.page End Init 0.000955 0.000955  
  Beginning Page Load Code HERE. The value of s: This string 0.001978 0.001022  
aspx.page Begin PreRender 0.002233 0.000255  
aspx.page End PreRender 0.002361 0.000129
```

```
Set customErrors mode='On' or "RemoteOnly" to enable custom error me
"Off" to disable.
Add <error> tags for each of the errors you want to handle.
"On" Always display custom (friendly) messages.
"Off" Always display detailed ASP.NET error information.
"RemoteOnly" Display custom (friendly) messages only to users not run
on the local Web server. This setting is recommended for security pur
so that you do not display application detail information to remote c
-->
<customErrors mode="RemoteOnly" />
```

Change the final line in this element to add the name of your custom error-message handling file, and remove RemoteOnly:

```
<customErrors defaultRedirect="genericerror.aspx" mode= "On" />
```

Only change mode to On temporarily. You do this so you can test your error-message handling, seeing what your users will see on their PDAs and phones. When finished testing, restore it to RemoteOnly, the default, so hackers don't benefit from detailed dumps of data about your system.

Now you have to create the genericerror.aspx file to hold your custom error messages. Use Project ➤ Add New Item, click the Mobile Web Form icon to select it, then type **genericerror.aspx** into the Name field. Switch to the genericerror.aspx file and add a Label to it. Double-click the label to get to the code-behind page and add this line to the Page_Load event:

```
Private Sub Page_Load(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Load
    Label1.Text = "Unfortunately there's been an error. Please call" & _
        "1-800-255-5333 and ask for Tricia--our friendly Happy Helper--" & _
        "to fix this minor problem. " & _
        "For your convenience, Tricia stays up all night!"
End Sub
```

This substitutes your understandable, even comforting, custom message for the terrifying *Look what you've done now! HIT THE DECK!* nature of the typical error message. If you don't divert error messages to your custom file, the user will see a message like the one shown in Figure 22.9

Instead, the user sees your reassuring message shown in Figure 22.10.

To trigger an error that will force VB.NET to display your custom error page, type this into your start form, MobileWebForm1.aspx:

```
Private Sub Page_Load(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Load
    MsgBox("Msg")
End Sub
```

When you start a new mobile application project (by clicking the ASP.NET Mobile Web Application icon in the New Project dialog box), you can see that the default files and dependencies that VB.NET creates for you are quite similar to an ASP.NET project's files and defaults. (The System.Web.UI.Controls reference is replaced by System.Web.UI.MobileControls in the .aspx file, but otherwise things are startlingly the same.)

Another change: In the `Web.config` file you'll see these two additional sections that aren't included in ordinary ASP.NET VB.NET projects:

```
>!-- SPECIFY COOKIELESS DATA DICTIONARY TYPE
    This will cause the dictionary contents to appear in the local request u
querystring.
    This is required for forms authentication to work on cookieless devices.
-->
<mobileControls cookielessDataDictionaryType=
"System.Web.Mobile.CookielessData"/>
<deviceFilters>
  <filter name="isJPhone" compare="Type" argument="J-Phone" />
  <filter name="isHTML32" compare="PreferredRenderingType" argument="html32
  <filter name="isWML11" compare="PreferredRenderingType" argument="wml11"
  <filter name="isCHTML10"
compare="PreferredRenderingType" argument="chtml10" />
  <filter name="isGoAmerica" compare="Browser" argument="Go.Web" />
  <filter name="isMME" compare="Browser"
argument="Microsoft Mobile Explorer" />
  <filter name="isMyPalm" compare="Browser" argument="MyPalm" />
  <filter name="isPocketIE" compare="Browser" argument="Pocket IE" />
  <filter name="isUP3x" compare="Type" argument="Phone.com 3.x Browser" />
  <filter name="isUP4x" compare="Type" argument="Phone.com 4.x Browser" />
  <filter name="isEricssonR380" compare="Type" argument="Ericsson R380" />
  <filter name="isNokia7110" compare="Type" argument="Nokia 7110" />
  <filter name="prefersGIF" compare="PreferredImageMIME"
argument="image/GIF" />
  <filter name="prefersWBMP" compare="PreferredImageMIME"
argument="image/vnd.wap.wbmp" />
  <filter name="supportsColor" compare="IsColor" argument="true" />
  <filter name="supportsCookies" compare="Cookies" argument="true" />
  <filter name="supportsJavaScript" compare="Javascript" argument="true" />
  <filter name="supportsVoiceCalls"
compare="CanInitiateVoiceCall" argument="true" />
</deviceFilters>
```

Memory-scarce, security-conscious mobile devices generally don't support cookies. Authentication, though, is sometimes required in any communication process, and the workaround is to stuff some session information and verification data in the query string.

The DeviceFilters section allows you to customize your applications to vary their behaviors in device-specific ways. You know the situation: If it's Netscape, avoid using DHTML, or if it's Internet Explorer, go ahead and do some animation because it supports DHTML.

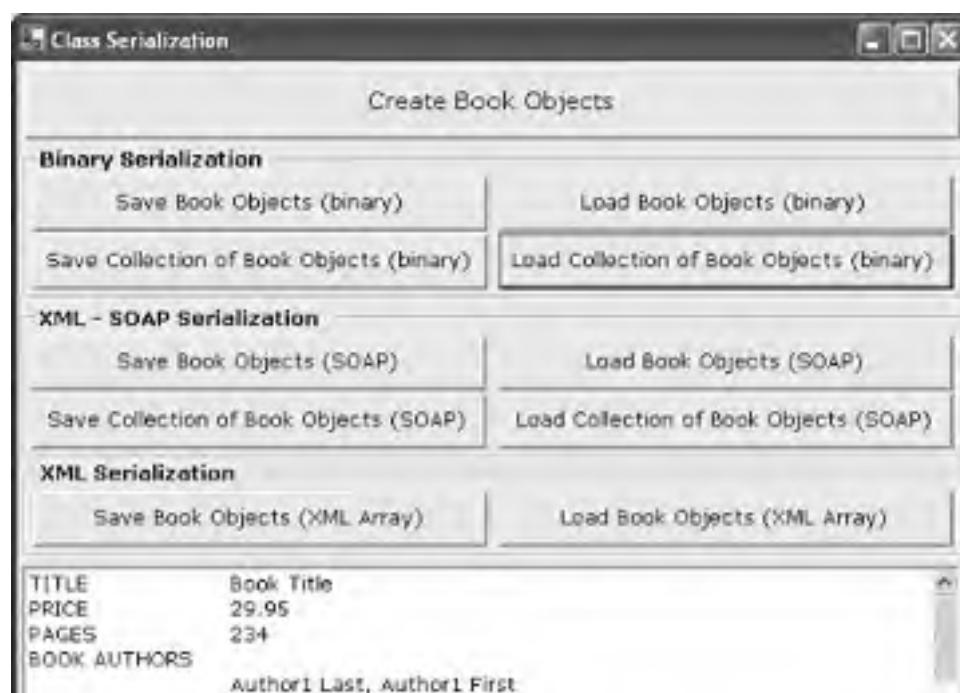
that aren't meaningful outside the current domain should be marked as non-serializable. A typical example is the ID assigned to a record by a database system. Here's a simple class that represents Products. The entire class is serializable, except for the ProductID field, which is assigned a value automatically by the database every time a new product is committed to the database.

```
<Serializable( )> _  
Public Class Product  
    Public Sub New( )  
    End Sub  
    <NonSerializable( )> _  
    Public ProductID As Integer  
    Public ProductName As String  
    Public ProductPrice As Decimal  
End Class
```

You can create an instance of the Product class, serialize it, and then deserialize it into another instance of the same class. While all other fields will be persisted, the ProductID field will not be persisted.

The ClassSerializer Project

The example of this section demonstrates how to serialize a fairly complicated class in binary, SOAP, and XML format (you can ignore the XML serialization samples for the moment and return to the project after reading about XML serialization). The project's code serializes individual objects as well as collections of objects. The ClassSerializer project's main form is shown in Figure 3.2. Initially, all buttons are disabled except for the Create Book Objects button. Click this button to create a few Book objects and then serialize/deserialize them with the appropriate buttons.



```
Author2 Last, Author2 First
Author3 Last, Author3 First
BOOK'S ROYALTY SCHEDULE
10000 0.15
15000 0.2
25000 0.25
ROYALTIES FOR 32500 COPIES : 205906.2500
```

FIGURE 3.2 The ClassSerializer project demonstrates the various serialization types in .NET.

Nokia <http://forum.nokia.com>

Openwave <http://developer.openwave.com>

The Visual Studio 2003 Pocket PC Emulator

Shipped with Visual Studio 2003 is emulator version 4.1, which is configurable within the IDE. This emulator is remarkably full of features—you feel as if you're working with a real PDA. The emulator currently emulates Pocket PC devices (and Windows CE), and word is that it will be expanded to include additional devices.

To see if you installed it during VS 2003 setup (it's a default), choose Tools ➤ Connect To Device and you should see the dialog box shown in Figure 22.11.



FIGURE 22.11 Connecting to the Pocket PC Emulator built into Visual Studio 2003.

Select Pocket PC 2002 Emulator (Default) and click the Connect button. You should see the full emulation appear, as shown in Figure 22.12:





FIGURE 22.12 Here's a Pocket PC you can test with, right inside your VS.NET environment.

Also notice that the status bar on the bottom of your VB.NET IDE displays Device Connected.

***TIP** If you cannot follow the above steps, you most likely deselected the Smart Device Programmability option during VS.NET setup. To install it, you must use the Add/Remove Programs feature in Control Panel to Change/Remove your version of VS.NET. But instead of removing it, go to the maintenance page and select Add or Remove Features. On the Options page, choose Smart Device Programmability. Then click Update Now.*

POCKET PC EMULATOR CONFIGURATION

You can't configure the emulator while it's running, but the changes won't take effect until you restart it. So shut it down then choose Tools ➤ Options ➤ Device Tools ➤ Devices. Click the Configure button and you see the Configure Emulator Settings dialog box, where you can adjust color depth, screen size, system memory, and serial and parallel port connections (which can be mapped to your programming computer for testing). Several settings are optimistically flexible. Screen size, for example, can be adjusted from 80×64 up to 1024×768 pixels. A 1024×768 pixel screen four inches square exceeds most people's requirements, not to mention the acuity of their vision.

New Technology, New Problems

As with many new technologies, you have to struggle a bit with workarounds and generally inaccurate, too brief, or beside the point documentation. Using the built-in emulator is no exception. The following problems using the Pocket PC emulator occur at the time of this writing.

You'd think that as soon as you execute the emulator, it would be available as an option, as a testing target device, within the VB.NET IDE. It's not. Although Help says you can change from Internet Explorer to another emulator by choosing File ➤ Browse With, that option isn't available on the File menu. Instead, you must right-click your Form1.aspx in Solution Explorer, then choose Browse With.

When the Browse With dialog box appears, you expect to see your emulators listed in the Browser list. Not so fast, dude. You have to manually add the emulators by locating their executable on your hard drive. Click the Add button in the Browse With dialog box, then click the Browse button. Locate `Program Files\Microsoft Visual Studio .NET 2003\Compact Framework SDK\Connection Manager\Bin\Emulator.exe`. Click it to add it to your list of available emulators.

Click the Set As Default button. This should make it the default for all your .aspx files. You should also right-click your application's name in Solution Explorer (it's the name in boldface). Choose Properties, then in the dialog box click Configuration Properties, Debugging. Deselect "Always use Internet Explorer when debugging Web pages."

Even with all these adjustments, though, you might still face problems. (This is a new technology; they're working on these problems.) When you try to test your mobile application in the emulator, you might get the following error message: "The CE boot image could not be opened. Please verify that the specified path name is valid." Of course, this error message doesn't tell you *how* to verify this or rectify the problem.

This problem (and others) have been reported if you're not on a networked computer. One workaround if you're using a stand-alone desktop machine is to install the Microsoft TCP Loopback Adapter that ships with Windows XP. Another workaround to try: Disable XP's Internet Connection

[Team Fly](#)

 Previous

Next 

This page intentionally left blank.

- AlternatingItemStyle property, of DataGrid control, [281](#)
- alternation, in regular expressions, [563–564](#)
- anchors, in regular expressions, [562](#)
- AND Boolean operator, for DataView object, [426](#)
- animated transitions, [636–641](#)
- API functions
 - importing, [187](#)
 - for screen capture, [186](#)
- Append property, of FileMode class, [43](#)
- Application Directory membership condition, [256](#)
- Application object, [104](#)
 - destroying, [95](#)
- application servers, business logic on, [449](#)
- application tier. *See* middle tier component
- ApplicationException class, class inheritance from, [229–230](#)
- applications. *See also* deployment process
 - adding to User's Programs Menu, [267](#)
 - appearance of reliability, [623–626](#), [624](#)
 - assigning permission set to, [255](#)
 - communicating data between, [93](#)
 - debugging, [216](#)
 - design, [215](#)
 - distributed, [2](#)
 - multiple processing queue messages, [367](#)
 - persisting settings to configuration file, [76–79](#)
 - robust, [215](#). *See also* structured exception handling debugging techniques, [231–240](#)
 - running, [53](#)
 - running after Internet-based deployment, [257–258](#)
 - starting from within another application, [259](#)
 - upgrading, [244](#)
- ArgumentException, of Open method, [219](#), [225](#), [226](#)
- ArgumentNullException, of Open method, [219](#), [226](#)
- ArgumentOutOfRangeException, of Open method, [219](#), [226](#)
- arguments for Response object, entire files as, [272](#)
- Arguments property, of Register User dialog box, [270](#)
- Arial font, [627](#)
- ArithmeticException, [226](#)

- Array class, [34](#)
- ArrayList, [35–37](#)
 - binary serialization of collection, [62–64](#)
 - deserialization, [64](#)
 - mass manipulation, [36](#)
- arrays, [14](#), [28–34](#)
 - binding to SelectionList control, [651](#)
 - control, [24–28](#)
 - creating and filling, [19–20](#)
 - deleting element, [35–36](#)
 - filling with random numbers, [52–53](#)
 - number of elements in, [29](#)
 - search and sort methods, [31–33](#)
 - zero-based collections, [28–29](#)
- ArrayTypeMismatchException, [226](#)
- As New command, [10](#), [45](#)
- .ASMX filename extension, [322](#)
- ASP.NET. *See also* DataGrid control
 - data display on WebForm, [273–283](#)
 - connecting to database, [273–274](#)
 - DataList control, [275](#)
 - detecting postback, [281–282](#)
 - Repeater control, [276](#)
 - templates, [275](#)
 - HTML controls, [287–288](#)
 - new features, [271–273](#)
 - sending entire files, [272](#)
 - server controls, [272–273](#)
 - sending graphics, [286–287](#)
 - validation, [282–285](#)
 - controls, [283–285](#)
 - programmatic, [282–283](#)
- assemblies, [xxii](#), [192–193](#)
 - downloading on demand, [258–259](#)
 - exploring unknown, [208–211](#)
 - loading file from, [200–201](#)
 - loading from file, [201–203](#)
 - need for Imports statement, [11](#)
 - path for accessing, [202](#)
 - reflection to access current project's, [198–199](#)

reflection to access loaded, [199–200](#)
security, [193](#)
asterisk (*), in regular expressions, [543](#), [558](#)
asymmetric public key system, [143](#)
asymmetrical encrypting, [151–158](#)
asynchronous system, [341](#)
attractors, [605](#)
attribute classes, [124](#)
AttributeCount property, of XMLReader object, [477](#)

- COM components, use with .NET clients, [461–467](#)
 - ActiveX controls, [462–465](#)
- COM wrapper, [94](#)
- ComboBox control, binding to columns, [506](#)
- COMCalculator, [462–463](#), [463](#)
- Command class (ADO.NET), [391–392](#), [409–415](#)
 - parameters, [410–413](#)
- Command window for debugging, [239–240](#)
- CommandText property, of Command class, [409](#)
- CommandType enumeration, [409](#)
- Commit method, of MessageQueueTransaction class, [371–372](#)
- Common Language Infrastructure (CLI), [214](#)
- Common Language Runtime (CLR)
 - evidence for, [251](#)
 - installing, [244](#)
 - and .NET applications, [462](#)
- communicating data between applications, [93](#)
- communication, programming as, [xx](#)
- Compact Framework. *See* .NET Compact Framework
- CompareValidator control, [283](#)
- comparisons in DataAdapter, null values and, [406](#)
- compatibility namespace, accessing, [201–202](#)
- Compiled member of RegexOptions enumeration, [549](#)
- complex numbers, [604](#)
 - operations, [620–621](#)
 - in transformation, [605–606](#)
- COMPlus component, [466–467](#)
- Component Services Explorer, [465](#)
- ComponentModel.InvalidEnumArgumentException, [226](#)
- ComputeHash method, [141](#), [151](#)
 - streaming file into, [142](#)
- computer Properties dialog box, Network Identification tab, [290](#)
- concurrency, [515](#)
 - DataAdapter handling of, [405–406](#), [407–408](#)
 - optimistic, [516](#)
- Condition property, for debugging, [237](#)
- config files

- for code-access security, [127](#)
- persisting application settings to, [76–79](#)
- Configure Emulator Settings dialog box, [662](#)
- Connect method, of chat client object, [306](#)
- connected application, NWProducts application as, [508](#)
- Connected event, code to handle, [304](#)
- Connection class (ADO.NET), [391–392](#), [402–403](#)
 - BeginTransaction method of, [436](#)
- ConnectionString property, of Connection object, [402](#)
- Console.WriteLine, to display variable, [17](#)
- constraints
 - in database testing, [439](#)
 - editing DataSet with, [421](#)
- constructors, [10–11](#)
 - reflection to report on, [196–197](#)
 - for Xml Serializer class, [72](#)
- content attribute, of XML elements, [486](#)
- ContinueUpdateOnError property, of DataAdapter, [408](#)
- control arrays, [24–28](#)
- controls
 - adding to form at runtime, [25–27](#)
 - data-bound, [506](#)
 - inherited, appearance of, [176](#)
- cookies, for mobile computing, [652](#), [659](#)
- coordinates, for printing Graphics object, [168](#)
- Copies property, of PrinterSettings object, [163](#)
- counting messages in queue, [358](#)
- crashes, [216](#), [217](#)
- Create Code Group wizard, [255–257](#)
- Create method, [5–6](#), [6](#)
 - of MessageQueue class, [347–348](#)
- Create Permission Set dialog box, [130](#), [254](#), [254](#)
- Create Permission Wizard, [254](#)
- CreateComInstanceFrom method, [211](#)
- CreatedAfter property, of MessageQueueCriteria class, [349](#)
- CreatedBefore property, of MessageQueueCriteria class, [349](#)
- CreateDirectory method, of DirectoryInfo class, [9](#)
- CreateInstance method, of Activator class, [211](#)
- CreateMeasurementGraphics property, of PrinterSettings object, [164](#)
- CreateScreenShot function, [187–188](#)

cryptology, [xx](#)

CryptoServiceProvider object, [145](#)

cryptostream, for DES, [145](#)–146

CultureInvariant member of RegexOptions enumeration, [549](#)

Custom Actions editor, [262](#)

custom class, for moving data out of SQL Server, [81](#)–[90](#)

custom exceptions, throwing, [228](#)–[230](#)

[Team Fly](#)

 Previous

Next 

The Book class provides some simple properties: the Title, Pages, and a few more properties that describe a book. The Authors property is an array of Author objects, and the Author class exposes the FirstName and LastName properties. Finally, the Book class exposes the AuthorRoyalties property, which describes how the book's royalties are calculated. The RoyaltySchedule class exposes a single property, which is an array of RoyaltyTier objects. Each RoyaltyTier object exposes the Tier and Royalty properties, which are the tiers (number of copies) and the corresponding royalties.

For a certain book, the royalties paid to the author(s) are calculated as follows: 15% of the book's price for the first 10,000 copies of the book, 20% for the next 5,000 copies and 25% for any number of copies after that. The last tier's value is actually irrelevant, because there's no higher tier. The following statements create a royalty schedule for a book by setting up an array of RoyaltyTier objects (*BK0* is an instance of the Book class):

```
Dim auRoyaltySchedule(2) As RoyaltySchedule.RoyaltyTier
auRoyaltySchedule(0) = New RoyaltySchedule.RoyaltyTier()
auRoyaltySchedule(0).Tier = 10000
auRoyaltySchedule(0).Royalty = 0.15

auRoyaltySchedule(1) = New RoyaltySchedule.RoyaltyTier()
auRoyaltySchedule(1).Tier = 15000
auRoyaltySchedule(1).Royalty = 0.2

auRoyaltySchedule(2) = New RoyaltySchedule.RoyaltyTier()
auRoyaltySchedule(2).Tier = 25000
auRoyaltySchedule(2).Royalty = 0.25

BK0.AuthorRoyalties = New RoyaltySchedule()
BK0.AuthorRoyalties.Tiers = auRoyaltySchedule
```

The complete code of the Book class is shown in Listing 3.3. It's not a trivial class, and you'll find it easier to understand if you create and configure a Book object. Besides its properties, the Book class exposes the CalculateRoyalties method, which returns the royalties generated by a specific number of copies of the current book. We'll use this method to verify that the deserialized objects match the serialized ones (they should both report the same royalties). To condense the listing, we're not showing the implementation of the trivial properties, but you can find them in the project's code.

LISTING 3.3: THE BOOK CLASS

```
Imports System.Runtime.Serialization
Imports System.Xml.Serialization

<Serializable(>> Public Class Book
    Private _title As String
    Private _pages As Integer
    Private _price As Decimal
    Private _authors() As Author
    Private _authorRoyalties As RoyaltySchedule

    Public Sub New()
```

- DataType property, of XmlElement, [74](#)
- DataView objects, [426–427](#)
- DataViewRowsStates enumeration, [427](#)
- Date function, [38](#)
- DateTime class, methods, [38](#), [39](#)
- DateTime data type, [39](#)
- DbType property, of Parameter object, [410](#)
- dead-letter messages, [343](#)
- Debug mode, [232](#)
- debugging, [216](#), [231–240](#)
 - breaking on all errors, [230–231](#)
 - Call Stack window, [240](#), [241](#)
 - with debugger windows, [238–239](#), [239](#)
 - logical errors, [231](#), [237–240](#)
 - breakpoints, [237–238](#)
 - mobile application, [654–656](#)
 - Output and Command windows for, [239–240](#)
 - with WriteLine method, [240](#)
- decimal value type, [14](#)
- declarative code access, [124](#)
- Decrypt procedure, [151](#)
- default startup object, in Visual Basic, [4](#)
- default unit of page, [168](#)
- DefaultPageSettings property
 - of PrintDocument object, [165](#)
 - of PrinterSettings object, [163](#)
- DefaultPropertiesToSend property, of MessageQueue class, [351](#)
- DefaultView property, of DataTable object, [426](#)
- Definitions element in WSDL, [331](#)
- DELETE statement (SQL), DataAdapter task for, [404](#)
- DeleteCommand property, [405](#)
 - of DataAdapter, [404](#)
- DeleteRule property, [431](#)
- deleting
 - array element, [35–36](#)
 - attributes in XML, [488](#)
 - elements in XML, [488–489](#)

- files, security and, [145](#)
- message queues, [345](#)
- messages in queue, [357](#)
- rows in DataTable, [419](#)
- Demand method, [257](#)
- DemoControl project, [175](#)–178
- dependencies of project, [264](#)–265
- deployment process, [243](#)
 - Internet-based, [244](#), [246](#)–[259](#)
 - assembly download on demand, [258](#)–[259](#)
 - code access permissions, [251](#)–[257](#)
 - preparation, [247](#)–[249](#)
 - running application, [257](#)–[258](#)
 - Windows application deployment on Web server, [249](#)–[250](#)
 - with Windows installer, [259](#)–[270](#), [260](#)
 - File System Editor, [263](#)–[266](#)
 - installer package creation, [261](#)–[262](#)
 - Registry Editor, [267](#)
 - shortcut creation, [266](#)–[267](#)
 - User Interface Editor, [267](#)–[270](#)
 - Windows installer package, [244](#)
 - XCopy method, [245](#), [245](#)–[246](#)
- depth, as user interface convention, [627](#)–628
- Depth property, of XMLReader object, [477](#)
- DES (Data Encryption Standard), [142](#), [143](#)
 - encrypting and decrypting file with, [143](#)–145
- Description property, of Setup Project, [266](#)
- descriptors, in code-access security, [127](#)
- deserialization, [56](#)–[57](#), [59](#)
 - overhead, [79](#)
 - source of process, [80](#)
- Deserialize method, [64](#)
- destructive concurrency, [407](#)–408
- DetectnewInstalledVersion property, of Setup Project, [266](#)
- development machines, [243](#)
- DHCP (Dynamic Host Configuration Protocol), [290](#)
- dialog boxes
 - adding to user interface, [268](#)
 - for printing, [164](#)–[167](#)
- digital signatures, [121](#)

Dim statement, [8](#)
dimension, vs. capacity, [28](#)
Direction property, of Parameter object, [410](#)
directories, adding items to user's, [267](#)
Directory object, [12](#)
 for getting file list, [11](#)
DirectoryInfo class, [9](#)
 CreateDirectory method of, [9](#)
 GetDirectories method of, [11](#)
DirectoryNotFoundException, of Open method, [220](#)
DirectoryServicesPermission code access permission, [252](#)

- message retrieval from queue, [378–379](#)
- order preparation, [375–377](#)
- referencing, [345–346](#)
- triggers, [382–388](#)
 - defining, [384–385](#)
 - ProcessOrders console application, [385–388](#)
 - rules, [382–384](#), [383](#)
 - types, [343–344](#)
- Message Queuing Triggering service, [382](#)
- MessageBox statement
 - for debugging, [240](#)
 - to display variable, [17](#)
- MessageEnumerator class, [355](#)
- MessageQueue class, [347–349](#)
- MessageQueueCriteria class, [348–349](#)
- MessageQueueEnumerator class, [349](#)
- MessageQueuePermission code access permission, [252](#)
- MessageQueueTransaction class, [371–372](#)
- MessageReadPropertyFilter property, of MessageQueue class, [361](#), [364](#)
- messages. *See also* error messages
 - in asynchronous communications
 - creating and sending, [352–358](#)
 - sending and receiving, [346](#)
 - settings to save, [344](#)
 - suppressing, [100](#)
- MessageType property, of acknowledgment messages, [361](#)
- metacharacters in regular expressions, [544](#), [558](#)
 - escaping, [563](#)
 - for lookahead and lookbehind assertions, [578](#)
 - quantifiers for, [546](#)
 - replacement, [553](#)
 - single character, [558–559](#)
 - white space and, [560](#)
- metadata, [16](#)
 - in schemas, [482](#)
- metallic shading in user interface, [635–636](#), [636](#)

- methods, [18](#). *See also* constructors
 - access to specific, [206–207](#)
 - reflection to report on, [197–198](#), [203–204](#)
 - shared, [8](#)
- Microsoft, and DOM specification, [478](#)
- Microsoft biztalk, [481](#)
- Microsoft Excel 10.0 Object Library, [111](#)
- Microsoft Intermediate language (MSIL), [214](#)
- Microsoft Message Queueing (MSMQ) component, [341](#)
 - fault tolerance and load balancing, [366–370](#)
 - management console snap-in, [344](#), [344–345](#)
 - message delivery guarantee, [343](#)
- Microsoft Office. *See* Excel; Outlook; Word
- Microsoft Outlook 10.0 Object Library, [109](#)
- Microsoft Pocket PC, emulator, [660](#)
- Microsoft SQL Server 2000 Desktop Engine (MSDE), potential problems, [329](#)
- Microsoft Word 10.0 Object Library, [94](#)
- Microsoft.VisualBasic namespace, [52](#)
- Microsoft.Win32 namespace, [49](#)
- middle tier components, [401](#)
 - advantages, [444](#)
 - business logic, [442](#), [443](#)
 - remoting, [449–461](#)
 - for business rule, [532–535](#)
 - changing reference to, [457](#)
 - from client/server to, [441–449](#)
 - business rules, [443–445](#)
 - COM+ applications, [465–473](#)
 - COMPlus component, [466–467](#)
 - exporting proxy, [467–468](#)
 - serviced components, [468–473](#)
 - converting BusinessLayer class to Web service, [450–457](#)
 - for database connection, [90](#)
 - designing with, [445–449](#)
 - discount policy component as, [447–449](#)
 - using COM components with .NET clients, [461–467](#)
 - as Web service, [247](#)
- MinimumPage property, of PrinterSettings object, [164](#)
- MinorGridWidth property, in PlotControl application, [592](#)
- MinorXTicks property, in PlotControl application, [592](#)

MinorYTicks property, in PlotControl application, [592](#)
mobile code, [125](#)
mobile computing, [643](#)
 connections, [644–645](#)
 debugging via tracing, [654–656](#)
 device specificity, [658–660](#)
 emulators, [660–663](#)
 security, [652–653](#)

```
End Sub

Public Property Title() As String
    Get . . .
    Set . . .
End Property

Public Property Pages() As Integer
    Get . . .
    Set . . .
End Property

Public Property Price() As Decimal
    Get . . .
    Set . . .
End Property

Public Property Authors() As Author()
    Get . . .
    Set . . .
End Property

Public Property AuthorRoyalties() As RoyaltySchedule
    Get
        Return (_AuthorRoyalties)
    End Get
    Set(ByVal Value As RoyaltySchedule)
        _AuthorRoyalties = Value
    End Set
End Property

Public Function CalculateRoyalties(ByVal UnitsSold As Integer) As Dec
    If _AuthorRoyalties Is Nothing Then Return 0
    Dim tier As Integer, lastTier As Integer
    lastTier = _AuthorRoyalties.Tiers.GetUpperBound(0)
    Dim currentTier As Integer = 0
    Dim runningCount As Integer = 0
    Dim tierCount As Integer = 0
    Dim royalty As Decimal = 0
    For tier = 0 To lastTier
        tierCount = _AuthorRoyalties.Tiers(tier).Tier
        If tier > 0 Then tierCount = _
            tierCount - _AuthorRoyalties.Tiers(tier - 1).Tier
        runningCount = Math.Min(UnitsSold, tierCount)
        If UnitsSold > 0 Then royalty = royalty + _
            _AuthorRoyalties.Tiers(tier).Royalty *
            _price * tierCount
        UnitsSold = UnitsSold - runningCount
    
```


- positive lookahead, [575](#)
- PositiveArrival member of AcknowledgeTypes
 - enumeration, [359](#)
- postback for DataGrid, detecting, [281–282](#)
- PostiveReceive member of AcknowledgeTypes
 - enumeration, [359](#)
- Precision property, of Parameter object, [410](#)
- Prefix property, of XMLReader object, [478](#)
- presentation tier, [442](#)
 - separation by middle tier, [446](#)
- primary key changes, propagating, [431](#)
- principal object, [122–123](#)
- Print dialog box, [164–166](#), [165](#)
- Print method, of PrintDocument object, [166](#)
- Print Preview dialog box, [164](#)
- PrintBMP subroutine, [188–189](#)
- PrintDialog control, [165–166](#)
- PrintDocument control, [159–161](#)
- Printer object, [159](#)
- PrinterName property, of PrinterSettings object, [164](#)
- PrinterResolution property, of PageSettings object, [163](#)
- PrinterResolutions property, of PrinterSettings object, [164](#)
- PrinterSettings object, [162](#), [163–164](#)
- PrinterSettings property, of PageSettings object, [163](#)
- printing
 - dialog boxes, [164–167](#)
 - in .NET, [159–162](#)
 - simple printout, [160–161](#)
 - page layout, [168–174](#)
 - DrawString method, [168–170](#)
 - PrintTests project, [170](#), [170–174](#)
 - plaintext, [174–178](#), [175](#)
 - pretty, from Excel, [113](#)
 - printer and page properties, [162–164](#)
 - PageSettings object, [162–163](#)
 - PrinterSettings object, [163–164](#)

- printscreen utility, [186](#), [186–189](#)
- punctuation symbols, [178](#)
- tabular data, [179–186](#)
 - with Word objects, [107–108](#)
- PrintingPermission code access permission, [252](#), [253](#)
- PrintOut method, [113](#)
- PrintPage event handler, [160](#)
- PrintPreview control, [166–167](#), [167](#)
- PrintRange property, of PrinterSettings object, [164](#)
- PrintScreen project, [186](#), [186–189](#)
- PrintTests project, [170](#), [170–174](#)
- Priority property, of Message class, [342](#), [347](#), [350](#)
- prisoners, in Mandelbrot Set, [608](#), [612](#)
- privacy, [139](#)
- private queues, [343](#)
 - referencing, [345](#)
- privileges, for .NET applications downloaded from server, [250](#)
- PRNListView control, [xxii](#), [179–186](#)
 - generating printout, [181–185](#)
 - initiating printout, [180–181](#)
- PRNTextBox control, [xxii](#)
- ProcessOrders console application, [385–388](#)
- production machines, [243](#)
- programming
 - aptitude for, [xix–xx](#)
 - error types, [232–236](#)
 - for security, [133–136](#)
- Project Gutenberg web site, [545](#)
- Project menu
 - Add Module, [46](#)
 - Add Reference, [11](#), [63](#), [109](#)
- project output, [263](#)
- projects, [193](#)
 - adding Web service to, [336](#)
 - BasicSerialization project, [61–65](#), [62](#)
 - BusinessLayer project, [450–457](#)
 - ClassSerializer project, [66](#), [66–71](#)
 - Book class, [67–70](#)
 - deserializing individual objects, [71](#)
 - serializing individual objects, [70–71](#)

COMCalculator, [462–463](#), [463](#)
componentizing, [444](#)
DataRelations project, [422–424](#), [423](#)
DemoControl project, [175–178](#)
DisconnectedOrders application, [374](#), [374–375](#)
DiscountServer project, [458–461](#)
fractal generator, [602–620](#)
invoicing application, [516–535](#)
 architecture, [518–525](#)
 interface, [516–518](#), [518](#)
MatchEvaluator project, [554–557](#), [555](#)
MSMQLoadBalancing project, [367–370](#)
 BalancedQueue setup, [368](#)

- random numbers, [49–53](#)
 - filling array with, [52–53](#)
- Randomize function, [49](#)
- Random.NextBytes method, [52](#)
- ranges of characters, in regular expressions, [546–547](#)
- RangeValidator control, [283](#), [284](#)
- RankException, [227](#)
- RCW (runtime-callable wrapper), [462](#)
- Read method, of FileStream object, [221](#)
- read-only access to database data, [413](#)
- read-only files, exception handling for, [222](#)
- reading files, [40–41](#)
- reading messages, from queue, [354–355](#)
- ReadLine method, of Stream object, [311](#)
- ReadToEnd method, of StreamReader object, [42](#)
- ReadWriteFile project, [219–224](#)
- real numbers, [603–604](#)
- Receive method
 - for messages in queues, [347](#), [354](#)
 - of socket, [295](#)
- ReceiveByCorrelationID method, [350](#), [362](#)
- ReceiveByID method, [358](#)
- ReceiveCompleted event, [357](#)
- Recoverable property, of Message class, [344](#), [351](#), [366](#)
- Reference Map, [336](#)
- reference types, [14](#)
- ReferenceEquals method, [13](#), [14](#)
- Reference.vb file, [336](#)
- referential integrity, [421](#)
- reflection, [xxii](#)
 - accessing a type, [194–198](#)
 - accessing current project's assembly, [198–199](#)
 - accessing loaded assembly, [199–200](#)
 - accessing specific members, [206–207](#)
 - emission, [213–214](#)
 - to execute discovered code, [208–213](#)

- to get methods in class, [203–204](#)
- searching for members or data, [207–208](#)
- uses for, [191–193](#)
 - assemblies, [192–193](#). *See also* assemblies
 - containers within containers, [193](#)
 - security, [193](#)
 - types and, [191–192](#), [205–206](#)
- Reflection.Emit namespace, [213](#)
- ReflectionPermission code access permission, [252](#)
- Regex, [282–283](#)
- RegExEditor project, [564–567](#)
 - Find & Replace dialog box, [564–567](#)
- RegexOptions enumeration, [549](#)
- Register User dialog box, [270](#)
- Registry, [48–49](#)
 - writing to, [49](#)
- Registry Editor, [262](#), [267](#)
- RegistryKey class, [48](#), [49](#)
- RegistryPermission code access permission, [252](#)
- regsvcs.exe tool, [472](#)
- Regular Expression Editor dialog box, [285](#), [285](#)
- regular expressions, [543](#)
 - advanced topics, [567–579](#)
 - grouping and backreferences, [568–573](#)
 - lookahead and lookbehind assertions, [575–578](#)
 - multiple captures, [573–575](#)
 - replacement operations, [578–579](#)
 - elements, [558–567](#)
 - alternation, [563–564](#)
 - anchors, [562](#)
 - characters and metacharacters, [558](#)
 - escaping metacharacters, [563](#)
 - quantifiers, [560–562](#)
 - ranges of characters, [559–560](#)
 - single character metacharacters, [558–559](#)
 - white space and metacharacters, [560](#)
 - greedy vs. non-greedy, [561–562](#)
 - ranges of characters in, [546–547](#)
- RegExEditor project, [564–567](#)
 - Find & Replace dialog box, [564–567](#)

- RegularExpressions project, [579–582](#)
- testing, [571](#)
- for validation, [282](#)
- Visual grep project, [582](#), [582–587](#)
- writing, [544–547](#)
- RegularExpressions class, [547–557](#)
 - Match and NextMatch methods, [551–552](#)
 - Matches method, [550–551](#)
 - Replace method, [552–557](#)
 - MatchEvaluator project, [554–557](#)
 - Split method, [552](#)
- RegularExpressions project, [544](#), [545](#), [579–582](#)
- RegularExpressionValidator control, [283](#), [285](#)
- RejectChanges method, of DataSet, [420](#), [425–426](#)

- Toolbox for design, [398](#)
- XSD, [481–482](#)
- science, and programming, [xx](#)
- screen display, printscreen utility, [186](#), [186–189](#)
- Script control, [462–463](#)
 - programming, [464–465](#)
- scripting, security and, [125](#)
- searching
 - in arrays, [31–33](#)
 - text, [106–107](#)
- security. *See also* .NET Security Policy management
 - agreement on, [124–125](#)
 - code-access, [124](#), [125–128](#)
 - encrypting, [143–151](#)
 - asymmetrical, [151–158](#)
 - DES (Data Encryption Standard), [143–145](#)
 - hashing with, [147–151](#)
 - initialization vectors for DES, [146–147](#)
 - key length, [147](#)
 - main problem, [139–143](#)
 - .NET Framework features, [121–122](#)
 - permissions for downloaded assembly, [251](#)
 - and privacy, [139](#)
 - programming for, [133–136](#)
 - role-based, [122](#), [469](#)
 - for Web services, [339](#)
- Security Administration Wizard, [132](#), [133](#)
- Security.Cryptography.CryptographicException, [227](#)
- SecurityPermission code access permission, [252](#)
- Security.XmlSyntaxException, [228](#)
- seeding random generator, [50–52](#)
 - for identical lists, [52](#)
- Select Case structure, for control array, [24](#)
- Select method, of DataTable object, [420](#)
- SELECT statement (SQL)
 - Data Adapter Configuration Wizard to create, [394](#), [394](#)

- DataAdapter task for, [404](#)
- WHERE clause, [405](#)
- SelectCommand property, of DataAdapter, [404](#)
- selection object in Word, [103](#), [104](#)
- SelectionColor property, of RichTextBox control, [587](#)
- SelectionFont property, of RichTextBox control, [587](#)
- SelectionList control
 - binding array to, [651](#)
 - for mobile computing, [650](#)
- self-describing object types, [12](#)
- Send method, for messages in queues, [347](#)
- sending
 - fax with Word's Wizard, [98](#)
 - graphics, [286–287](#)
 - message to TCP server, [299–300](#)
- SendKeys command, [53](#)
- SentTime property, of acknowledgment messages, [361](#)
- <seq>, to require complex element sequence, [484](#), [485](#)
- <Serializable> attribute, [65](#), [460](#)
 - and XML serialization, [72](#)
- serializable objects
 - creating, [65–79](#)
 - ClassSerializer project, [66](#), [66–71](#)
 - XSD to generate, [84–86](#)
- serialization, [53–57](#), [59](#), [60–65](#)
 - basic steps, [61–65](#)
 - custom, [79–81](#)
 - destination, [80](#)
 - firewall and, [61](#)
 - Imports statements for, [54](#)
 - overhead, [79](#)
 - reading back mixed data, [56–57](#)
 - of SQL Server data, [81–90](#)
 - types, [60–61](#)
 - XML, [495–501](#)
- SerializationContext parameter, of GetObjectData method, [80](#)
- SerializationInfo class, AddValue method, [79](#)
- serialized objects, messages as, [342](#)
- server controls, [272–273](#)
- Server Explorer

to add queue reference, [346](#)

Data Connections node, [327](#), [393](#)

ServerKey.snk file, [470](#)

servers

application, business logic on, [449](#)

database, [441](#)

IP addresses responded to, [291](#)

proxy, [290](#)

SQL Server

account for all users, [403](#)

connection to instance running on other machine, [403](#)

- security, [120](#)
- serialization of data, [81–90](#)
- TCP, sending message to, [299–300](#)
- TcpChatServer application, [301–305](#)
 - ChatClass, [302–303](#)
 - incoming messages, [305](#)
 - listening for requests on separate thread, [303](#)
- TCPServer project, [297–298](#)
- UDPServer application, [295](#), [295–296](#)
- web
 - downloading document from, [315](#)
 - user download of application from, [246](#)
 - Windows application deployment on, [249–250](#)
- Service element in WSDL, [331](#)
- ServiceControllerPermission code access permission, [252](#)
- serviced components, [468–473](#)
- session keys, [153](#)
- session state, in ASP.NET, [325](#)
- Set Application Identity window, [466](#), [467](#)
- SetAttributes method, of File class, [224](#)
- SetClip method, of Graphics object, [162](#)
- Setup Project
 - as New project option, [261](#)
 - properties, [265–266](#)
- Setup Wizard, as New project option, [262](#)
- Setup.exe, [265](#)
- Setup.ini, [265](#)
- SetValue method, [33](#)
- SHA1 algorithm, [141](#), [142](#)
- shallow copy of instance, [12](#)
- Shared command, [15](#)
- shared methods, [8](#)
- Shell command, [53](#)
- shortcuts for Internet-based application deployment, [246](#), [257](#)
 - creating, [266–267](#)
- ShowDialog method
 - of PageSetupDialog control, [165](#)

- of PrintPreview control, [167](#)
- ShowMajorGrid property, in PlotControl application, [592](#)
- ShowMinorGrid property, in PlotControl application, [592](#)
- ShowNetwork property, of PrintDialog control, [166](#)
- simple XML data types, declaring, [483](#)
- SimpleQueue project, [362–363](#)
 - processing acknowledgment messages, [365–366](#)
- single value type, [14](#)
- SingleLine member of RegExOptions enumeration, [549](#)
- singularities, [602](#)
- Site membership condition, [256](#)
- SkipVerification permission set, [251](#)
- slide transitions in user interface, [640](#), [640–641](#)
- sn command line tool, [470](#)
- SOAP, [333](#)
 - persistence with, [495–503](#)
- SOAP serialization, [60–61](#)
 - using, [63–64](#)
- SoapFormatter class, [61](#)
 - deserialize method, [499](#)
- Socket class
 - Bind method of, [294](#)
 - Send method, [295](#)
- SocketPermission code access permission, [252](#)
- sockets, [292–300](#)
- SocketType enumeration, [294](#)
- Software Restriction Policies (Windows), [122](#), [127–128](#)
- Solution Explorer, [459](#)
- solutions, [193](#)
- Sort property, of DataView object, [426](#)
- sorting
 - arrays, [31–33](#)
 - customized, [33–34](#)
 - DataGrid support, [276](#)
 - DataViews, [426](#)
- source code. *See* code
- Source property, of exception objects, [225](#)
- Specified property, [85](#)
- spell-check with Word objects, [94–98](#)
 - passing text directly, [96](#)

retrieving misspelled word list, [96–98](#)
for VB.NET TextBox, [94–95](#)

Split method

of RegEx class, [552](#)

of String class, [19](#)

SQL commands, Data Adapter Configuration Wizard to set up, [394](#), [394](#)

SQL connection to database, [273–274](#)

SQL Server

account for all users, [403](#)

connection to instance running on other machine, [403](#)

```
        Next
        If UnitsSold > 0 Then
            royalty = royalty + _
                _AuthorRoyalties.Tiers(lastTier).Royalty * _
                _price * UnitsSold
        End If
        Return royalty
    End Function
End Class

<Serializable()> Public Class Author
    Private _firstname As String
    Private _lastname As String
    <XmlAttributeAttribute()> Public Property FirstName() As String
        Get . . .
        Set . . .
    End Property

    <XmlAttributeAttribute()> Public Property LastName() As String
        Get . . .
        Set . . .
    End Property
End Class

<Serializable()> Public Class RoyaltySchedule
    Private _royaltyTiers() As RoyaltyTier

    Public Property Tiers() As RoyaltyTier()
        Get
            Return _royaltyTiers
        End Get
        Set(ByVal Value As RoyaltyTier())
            Dim tier As Integer
            For tier = 1 To Value.GetUpperBound(0)
                If Value(tier).Tier <= Value(tier - 1).Tier Then
                    ReDim Preserve Value(tier - 1)
                    _royaltyTiers = Value
                    Exit Property
                End If
            Next
            _royaltyTiers = Value
        End Set
    End Property

    <Serializable()> Public Class RoyaltyTier
        Private _tier As Integer
        Private _royalty As Decimal
        Public Property Tier() As Integer
            Get
```

Too bad. What we want isn't always what we get. For all its power and virtues, .NET is riddled with idiosyncratic, patternless constructions. Solecisms abound.

The situation is similar to the way that prepositions are used in English. *There are no fixed rules* that govern prepositions, yet they serve to shade the meaning of many sentences. Probably the single most obvious clue that a speaker is relatively illiterate (once they get verb forms figured out) is the misuse of prepositions. Because there are no rules, each person refines his use of prepositions by listening to educated speakers (not local TV reporters) and reading well-written text (not local newspaper stories). For example, many people use the preposition *for* inappropriately—using it as an all-purpose substitute when they don't know the correct preposition. An example: "There's a good chance for rain." (Should be "of.")

Having worked with VB.NET on a daily basis for years now, I can tell you that your educated guesses about writing instantiation code and how to employ other syntaxes will improve over time. But you'll never discover any fixed, reliable rules. There are too many exceptions, too many peculiarities.

Here's an illustration. The first confusion when trying to grammatically diagram the items shown in Figure 1.2 comes from the fact that *two* classes appear to be working together to accomplish the job of creating "a file in the specified path": `System.IO.FileStream` and `System.IO.File`. *Both* a filestream and a file object must be used in your code. However, since they both belong to the `System.IO` namespace, you need only import that single namespace:

```
Imports System.IO
```

The namespaces that are automatically added as defaults to each VB.NET Windows-style project are: `System`, `System.Data`, `System.Drawing`, `System.Windows.Forms`, and `System.XML`. These defaults, however, have changed in the past, and are likely to change in the future. One wonders: Why include the little used `System.Drawing` as a default namespace?

In addition, .NET automatically includes some less visible default namespaces—as you'll see later in this chapter in the section titled "Assemblies Three Ways."

Given that essentially *everything* is an object during runtime (including an integer variable, for example), you can expect that sometimes you may have a single line of code that involves objects from more than one namespace. Your first job when translating .NET documentation, then, is to see what namespaces are referenced in the example code, and *Imports* them.

Next, you usually need to figure out what kind of object is returned from the method you're working with. In the example in Figure 1.1, you can see that you must instantiate a `FileStream` object. Why? Because the function (method) `Create` returns a `FileStream` object. You can instantiate your `FileStream` object two ways:

```
Dim fstream As FileStream  
fstream = File.Create("c:\myfile.tst")
```

or you can put the whole thing on a single line:

```
Dim fstream As FileStream = File.Create("c:\myfilex.tst")
```

Notice that *sometimes* when you instantiate an object you must use the *New* keyword (triggering its constructor method). *Other times* you don't need it. A FileStream object can be created without the New constructor, as can other common objects, such as a string. But don't relax: other, equally common, objects must be instantiated with New. Sometimes you use New; sometimes you don't—and there's *no pattern or rule* you can learn.

```
        Return _tier
    End Get
    Set(ByVal Value As Integer)
        _tier = Value
    End Set
End Property

Public Property Royalty() As Decimal
    Get
        Return _royalty
    End Get
    Set(ByVal Value As Decimal)
        _royalty = Value
    End Set
End Property
End Class
End Class
```

The Create Book Objects button creates a few instances of the Book class and sets their properties. These instances are the variables BK0, BK1, and BK2. Each of the buttons on the left column serializes the Book objects, either individually or as a collection. Listing 3.4 shows the code behind the "Save Book Objects (Binary)" button, which serializes all of the Book objects in binary format.

LISTING 3.4: SERIALIZING INDIVIDUAL OBJECTS

```
Private Sub btnSaveBookBinary_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles btnSaveBookBinary.Click
    Me.Cursor = Cursors.WaitCursor
    Dim FS As FileStream
    Dim BF As New BinaryFormatter()
    Try
        FS = New FileStream("../SerializedObjects.bin", FileMode.Create)
        BF.Serialize(FS, BK0)
        BF.Serialize(FS, BK1)
        BF.Serialize(FS, BK2)
    Catch exc As Exception
        MsgBox(exc.Message)
    Finally
        FS.Close()
    End Try
    Me.Cursor = Cursors.Default
    btnLoadBooksBinary.Enabled = True
    TextBox1.Clear()
    TextBox1.Text = "Book objects saved in file SerializedObjects.bin"
End Sub
```


To serialize a series of objects, we simply call the appropriate formatter's `Serialize` method for each object. Each object's binary serialized version is appended to the file. To deserialize the original objects from the same file, click the `Load Book Objects (Binary)` button on the application's form. The code behind this button is shown in Listing 3.5.

LISTING 3.5: DESERIALIZING INDIVIDUAL OBJECTS

```
Private Sub btnLoadBooksBinary_Click(ByVal sender As System.Object,
                                     ByVal e As System.EventArgs) _
    Handles btnLoadBooksBinary.Click
    Me.Cursor = Cursors.WaitCursor
    Dim book0, book1, book2 As Book
    Dim FS As FileStream
    Try
        FS = New FileStream("../SerializedObjects.bin", FileMode.Open)
    Catch exc As Exception
        MsgBox(exc.Message)
        Me.Cursor = Cursors.Default
        Exit Sub
    End Try
    Dim BF As New BinaryFormatter()
    Try
        book0 = CType(BF.Deserialize(FS), Book)
        book1 = CType(BF.Deserialize(FS), Book)
        book2 = CType(BF.Deserialize(FS), Book)
    Catch exc As Exception
        MsgBox(exc.Message)
        Me.Cursor = Cursors.Default
        Exit Sub
    Finally
        FS.Close()
    End Try
    TextBox1.Clear()
    ShowBook(book0)
    ShowBook(book1)
    ShowBook(book2)
    Me.Cursor = Cursors.Default
End Sub
```

You can examine the code behind the remaining buttons to see how they serialize/deserialize individual objects in binary and SOAP format. The code is quite trivial and can be used by any application that has access to the `Books` class.

XML Serialization

Besides binary and SOAP serialization, the .NET Framework provides support for XML serialization. XML serialization differs from the other two serialization forms in that it serializes only public properties and fields; read-only and private properties are not serialized. Therefore, XML serialization doesn't preserve the state of the object being serialized. The output of XML serialization is both human and machine readable and doesn't require that classes are marked with the `<Serializable>` attribute. Moreover, you have control over the schema of the XML document that's produced with the help of attributes.

To use XML serialization, you must create an instance of the `XmlSerializer` class and then call its `Serialize` method (or the `Deserialize` method to extract data from an XML stream and populate an instance of a custom class). There's a major difference, however. The `XmlSerializer` class must be told in its constructor the type of object it's going to serialize. The constructor of the `XmlSerializer` class requires an argument, which is the type of the objects it will serialize or deserialize. Here's how we set up a new instance of the `XmlSerializer` class:

```
Imports System.Xml.Serialization
Dim serializer As New XmlSerializer(CO.GetType)
Dim FS As FileStream
FS = New FileStream(path, FileMode.Create)
serializer.Serialize(FS, CO)
FS.Close()
```

The first statement imports the `System.Xml.Serialization` namespace so that we won't have to fully qualify our references to the members of this class. The `CO` variable is an instance of the custom class (custom object), whose instances we intend to serialize. You can also pass the name of the class itself to the constructor, with a statement like the following:

```
Dim serializer As New XmlSerializer(GetType(CustomClass))
```

The `Serializer` object can only be used to serialize instances of the specific class and it will throw an exception if you attempt to deserialize a different class with it. Note also that all classes are XML-serializable by default and you don't have to prefix them with the `<Serializable>` attribute.

NOTE *XML Serializer can't serialize arbitrary objects. You must tell the `XmlSerializer` class the type of object it's going to serialize. In the background, .NET will create a temporary assembly, a process that will take a few moments. The temporary assembly, however, will remain in memory as long as the application is running and after the initial delay, XML serialization will be quite fast.*

Let's return to the ClassSerializer project and look at the code that serializes an array of custom objects. First, notice that you can't serialize individual objects into the same XML document, because an XML document can't have multiple root nodes. To XML-serialize multiple objects, you must assign them to the elements of an array and serialize the entire array. Notice also that the XmlSerializer class can't serialize other collections, such as ArrayLists and HashTables. The code that serializes the array of Book objects is shown in Listing 3.6.

LISTING 3.6: XML SERIALIZATION OF AN ARRAY OF OBJECTS

```
Private Sub btnSaveArrayXML_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles btnSaveArrayXML.Click
    Me.Cursor = Cursors.WaitCursor
    Dim AllBooks(3) As Book
    AllBooks(0) = BK0
    AllBooks(1) = BK1
    AllBooks(2) = BK2
    AllBooks(3) = BK3

    Dim serializer As New XmlSerializer(AllBooks.GetType)

    Dim FS As FileStream
    Try
        FS = New FileStream('..\SerializedXMLArray.xml', _
            FileMode.Create)
        serializer.Serialize(FS, AllBooks)
    Catch exc As Exception
        MsgBox(exc.InnerException.ToString)
    Exit Sub
    Finally
        FS.Close()
    End Try
    Me.Cursor = Cursors.Default
    btnLoadArrayXML.Enabled = True
    TextBox1.Clear()
    TextBox1.Text = _
        "Array of Book objects saved in file SerializedXMLArray.xml"
End Sub
```

The `XmlSerializer` class's constructor accepts as argument the array type. Because the array is typed, it can figure out the type of custom objects it's going to serialize.

CONTROLLING XML OUTPUT

One of the advantages of XML serialization is that it enables you to control the structure of the XML document that will be generated. You can do so by applying special attributes to the members of the class. To control the process of XML serialization, you can use the `XmlAttribute` class, which provides the following properties. To use any of the following attributes in the definition of your class, you must import the `System.Xml.XmlAttribute` namespace into the module that implements the custom class.

XmlAttribute Sometimes you may have to deserialize an XML document received from another domain. The document should comply with an existing schema, but you can't be sure it doesn't contain additional elements or attributes. To make sure that the deserialization process

won't stop with an exception, create a property in your class and mark it with the `XmlAny-Attribute` attribute. This property will be populated with any unknown attribute the XML document may contain.

XmlAnyElements This attribute is equivalent to the `XmlAnyAttribute` attribute, but handles unknown elements. A property marked with this attribute will be populated with any unknown element the XML document may contain.

XmlArray This attribute serializes the contents of a property as an XML array and is applied to all the properties that return arrays of objects.

XmlArrayItems This attribute is used in tandem with the `XmlArray` attribute and is applied to all properties that return an array of objects. The `XmlArrayItems` attribute specifies how the serializer renders the items of the array.

XmlAttribute By default, each property is rendered in the output as an element. Use this attribute to serialize a property as an attribute of the preceding element.

XmlChoiceIdentifier This attribute allows you to specify a set of values, from which the value of a property will be selected.

XmlDefaultValue This attribute sets the default value of an XML element or attribute.

XmlElement This attribute forces the serializer to render a public field as an XML element.

XmlAttribute This attribute determines how an enumeration member is serialized.

XmlAttribute This attribute tells the XML serializer to ignore (not serialize) the property to which it's applied.

XmlAttribute This attribute is applied to a single property in the class and makes it the root element of the XML document. You can also specify the name of the root element, if you want it to be different than the name of the element to which the attribute is applied.

XmlAttribute This attribute renders the property to which it's applied as XML text. Only one property in the class may be prefixed with the `XmlAttribute` attribute.

XmlAttribute This attribute controls how a type is serialized.

The two attributes used most often in the definition of a serializable class are the `XmlElement` and `XmlAttribute` attributes. The `XmlElement` attributes has a few properties, which can be specified in a pair of parentheses following the attribute's name, and they are:

IsNullable Allows you specify whether the property should be rendered even if set to null.

Data Type Allows you to specify the XSD type of the element the serializer will generate.

Element Name Allows you to specify the name of the element.

Namespace Allows you to associate the element with a namespace URI.

The `XmlAttribute` attribute, which has a similar function to that of the `XmlElement` attribute, supports only the `Data Type` and the `Namespace` properties, but not the `IsNullable` property. As

[Team Fly](#)

 Previous

Next 

you'd expect, the `XmlAttribute` attribute provides the `AttributeName` property instead of the `Element-Name` property, which allows you to replace the default name of the attribute with the string assigned to the `AttributeName` property.

Let's consider a very simple class, the `Product` class:

```
Public Class Product
    Public ProductID As Integer
    Public ProductName As String
    Public ProductPrice As Decimal
End Class
```

By default, an instance of the `Product` class will be serialized as follows:

```
<?xml version='1.0'?>
<Product xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ProductID>332</ProductID>
  <ProductName>Product Name</ProductName>
  <ProductPrice>42.15</ProductPrice>
</Product>
```

If you apply the `XmlAttribute` to the `ID` property, the `ID` of the product will become an attribute of the `Product` element:

```
<?xml version="1.0"?>
<Product xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  ProductID="332" >
  <ProductName>Product Name</ProductName>
  <ProductPrice>42.15</ProductPrice>
</Product>
```

The revised definition of the `Product` class with the `XmlAttribute` attribute is:

```
Imports System.Xml.Serialization
Public Class Product
    <XmlAttributeAttribute()> Public ProductID As Integer
    Public ProductName As String
    Public ProductPrice As Decimal
End Class
```

The `XmlAttribute` lets you modify the names of the members of an enumeration. The following enumeration can be used to replace state abbreviations with state names:

```
Public Enum States
    <XmlAttribute("Alaska")> AK
    <XmlAttribute("California")> CA
    . . .
End Enum
```

We're assuming that the XML document you're deserializing contains abbreviations and the objects that will accept the deserialization output have a field for storing state names.

By default, the root element of an XML serialized object is the name of the class from which the object was created. To change the name of the root element, apply the `XmlRootAttribute` attribute to the class declaration:

```
<XmlRootAttribute(ElementName='NWEmployee')>
Public Class Employee
    ' the class's definition
End Class
```

What good is this attribute if we can just change the name of the class? You may need it when the class definition must conform to specific rules, which may not apply to serialized instances of the class. You can also change the name of the root attribute programmatically, when you have no access to the class's code. The following code segment does the same by creating a new `XmlRootAttribute` object and passing it as an argument to the `XmlSerializer` class's constructor:

```
Dim root As New XmlRootAttribute()
root.ElementName = "NWEmployee"
Dim ser As New XmlSerializer(typeof(Employee), root)
```

The NETConfigFiles Project

In this section we'll develop a class for persisting application settings to a configuration file. We'll write a class that represents the application's settings, populate the class's members, and persist the instance of the class to a file. The settings can be persisted in either XML or binary format. The advantage of XML files is that they can be edited outside the context of the application they describe. If you don't want users to edit the settings, you can use binary serialization, which is faster. However, you can allow administrators to edit an application's settings in an XML file.

The `NETConfigFiles` project contains two classes that describe the application settings: the `XMLConfigurationClass` and the `BINConfigurationClass`. Both classes store the same data, which are mapped to the following private data:

```
Public Class AppConfig
    Private _location As Point
    Private _size As Size
    Private _datapath As String
    Private _recentfiles() As String
    Private _passwordDigest(16) As Byte
    Private _printerName As String
End Class
```

Each of the two classes contain Property Set and Get procedures for each of these properties. In addition, they provide a `Save` and a `Load` method, which persist the settings to a file and load the settings from a file, respectively. The `Save` method accepts two arguments, the path of the file and an instance of the `XMLConfigurationClass` class (or the `BINConfigurationClass` class). The `Load` method accepts the path of the file with the settings and returns an instance of one in the corresponding class. The `Save` method of the `XMLConfiguration` class is shown in Listing 3.7:

[Team Fly](#)

 Previous

Next 

LISTING 3.7: PERSISTING AN INSTANCE OF THE APPCONFIG CLASS IN XML

```
Public Function Save(ByVal path As String, ByVal CF As AppConfig) _  
    As Boolean  
    Dim serializer As New XmlSerializer(CF.GetType)  
    Dim FS As FileStream  
    Try  
        FS = New FileStream(path, FileMode.Create)  
        serializer.Serialize(FS, CF)  
        FS.Close()  
        Return True  
    Catch exc As Exception  
        Throw exc  
    End Try  
    FS.Close()  
    Return True  
End Function
```

The code simply sets up an `XmlSerializer` class and calls its `Serialize` method. The `Load` method is just as simple; it's shown in Listing 3.8.

LISTING 3.8: RESTORING AN INSTANCE OF THE APPCONFIG CLASS FROM XML

```
Public Function Load(ByVal path As String) As AppConfig  
    Dim CF As New AppConfig  
    Dim serializer As New XmlSerializer(CF.GetType)  
    Dim FS As FileStream  
    Try  
        FS = New FileStream(path, FileMode.Open)  
        CF = serializer.Deserialize(FS)  
        FS.Close()  
        Return CF  
    Catch exc As Exception  
        FS.Close()  
        Return Nothing  
    End Try  
End Function
```

To save its settings, the application's code creates an instance of the `AppConfig` class and sets its properties. Then it serializes the `CF` object, which contains the current settings of the application. These settings include the name of a printer, a list of recent files, and a password's hash code. Identical passwords produce identical hash codes, so we don't have to store the actual password in the file. We know that the user supplied the correct password if its hash code matches the hash code of the correct password. Listing 3.9 shows the code that reads back the application settings (it's the code of the `Save XML Configuration` button of the test form, shown in Figure 3.3).

LISTING 3.9: SAVING THE APPLICATION CONFIGURATION

```
Private Sub btnSaveConfig_Click(ByVal sender As System.Object, _  
                                ByVal e As System.EventArgs) _  
                                Handles btnXMLSaveConfig.Click  
  
    Dim printerName As String  
    PrintDialog1.PrinterSettings = New _  
        System.Drawing.Printing.PrinterSettings  
    If PrintDialog1.ShowDialog() = DialogResult.OK Then  
        printerName = PrintDialog1.PrinterSettings.PrinterName()  
    End If  
  
    Dim AppCFG As New XMLConfigurationClass.AppConfig  
    Dim files() As String = {'drive\folder1\folder2\file1', _  
                            "drive\folder1\folder3\file2", _  
                            "drive\folder2\folder2\file3"}  
  
    AppCFG.recentfiles = files  
    AppCFG.datapath = "Application s\data\path\here"  
    AppCFG.Location = New Point(230, 320)  
    AppCFG.Size = New Size(100, 300)  
    Dim DigestProvider As New _  
        System.Security.Cryptography.MD5CryptoServiceProvider  
    AppCFG.PasswordDigest = _  
        DigestProvider.ComputeHash(_  
            System.Text.Encoding.Unicode.GetBytes("secret password"))  
    AppCFG.PrinterName = printerName  
    Try  
        XMLConfig.Save("../AppConfigFile.xml", AppCFG)  
        MsgBox("Configuration file saved successfully")  
    Catch exc As Exception  
        MsgBox("Failed to save configuration file!")  
    End Try  
End Sub
```

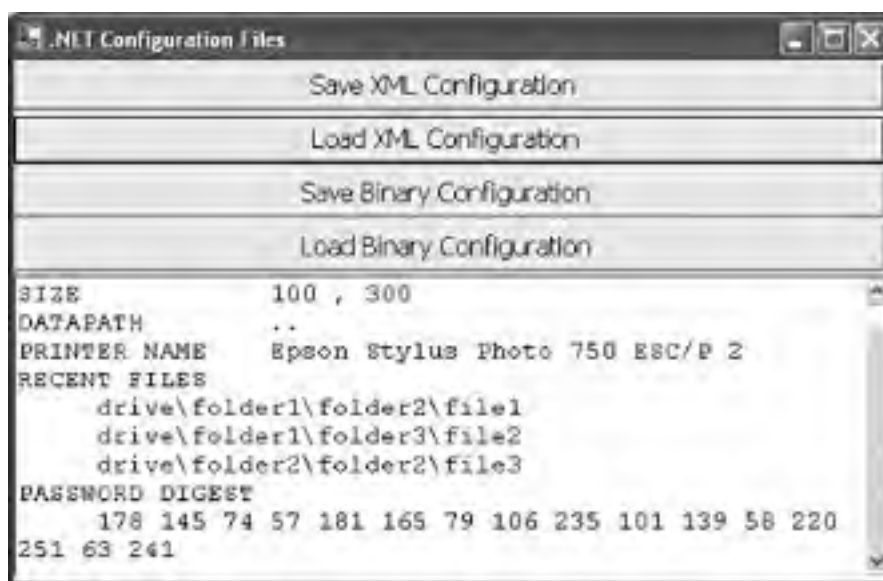


FIGURE 3.3 The NETConfig-Files project's form

[Team Fly](#)

 Previous

Next 

Figure 1.2 is a diagram illustrating how to translate the information found in the Object Browser into usable source code:

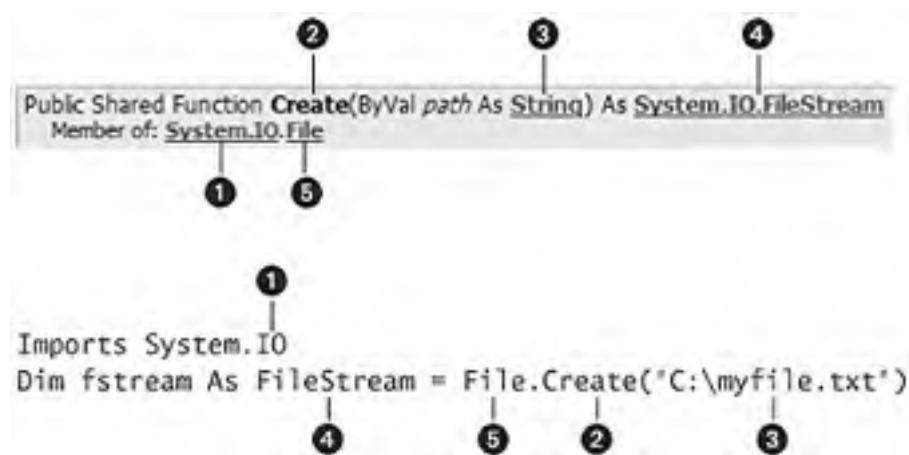


FIGURE 1.2 Make these adjustments to the object description to come up with usable code.

The technical reasons why *some* objects demand that you instantiate them (use a `New` command) and others don't are rather tedious, and are of more interest to the people at Microsoft who wrote .NET and who continue to revise it than to us programmers.

The distinction is essentially trivial—you must use `New` when they say you must. They've simply decided that some objects need it; others don't. Although virtually everything in VB.NET is an object, some methods are considered "common" enough that they are just there, just available to you, like the `ABS` command (you don't have to instantiate the `Math` class to use its `ABS` method, though you *do* have to mention the `Math` namespace). These "shared" methods require no instantiation. The answer to this discrepancy can be found later in this chapter in the sidebar titled "What's Shared, What's Static?"

If you're trying to instantiate an object, and get an error message saying, for example, that one data type "cannot be converted" into another data type (or the infamous "Object reference not set to an instance of an object" error message), this usually means that you need the `New` command. You must use it. Just try adding it to see if this solves the problem. Sure, it's whimsy, but so is the fact that some collections in .NET are zero-based and others are one-based. You just have to learn all the little exceptions to the "rules."

For example, this line causes the IDE to protest:

```
Dim dirI As DirectoryInfo = ("c:\")
```

Instead, you must use this format:

```
Dim dirI As DirectoryInfo  
dirI = New DirectoryInfo("c:\")
```

What happens here is that you are trying to assign a string data type "c:\" to your DirectoryInfo object dirI. No can do. You must instead create the object variable dirI, then instantiate a new DirectoryInfo object and assign it to your object variable. You can also use this format:

```
Dim dirI As DirectoryInfo = New DirectoryInfo("c:\")
```

Do you suppose this format would work?

```
Dim dirI As New DirectoryInfo
```

[Team Fly](#)

 Previous

Next 

The *Info* argument exposes methods for reading all data types of the .NET language runtime. The following constructor corresponds to the custom serializer shown above:

```
Friend Sub New(ByVal info As SerializationInfo, _
              ByVal context As StreamingContext)
    _fld1 = info.GetString("classField1")
    _fld2 = info.GetDecimal("classField2")
    _fld3 = info.GetBoolean("classField3")
    Dim creationDate As DateTime = info.GetDate("CreationDate")
End Sub
```

Notice that this overloaded form of the custom class's constructor is declared with the *Friend* modifier. The class should contain a simple *New* public constructor, which other applications can access, but the constructor responsible for deserializing a class should not be visible outside the containing class.

The *CreationDate* item is serialized, but there's no property in the class to store this value. That's why the deserializer stores this value to a local variable.

The SerializationContext Parameter

The second parameter of the *GetObjectData* method is a *SerializationContext* object, which describes the destination of the serialization process, or the source of the deserialization process (where the serialized data are sent, or where the deserialized data are coming from). The *SerializationContext* class has a few methods; the most important of these is the *State* property, which can be a member of the *StreamingContextState* enumeration, shown in Table 3.1.

TABLE 3.1: THE STREAMINGCONTEXTSTATE ENUMERATION

Member	Description
All	Encapsulates all possible contexts
Clone	Specifies that the object being serialized is cloned
CrossAppDomain	Specifies that the context is that of a different application domain
CrossMachine	Specifies that the source or destination is that of a different computer
CrossProcess	Specifies that the source or destination is a different process on the same computer
File	Specifies that the source or destination is a file
Other	Specifies that the serialization context is unknown
Persistence	Specifies that the source or destination is a persisted store (database, file, or private storage)
Remoting	Specifies that the serialized object is remoted to an unknown location

The Serialize Order button creates a new order by setting up an Orders object and serializes it to the file. The Deserialize Order button deserializes the order's values into a new instance of the Orders class and displays them on the TextBox control. To commit the new order to the database, click the Commit Order button. The last button, Read Order, prompts you for an order's ID and reads the corresponding order from the database directly into an instance of the Orders class.

First, we must create a class for describing the orders. Since the class should be able to accept data from the database, we'll use some of Visual Studio's tools to design the class. Start by implementing and executing a query that will return the desired data from the database. Listing 3.10 is a query that returns the header and details of a specific order from the Northwind database.

LISTING 3.10: RETRIEVING ORDER INFORMATION IN XML FORMAT

```
SELECT Orders.OrderID AS ID,
       Orders.CustomerID,
       CONVERT(datetime, Orders.OrderDate ) AS OrderDate,
       CONVERT(datetime, Orders.ShippedDate ) AS ShipDate,
       (SELECT CompanyName FROM
        Customers WHERE
        Customers.CustomerID=Orders.CustomerID) AS Customer
       (SELECT ContactName FROM
        Customers WHERE
        Customers.CustomerID=Orders.CustomerID) AS Contact,
       Products.ProductID, Products.ProductName,
       CAST(Details.UnitPrice AS numeric(8,2)) As Price,
       CAST(Details.Quantity AS int) AS Quantity,
       CAST(Details.Discount AS numeric(8,2)) AS Discount
FROM [Order Details] Details
     INNER JOIN Orders ON Orders.OrderID = Details.OrderID
     INNER JOIN Products ON Products.ProductID = Details.ProductID
WHERE Orders.OrderID = 10910
FOR XML AUTO
```

The last clause of the statement instructs SQL Server to return the result of the query in XML format, as shown in Figure 3.5. The entire query is returned as a single line, but we've edited the XML document on the result pane of the Query Analyzer window to make it easier to read.

The result of this query is an XML document with the following structure:

```
<Orders ID=''10910" CustomerID="WILMK"
       Customer="Wilman Kala" Contact= "Matti Karttunen" >
<Products ProductID="19"
       ProductName="Teatime Chocolate Biscuits" Price="9.20"
       Quantity="12" Discount="0.00"/>
<Products ProductID="49" ProductName= "Maxilaku" Price="20.00"
       Quantity="10" Discount="0.00"/>
<Products ProductID="61" ProductName= "Sirop d&apos;érable"
       Price="28.50" Quantity="5" Discount="0.00"/>
</Orders>
```




FIGURE 3.5 Retrieving the result of an XML query in Query Analyzer

The <Orders> element represents the header of the order and the <Products> elements represent detail lines. Notice that the tables involved in the query have become elements of the XML document and all the fields are attributes of their corresponding table element.

There's a catch here: We have selected a few columns from the Customers table with subqueries, but the Customers table doesn't appear anywhere in the selection list of the query. If we introduced another join with the Customers table, the XML document would have a Customers element, and the customer's fields would become attributes of the Customers element. We wanted to include the customer's data as attributes of the Orders element, without introducing another element. You can experiment by rewriting the query and see how SQL Server creates the XML document with the hierarchy of the requested data. You will have to tweak the query to get a document with the desired structure; don't forget to append the FOR XML AUTO clause to the query.

At this point, you can write a class with public fields and subclasses that reflect the hierarchy of the XML document returned by the query. Although it's fairly straightforward to build this class manually, you can use the XSD command-line tool to automate the generation of the class. To use this tool, copy the XML document returned by the query, paste it into a new text document, and save the document in a file with a short path with extension XML. You can save it as Orders.xml in the root path (don't save it to a folder with a long pathname, because

you'll have to type the entire pathname, as you'll see in the following statements). Then open a Command Prompt window and switch to the `SDK/Bin` folder under the folder of Visual Studio 2003 (or the `FrameworkSDK/Bin` folder if you're using Visual Studio .NET). There you can execute the following statement to extract an XML schema from the document:

```
xsd c:\Orders.xml
```

The XSD utility will process the XML file and will generate a new file with the document's XSD schema. The `Orders.xsd` file will be saved in the current folder. Run again the XSD utility, this time

specifying the name of the XSD file and two options: the /classes option (to generate the classes that correspond to the specified schema) and the /language option (to generate VB code):

```
xsd Orders.xsd /classes /language:vb
```

This command will generate a new file, the `Orders.vb` file, which contains a serializable class that has the same structure as the XML document. Listing 3.11 shows what this file looks like (we've removed all the comments inserted by the code generator in the listing).

LISTING 3.11: THE NWORDERS CLASS

```
Option Strict Off
Option Explicit On

Imports System.Xml.Serialization

<System.Xml.Serialization.XmlRootAttribute([Namespace] := "", IsNullak
    <System.Xml.Serialization.XmlElementAttribute("Orders",
        Form:=System.Xml.Schema.XmlSchemaForm.Unqualified)> _
    Public Items() As Schema1Orders
End Class

Public Class Schema1Orders

    <System.Xml.Serialization.XmlElementAttribute("Products",
        Form:=System.Xml.Schema.XmlSchemaForm.Unqualified)> _
    Public Products() As Schema1OrdersProducts

    <System.Xml.Serialization.XmlAttributeAttribute()> _
    Public ID As Integer

    <System.Xml.Serialization.XmlIgnoreAttribute()> _
    Public IDSpecified As Boolean

    <System.Xml.Serialization.XmlAttributeAttribute()> _
    Public OrderDate As Date

    <System.Xml.Serialization.XmlIgnoreAttribute()> _
    Public OrderDateSpecified As Boolean

    <System.Xml.Serialization.XmlAttributeAttribute()> _
    Public ShipDate As Date

    <System.Xml.Serialization.XmlIgnoreAttribute()> _
    Public ShipDateSpecified As Boolean
```

```
<System.Xml.Serialization.XmlAttributeAttribute()> _
Public Customer As String

<System.Xml.Serialization.XmlAttributeAttribute()> _
Public Contact As String
End Class

Public Class Schema1OrdersProducts

<System.Xml.Serialization.XmlAttributeAttribute()> _
Public ProductID As Integer

<System.Xml.Serialization.XmlIgnoreAttribute()> _
Public ProductIDSpecified As Boolean

<System.Xml.Serialization.XmlAttributeAttribute()> _
Public ProductName As String

<System.Xml.Serialization.XmlAttributeAttribute()> _
Public Price As Decimal

<System.Xml.Serialization.XmlIgnoreAttribute()> _
Public PriceSpecified As Boolean

<System.Xml.Serialization.XmlAttributeAttribute()> _
Public Quantity As Integer

<System.Xml.Serialization.XmlIgnoreAttribute()> _
Public QuantitySpecified As Boolean

<System.Xml.Serialization.XmlAttributeAttribute()> _
Public Discount As Decimal

<System.Xml.Serialization.XmlIgnoreAttribute()> _
Public DiscountSpecified As Boolean
End Class
```

The Boolean public fields with the "Specified" suffix are there to handle Null values. If a field has a Null value, you should set the corresponding Specified property to True to indicate this condition. We aren't going to use these properties, so you can safely remove them.

All the elements of the XML schema generated by the XSD class are prefixed with the "Schema1" string. You can edit the code of the class and rename the elements by removing the "Schema1" prefix. This default prefix is generated by SQL Server and it doesn't help the readability of the code.

The class designed by the XSD command-line tool represents an order. We can now retrieve an order from SQL Server as an XML document, deserialize each order into an instance of the Schema1 Orders class, and pass it to the presentation tier. Using XML serialization, you can either persist this object

between sessions or transmit it from one application (or computer) to another. Let's say you're developing an ordering system for the Northwind corporation. You can share with certain trusted customers the company's price list, as well as the structure of the XML file for the orders. Other companies can create their own orders and submit them to your corporation as XML files.

Every time your application receives an order in XML format, it can deserialize it and use the data to commit a new order into the database. Customers using .NET technology can make use of the custom class to create orders: they can populate an instance of the custom class, serialize the object into XML, and send it to your system for processing. Let's see how this can be done.

The NWOrders Project

Now we can build the sample application that retrieves data from SQL Server and passes them to the application tier as custom objects. Typed DataSets, which are discussed in Chapters 15 and 17, are quite convenient, but DataSets are not business objects. We still have to worry about relations between tables, access tables, and rows, and work with a model that resembles a database. The code at the presentation tier is simplified a lot if we can program against objects.

Start a new project, name it NWOrders, and add a new class to it, the NWOrders class. Then paste the class definition generated by the XSD command-line tool into its code window (just delete the public fields suffixed by the string "Specified," as explained already). Switch to the project's main form and place the controls you see in Figure 3.4. The Serialize Order button creates a new order by populating the fields of an instance of the Orders class. The code always creates the same order, with the statements shown in Listing 3.12. Then the order is serialized in the `Order.xml` file, which is read back by the code of the other two buttons on the form. We've chosen to create the same order to simplify the code (you can create random new orders, or use the interface we discuss in Chapter 18 to create new orders).

LISTING 3.12: CREATING AND SERIALIZING A NEW ORDER

```
Private Sub btnSerialize_Click( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnSerialize.Click
    Dim NW As New NWOrders.Orders
    Dim orderedproducts(2) As NWOrders.OrdersProducts

    orderedproducts(0) = New NWOrders.OrdersProducts
    orderedproducts(0).ProductID = 12
    orderedproducts(0).Quantity = 12
    orderedproducts(1) = New NWOrders.OrdersProducts
    orderedproducts(1).ProductID = 48
    orderedproducts(1).Quantity = 6
    orderedproducts(2) = New NWOrders.OrdersProducts
    orderedproducts(2).ProductID = 30
    orderedproducts(2).Quantity = 6
```

```
NW.CustomerID = "ALFKI"  
NW.Customer = "Alfreds Futterkiste"
```

[Team Fly](#)

 Previous

Next 

```
NW.Contact = 'Maria Anders"  
NW.OrderDate = Now.Today  
NW.Products = orderedproducts  
  
Dim serializer As New XmlSerializer(NW.GetType)  
Dim FS As FileStream  
Try  
    FS = New FileStream("../Order.xml", FileMode.Create)  
    serializer.Serialize(FS, NW)  
Catch exc As Exception  
    MsgBox(exc.InnerException.ToString)  
Finally  
    FS.Close()  
End Try  
txtOrder.Clear()  
txtOrder.AppendText("Order serialized successfully")  
End Sub
```

The code is straightforward. We start by creating an array of `OrdersProducts` objects, with the order's detail lines. This array contains the IDs of the ordered products and the corresponding quantities. Notice that we don't set the `Price` field, because customers can't set their prices. The prices will be determined by the system that will process the order (we only assume that customers have the most up-to-date version of our price list and the discount policy is simple and clear). We're not going to deal with the practical issues that may arise, just how to serialize and deserialize objects.

Once we've populated the `NW` object's fields, we serialize it into the `Order.xml` file with the `XmlSerializer` and display the appropriate message. Let's now see how the `Deserialize Order` button handles the deserialization of the `Order.xml` file and creates a new instance of the `NW` order, this time on a different machine. The sample project doesn't use any variables to maintain state. The `Deserialize Order` button simply reads the XML file created by the `Serialize Order` button and displays the order's details on a `TextBox` control, as shown in Listing 3.13.

LISTING 3.13: DESERIALIZING XML INTO AN INSTANCE OF A CUSTOM CLASS

```
Private Sub btnDeserialize_Click(  
    ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles btnDeserialize.Clic  
Dim NW As New NWOrders.Orders  
Dim serializer As New XmlSerializer(NW.GetType)  
Dim FS As FileStream  
Try  
    FS = New FileStream("../Order.xml", FileMode.Open)  
    NW = CType(serializer.Deserialize(FS), NWOrders.Orders)  
Catch exc As Exception  
    MsgBox(exc.InnerException.ToString)  
Finally  
    FS.Close()
```

```
End Try
Dim msg As String
msg = "'Order placed on" & NW.OrderDate
msg = msg & " By " & NW.Customer & " (" & NW.CustomerID & ")"
Dim itm As Integer
For itm = 0 To NW.Products.Length - 1
    msg = msg & vbCrLf & "ID " & NW.Products(itm).ProductID
    msg = msg & QTY & NW.Products(itm).Quantity
Next
txtOrder.Clear()
txtOrder.AppendText(msg)
End Sub
```

As soon as you type the name of the object that represents the order, the NW object in Listing 3.13, and the period following it, the order's fields will appear in a drop-down list and you can select the desired member. The ID that relates the order to its details becomes irrelevant, because each Order's object contains both the header and the details. We just pass the responsibility of handling primary and foreign keys to the database and we don't have to worry about relations between DataTables.

The Commit Order button performs the same deserialization as the Deserialize Order button to reconstruct another NWOrders.Orders object. It then uses its fields as arguments to the appropriate SQL statements that commit the order to the database in a transactional mode, as shown in Listing 3.14.

LISTING 3.14: COMMITTING AN ORDER TO THE DATABASE

```
Private Sub btnCommit_Click( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnCommit.Click
Dim NW As New NWOrders.Orders
' retrieve the order from XML file and store it in the NW object
Dim serializer As New XmlSerializer(NW.GetType)
Dim FS As FileStream
Try
    FS = New FileStream("../Order.xml", FileMode.Open)
    NW = CType(serializer.Deserialize(FS), NWOrders.Orders)
Catch exc As Exception
    MsgBox(exc.InnerException.ToString)
Finally
    FS.Close()
End Try
' now commit the order to the database by executing the appropriate SQL
' statements and passing the fields of the NW object as arguments
Dim CMD As New SqlClient.SqlCommand
CMD.CommandText = "INSERT INTO ORDERS " & _
    "(CustomerID, OrderDate) VALUES ('" & _
    NW.CustomerID & "', '" & NW.OrderDate & " ' ); " & _
    "SELECT @@IDENTITY"
```

```
CMD.CommandType = CommandType.Text
Dim CN As New SqlConnection
CN.ConnectionString = data source=.initial catalog=Northwind; ' & _
    "integrated security=SSPI;persist security info=False; "
    "workstation id=POWERTOOLKIT;packet size=4096"
CMD.Connection = CN
Dim TRN As SqlConnection.SqlTransaction
CN.Open()
TRN = CN.BeginTransaction
CMD.Transaction = TRN
Try
    Dim orderID As Integer = CInt(CMD.ExecuteScalar)
    Dim itm As Integer, rows As Integer
    For itm = 0 To NW.Products.Length - 1
        CMD.CommandText = INSERT INTO [Order Details] " & _
            "(OrderID, ProductID, Quantity, UnitPrice, Discount) " & _
            "SELECT "& orderID & ", "& NW.Products(itm).ProductID & _
            ", " & NW.Products(itm).Quantity & ", "& _
            "UnitPrice, " & NW.Products(itm).Discount & _
            "FROM Products WHERE ProductID =" & NW.Products(itm).ProductID
        rows = CMD.ExecuteNonQuery()
        If rows = 0 Then
            TRN.Rollback()
            CN.Close()
            MsgBox("TRANSACTION ABORTED! " & vbCrLf & _
                "Error in inserting detail line for Product ID " & _
                NW.Products(itm).ProductID & vbCrLf)
            Exit Sub
        End If
    Next
    TRN.Commit()
    txtOrder.Clear()
    txtOrder.AppendText("ORDER " & _
        orderID.ToString & " COMMITTED TO DATABASE")
Catch ex As Exception
    TRN.Rollback()
    MsgBox("Could not insert order into Northwind database! " & _
        vbCrLf & ex.Message)
Finally
    CN.Close()
End Try
End Sub
```

The last button on the form, the Read Order button, demonstrates how to read XML data out of SQL Server and serialize it into an instance of the Orders class. First, you must create the ReadOrder stored procedure and attach it to the database. The code of the ReadOrder stored procedure contains the T-SQL code of Listing 3.10 (you can find the stored procedure's code in the project's README

Nope. Try it and you're told that "Overload resolution failed because no accessible 'New' accepts this number of arguments." Too bad that whoever wrote this error message didn't bother to translate it into English. Visual Basic is famous for its English-like syntax and its avoidance of silly techno-talk and garbled phrasing. That's why VB is by far the world's most popular programming language. It's sensible and attempts to be straightforward. Alas, .NET is something of a setback in this department, but the Help feature and other documentation has been considerably improved in the past three years, and continues to become increasingly clear and sensible. Perhaps Microsoft will hire a writer who knows VB to continue the improvements. One can hope.

This "Overload resolution failed because no accessible 'New' accepts this number of arguments" error message seems to mean that the `DirectoryInfo` class doesn't have a constructor that requires no parameters. In other words, in the `DirectoryInfo` class, there is no `Sub New()`, a constructor that requires no parameters. However, that's not precisely the problem.

The `DirectoryInfo` class requires that you pass it a string containing a filepath. If you look up `DirectoryInfo` in Help, and look at its constructor, you find this, its *only* constructor:

```
[Visual Basic]
Public Sub New( _
    ByVal path As String _
)
```

This seems to mean that you cannot instantiate a `DirectoryInfo` object without passing a file path to it. How about the `FileStream` object? It's similar to `DirectoryInfo` because it also needs a file path to do its job. Why aren't these two similar objects—`FileStream` and `DirectoryInfo`—created the same way?

It's really curious, but some of the rules and behaviors in .NET are merely quixotic, a matter of what seems to us programmers to be mere whimsy. The `FileStream` object's constructor is overloaded; there are nine versions of its constructor (nine different "signatures" meaning "parameter lists"). But `FileStream`, like `DirectoryInfo`, offers no constructor that requires no parameters.

Here's the simplest description given by the Object Browser of how to use the `DirectoryInfo` object's `CreateDirectory` method:

```
Public Function GetDirectories() As System.IO.DirectoryInfo()
    Member of: System.IO.DirectoryInfo
Summary:
Returns the subdirectories of the current directory.
Return Values:
An array of System.IO.DirectoryInfo objects.
```

Clearly, you must create an array that can hold `DirectoryInfo` objects, but first you have to create another `DirectoryInfo` object (and in the process of instantiating it, its constructor demands that you supply a path):

```
Dim di As DirectoryInfo = New DirectoryInfo("c:\")
```

Then you instantiate the array (without using `As New`) and execute the `GetDirectories` method:

```
Dim dirs As DirectoryInfo() = di.GetDirectories()
```

[Team Fly](#)

 Previous

Next 

file). This stored procedure returns an XML document with the values of an order and must be executed with the ExecuteXmlReader method of the SqlCommand object. This method returns an Xml-Reader object, which you can feed to the Deserialize method of the appropriate XmlSerializer object to populate an instance of the Orders class. Listing 3.15 shows the code behind the Read Order button.

LISTING 3.15: DESERIALIZING AN XML DOCUMENT REPRESENTG AN ORDER

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

    Dim NW As New NWOrders.Orders
    Dim serializer As New XmlSerializer(NW.GetType)
    Dim CMD As New SqlClient.SqlCommand
    CMD.CommandText = "ReadOrder"
    CMD.CommandType = CommandType.StoredProcedure
    CMD.Parameters.Clear()
    Dim orderID As Integer = Convert.ToInt32(TextBox(
        "Enter the ID of the desired order", , "10903" ))
    CMD.Parameters.Add( @orderID , orderID)
    Dim CN As New SqlClient.SqlConnection
    CN.ConnectionString = "data source=localhost;initial " & _
        "catalog=Northwind;integrated security=SSPI; " & _
        "persist security info=False; " & _
        "workstation id=POWERTOOLKIT;packet size=4096"
    CMD.Connection = CN
    CN.Open()
    Dim XMLReader As System.Xml.XmlReader = CMD.ExecuteXmlReader
    Try
        NW = CType(serializer.Deserialize(XMLReader), NWOrders.Orders)
        ShowOrder(NW)
    Catch Ex As Exception
        MsgBox(Ex.Message)
    Exit Sub
    Finally
        CN.Close()
    End Try
    ShowOrder(NW)
End Sub
```

The code is straightforward and doesn't contain any ADO.NET-specific code. We request SQL Server to return the result of a query in XML format, with a predefined schema. The response's schema matches the definition of a custom class, and we can deserialize the XML document returned by SQL Server into an instance of this class. The advantage of this approach is that presentation tier developers need not be concerned with the actual structure of the database, or work with ADO.NET objects; they simply program against business objects and call a middle tier component to communicate with the database.

This page intentionally left blank.

```
Dim t As String = TextBox1.Text

If t <> "" Then Clipboard.SetDataObject(TextBox1.Text) 'move
clipboard

With d
    .Content.Paste()
    .CheckSpelling()
    .Content.Copy()
    id = Clipboard.GetDataObject

    TextBox1.Text = CType(id.GetData(DataFormats.Text), Strin
    .Close()
End With

w.Quit(false) 'exit word

Me.Text = "Spellcheck complete"
End Sub
```

If there were no errors, nothing seems to happen so you want to use the Me.Text message at the end, or a message box, to inform the user that indeed the spelling has been checked. If there is an error, the usual Word spell-check dialog box appears and the user can edit their work.

To use the Word spell-check dialog box, you must resort to the ancient technique of transferring your TextBox's contents to the Clipboard, then pasting it into your Word document object. After the spell-check is finished, the Clipboard is again used as a way station between the document object, which copies its contents (using .Content.Copy) back into the Clipboard, and the .NET GetData method, which is used to paste the Clipboard contents back into the TextBox. To also check the grammar, simply replace .CheckSpelling with .CheckGrammar. The latter option includes a spell-check.

WHOOOPS: DESTROYING THE APPLICATION OBJECT

When working with instantiated Office objects you may get an error message that says: "An unhandled exception of type 'System.Runtime.InteropServices.COMException' occurred in mscorlib.dll. Additional information: The RPC server is unavailable." If you see this, it means that you've destroyed the application object and are now trying to use it. It's gone, so you cannot continue to reference it in your source code. This could happen, for example, if you use the Word application object's Quit method (destroying the application object) prematurely. Look in your source code to see where you've used Quit and move it to a better location, such as your form's Closing event, like this:

```
Private Sub Form1_Closing(ByVal sender As Object, _
    ByVal e As System.ComponentModel.CancelEventArgs) Handles MyBase.Clo
    WordApp.Quit(False)
End Sub
```

Notice the pattern in the code in this example. First you instantiate the Word application object, followed by instantiation of a Word document object. After that you're free to use a variety of methods of the document object: `Content.Paste`, `CheckSpelling`, `Content.Copy`, and `Close`.

Notice, too, the use of the argument *false* following the Word application object's `Quit` method. This instructs Word not to throw up a dialog box asking if you want to save the contents of the document. You don't; you're simply borrowing functionality from Word and you put your `TextBox`'s contents into this Word document merely temporarily so you could spell-check it. You can dump the document after you've returned the correct contents back to the VB.NET `TextBox`.

PASSING TEXT DIRECTLY

The next example is simpler (see Listing 4.2); it doesn't involve the Clipboard because you pass the text in the `TextBox` directly to Word's spell-check function and get a yes or no answer—it checks out, or there's an error somewhere in the text. This would be useful if you wanted to allow your users to type in a single word or small phrase and verify its spelling.

LISTING 4.2: PASSING TEXT DIRECTLY TO THE WORD SPELL-CHECKER

```
Private Sub Button1_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles Button1.Click  
  
    Dim w As Object = New Word.Application  
    w.Visible = False  
  
    Dim t As String = TextBox1.Text  
  
    If w.CheckSpelling(t) Then  
        MsgBox(t & " contains no spelling errors.")  
    Else  
        MsgBox(t & " does include an error in spelling.")  
    End If  
  
    w.Quit()  
  
End Sub
```

RETRIEVING A LIST OF MISSPELLED WORDS

As usual in .NET, there are several ways to accomplish the same goal. This final spell-check example shows you how to obtain a list of misspelled words in a document you submit to Word, and a separate list of the spell checker's suggested alternatives to each misspelled word. With this functionality, you don't need to employ the Word spell-check dialog box as your user interface. Instead, you can design your own custom interface in VB.NET, showing the user a list box, for example, containing all the misspelled words in the entire document.

Put a TextBox, two ListBoxes, and a button on a form. You'll need to give the Word application instantiation form-wide scope because it's needed by more than just the button's event. So type this line outside any event:

```
Dim w As Object = New Word.Application
```

Then type in the code in Listing 4.3.

LISTING 4.3: RETRIEVING MISSPELLED WORDS AND ASSOCIATED SUGGESTED ALTERNATIVES

```
Private Sub Button1_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles Button1.Click  
  
    w.Documents.Add()  
    w.visible = False  
  
    Dim range As Word.Range  
    range = w.ActiveDocument.Range  
    range.InsertAfter(TextBox1.Text) 'dump in the TextBox.text  
  
    Dim s As Word.ProofreadingErrors  
    s = range.SpellingErrors  
    If s.Count > 0 Then  
        ListBox1.Items.Clear()  
        Dim i As Integer  
        Dim ss As String  
        For i = 1 To s.Count  
            ss = s.Item(i).Text  
            If ListBox1.FindStringExact(ss) < 0 Then  
                ListBox1.Items.Add(ss)  
            End If  
        Next  
    End If  
  
End Sub  
  
Private Sub ListBox1_SelectedIndexChanged(ByVal sender As System.  
ByVal e As System.EventArgs) Handles ListBox1.SelectedIndexChanged  
    Dim s As Word.SpellingSuggestions = w.GetSpellingSuggestions(  
    Dim c As Integer = s.Count  
  
    ListBox2.Items.Clear()  
    If c > 0 Then  
        Dim i As Integer  
        For i = 1 To c  
            ListBox2.Items.Add(s.Item(i).Name)  
        Next
```

```
        Else
            ListBox2.Items.Add(''No spelling suggestions.'')
        End If
    End Sub

    Private Sub Form1_Closing(ByVal sender As Object, _
        ByVal e As System.ComponentModel.CancelEventArgs) Handles MyBase.Clc
        w.quit(False)
    End Sub
```

When the user clicks the button, the text in the TextBox is dumped into the document using the InsertAfter method. Then you create a ProofreadingErrors collection and assign to it any spelling errors within the range (which has expanded to embrace the text you dumped into the document). The collection is a set of strings, which you add to the ListBox using the FindStringExact method to ensure there aren't any duplicates.

Also available from Word is the WordSpellingSuggestions collection (one collection per misspelled word). When the user clicks on one of the misspelled words displayed in ListBox1, the WordSpellingSuggestions collection is provided, via the GetSpellingSuggestions method. ListBox2 is then filled with the collection. In a finished VB.NET application, you would permit the user to click one of the suggested spellings in ListBox2 and replace the misspelling in the TextBox with the correction.

Sending a Fax

You can borrow Word's Fax Wizard, which steps the user through the process of sending a fax (based on either the current document in Word, or text the user wants to add during the Wizard process). If you want to permit the user to fax text contained in your VB.NET application—in a TextBox, for example—use the code in the previous example to transfer the text from the TextBox to the Clipboard, and thence to the Word document (see Listing 4.4).

LISTING 4.4: HOW TO SEND A FAX

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

    Dim w As Object = New Word.Application
    w.visible = False
    Dim d As Object = w.Documents.Add

    w.SendFax()

    w.Quit()

End Sub
```

Loading Documents

Many of the behaviors you can access in Office applications correspond to those applications' menu items. If you can see it listed in a menu, you can nearly always find a way to achieve it via VB.NET.

Say that you want to load the contents of a Word document into a TextBox in a VB.NET application. Here's one way to do that. This loads the most recently edited file. It corresponds to the list of recently opened documents found on the File menu in Word. With a button and TextBox on a form, type Listing 4.5 into the button's Click event.

LISTING 4.5: IMPORTING A WORD DOCUMENT

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

    Dim w As Object = New Word.Application
        w.visible = False

    Dim id As IDataObject

    Try

        w.RecentFiles(1).Open() 'get the most recently edited fil

        Dim s As String = w.recentfiles(1).name
        Me.Text = s

        'reference the word document most recently loaded
        Dim d As Object = w.Documents(1)

        With d
            .Content.Copy()
            id = Clipboard.GetDataObject
            TextBox1.Text = CType(id.GetData(DataFormats.Text), S
            .Close()
        End With

    Catch ex As Exception

        MsgBox(ex.ToString)

    End Try

    w.Quit() 'exit word

End Sub
```

Add print and preview capabilities to basic Window controls



PRNListView: Add preview and print capabilities to the ListView control (Chapter 7)



PRNTextBox Control, an enhanced TextBox control that provides its own Print and Preview methods. The control prints its text using the page settings provided by the user, using the control's Font and WordWrap properties (Chapter 7).

This page intentionally left blank.

Associate Publisher: Joel Fugazzotto
Acquisitions and Developmental Editor: Tom Cirtin
Production Editor: Leslie E.H. Light
Technical Editor: Greg Guntle
Copyeditor: Suzanne Goraj
Compositor: Maureen Forys, Happenstance Type-O-Rama
Graphic Illustrator: Jeffery Wilson, Happenstance Type-O-Rama
Proofreaders: Nancy Riddiough, Amey Garber, Emily Hsuan, Sarah Tannehill, and Laurie O'Connell
Indexer: Nancy Guenther
Book Designer: Maureen Forys, Happenstance Type-O-Rama
Cover Designer: Richard Miller, Calyx Designs
Cover Illustrator: Richard Miller, Calyx Designs

Copyright © 2004 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. The author(s) created reusable code in this publication expressly for reuse by readers. Sybex grants readers limited permission to reuse the code found in this publication or its accompanying CD-ROM so long as the author(s) are attributed in any application containing the reusable code and the code itself is never distributed, posted online by electronic transmission, sold, or commercially exploited as a stand-alone product. Aside from this specific exception concerning reusable code, no part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic, or other record, without the prior agreement and written permission of the publisher.

Library of Congress Card Number: 2003109130

ISBN: 0-7821-4242-7

SYBEX and the SYBEX logo are either registered trademarks or trademarks of SYBEX Inc. in the United States and/or other countries.

Screen reproductions produced with FullShot 99. FullShot 99 © 1991-1999 Inbit Incorporated. All rights reserved.

FullShot is a trademark of Inbit Incorporated.

Internet screen shots using Microsoft Internet Explorer reprinted by permission from Microsoft Corporation.

TRADEMARKS: SYBEX has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer.

The author and publisher have made their best efforts to prepare this book, and the content is based upon final release software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturer(s). The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

[Team Fly](#)

 Previous

Next 

Software License Agreement: Terms and Conditions

The media and/or any online materials accompanying this book that are available now or in the future contain programs and/or text files (the "Software") to be used in connection with the book. SYBEX hereby grants to you a license to use the Software, subject to the terms that follow. Your purchase, acceptance, or use of the Software will constitute your acceptance of such terms.

The Software compilation is the property of SYBEX unless otherwise indicated and is protected by copyright to SYBEX or other copyright owner(s) as indicated in the media files (the "Owner(s)"). You are hereby granted a single-user license to use the Software for your personal, noncommercial use only. You may not reproduce, sell, distribute, publish, circulate, or commercially exploit the Software, or any portion thereof, without the written consent of SYBEX and the specific copyright owner(s) of any component software included on this media.

In the event that the Software or components include specific license requirements or end-user agreements, statements of condition, disclaimers, limitations or warranties ("End-User License"), those End-User Licenses supersede the terms and conditions herein as to that particular Software component. Your purchase, acceptance, or use of the Software will constitute your acceptance of such End-User Licenses.

By purchase, use or acceptance of the Software you further agree to comply with all export laws and regulations of the United States as such laws and regulations may exist from time to time.

REUSABLE CODE IN THIS BOOK

The author(s) created reusable code in this publication expressly for reuse by readers. Sybex grants readers limited permission to reuse the code found in this publication, its accompanying CD-ROM or available for download from our website so long as the author(s) are attributed in any application containing the reusable code and the code itself is never distributed, posted online by electronic transmission, sold, or commercially exploited as a stand-alone product.

SOFTWARE SUPPORT

Components of the supplemental Software and any offers associated with them may be supported by the specific Owner(s) of that material, but they are not supported by SYBEX. Information regarding any available support may be obtained from the Owner(s) using the information provided in the appropriate read.me files or listed elsewhere on the media.

Should the manufacturer(s) or other Owner(s) cease to offer support or decline to honor any offer, SYBEX bears no responsibility. This notice concerning support for the Software is provided for your information only. SYBEX is not the agent or principal of the Owner(s), and SYBEX is in no way responsible for providing any support for the Software, nor is it liable or responsible for any support provided, or not provided, by the Owner(s).

WARRANTY

SYBEX warrants the enclosed media to be free of physical defects for a period of ninety (90) days after purchase. The Software is not available from SYBEX in any other form or media than that enclosed herein or posted to www.sybex.com. If you discover a defect in the media during this warranty period, you may obtain a replacement of identical format at no charge by sending the defective media, postage prepaid, with proof of purchase to:

SYBEX Inc.
Product Support Department
1151 Marina Village Parkway
Alameda, CA 94501
Web: <http://www.sybex.com>

After the 90-day period, you can obtain replacement media of identical format by sending us the defective disk, proof of purchase, and a check or money order for \$10, payable to SYBEX.

DISCLAIMER

SYBEX makes no warranty or representation, either expressed or implied, with respect to the Software or its contents, quality, performance, merchantability, or fitness for a particular purpose. In no event will SYBEX, its distributors, or dealers be liable to you or any other party for direct, indirect, special, incidental, consequential, or other damages arising out of the use of or inability to use the Software or its contents even if advised of the possibility of such damage. In the event that the Software includes an online update feature, SYBEX further disclaims any obligation to provide this feature for any specific duration other than the initial posting.

The exclusion of implied warranties is not permitted by some states. Therefore, the above exclusion may not apply to you. This warranty provides you with specific legal rights; there may be other rights that you may have that vary from state to state. The pricing of the book with the Software by SYBEX reflects the allocation of risk and limitations on liability contained in this agreement of Terms and Conditions.

SHAREWARE DISTRIBUTION

This Software may contain various programs that are distributed as shareware. Copyright laws apply to both shareware and ordinary commercial software, and the copyright Owner(s) retains all rights. If you try a shareware program and continue using it, you are expected to register it. Individual programs differ on details of trial periods, registration, and payment. Please observe the requirements stated in appropriate files.

COPY PROTECTION

The Software in whole or in part may or may not be copy-protected or encrypted. However, in all cases, reselling or redistributing these files without authorization is expressly forbidden except as specifically provided for by the Owner(s) therein.

To Nefeli
—Evangelos Petroustos

To David Lee and Cliff
—Richard Mansfield

Using the Old-Style Double DateTime Data Type	39
Finding Days in a Month	39
File I/O (Streaming)	39
Reading a File	40
How Do You Know You're at the End?	41
Writing to a File	42
Form References: Communication Between Forms	45
Using Handles	46
Runtime Handles	47
Detecting Key Presses	47
Loading Graphics with LoadPicture	48
Managing the Registry	48
Writing to the Registry	49
Random Numbers	49
Filling an Array	52
SendKeys	53
Serializing	53
Mixing Types into the Same Stream	55
Reading Back Mixed Data	56
Summary	57
Chapter 3 • Serialization Techniques	59
How Serialization Works	60
Serialization Types	60
Basic Serialization	61
Creating Serializable Objects	65
The ClassSerializer Project	66
XML Serialization	72
The NETConfigFiles Project	76
Custom Serialization	79

The SerializationContext Parameter	80
Serializing SQL Server Data	81
The NWOrders Project	86
Summary	91
Chapter 4 • Leveraging Microsoft Office in Your Applications	93
Using Word's Features	94
Spell-Checking	94
Sending a Fax	98
Loading Documents	99
Finding Files	100
Feeding Individual Strings and Specialized Formatting	102
Text Manipulation and Insertion	104
Replacing Text	106
Borrowing Word's Printing Features	107
Using Outlook Objects	109

Accessing Excel	111
Evaluating Math Expressions	111
Pretty Printing	113
Sending Data to Excel, Formatting, Calculating, and Saving	113
Retrieving Data from Excel	115
Summary	117
Chapter 5 • Understanding .NET Security	119
Security: An Overview	120
.NET's Strong Features	121
Users and Groups	122
The Principal	122
Code-Access Security	124
Everyone Must Agree	124
Understanding Code-Access Security	125
CAS Config Files	127
Descriptors	127
Software Restriction Policy	127
Managing .NET Security Policy	128
Using the Framework Configuration Tool	129
Programming for Security	133
Summary	136
Chapter 6 • Encryption, Hashing, and Creating Keys	139
The Main Problem	139
Hashing a Password	140
Hashing a File	142
Encrypting	143
Understanding Initialization Vectors	146
Discovering Key Sizes	147
Hashing while Encrypting	147

Asymmetrical Encryption	151
How RSA Works	153
Encrypting and Decrypting Using RSA	156
Summary	158
Chapter 7 • Advanced Printing	159
Printing in .NET	159
Printer and Page Properties	162
The PageSettings Object	162
The PrinterSettings Object	163
The Printing Dialog Boxes	164
Page Layout and Printing	168
The DrawString and MeasureString Methods	168
Printing Plain Text	174
Printing Tabular Data	179
A PrintScreen Utility	186
Summary	189

Chapter 8 • Upon Reflection	191
What Use Is It?	191
Understanding Types	191
Getting a Grip on Assemblies	192
Containers within Containers	193
Security Issues	193
Seeing Reflections	194
Accessing a Type	194
Accessing the Current Project's Assembly	198
Accessing a Loaded Assembly	199
Loading a File from an Assembly	200
Loading an Assembly from a File	201
Getting the Methods in a Class	203
More about Types	205
Accessing Specific Members	206
Searching for Members or Data	207
Executing Discovered Code with CreateInstance and Invoke	208
Emission	213
Summary	214
Chapter 9 • Building Bug-Free and Robust Applications	215
Structured Exception Handling	216
The Finally Clause	219
The ReadWriteFile Project	219
Resuming Statements That Failed	224
The Exception Class	225
Throwing Custom Exceptions	228
Bypassing Error Handlers	230
Debugging Techniques	231
Types of Programming Errors	232

Dealing with Logic Errors	237
Summary	241
Chapter 10 • Deploying Windows Applications	243
Installing the .NET Framework Runtime	244
XCopy Deployment	245
Internet Deployment	246
Preparing for Internet-Based Deployment	247
Deploying a Windows Application on a Web Server	249
Code Access Permissions	251
Running the Application	257
Downloading Assemblies on Demand	258
Deploying with Windows Installer	259
Creating a Windows Installer Package	261
Using the File System Editor	263
Creating Shortcuts	266

The Registry Editor	267
Using the User Interface Editor	267
Summary	270
Chapter 11 • Building Data-Driven Web Applications	271
New Features in ASP.NET	271
Sending Entire Files	272
Using Server Controls	272
Displaying Data on a WebForm	273
The DataList, Repeater, and Templates	275
Using the DataGrid	276
Detecting Postback	281
Validation	282
Programmatic Validation	282
Validation Controls	283
Sending Graphics	286
Using HTML Controls	287
Summary	288
Chapter 12 • Peer-to-Peer Programming	289
Internet Addressing	289
Using Sockets	292
Using UDP Sockets	295
Using TCP Sockets	297
The TCPChat Application	300
The TCPChatServer Application	301
The TCPChatClient Application	305
Interacting with Web Resources	308
Downloading Documents with WebClient	311
Uploading Documents with WebClient	312
The WebRequest and WebResponse Classes	314
Summary	317

Chapter 13 • Advanced Web Services	319
What Are Web Services?	319
Creating a Web Service	320
Caching Web Service Data	322
Consuming a Web Service	323
Preserving State	325
Using Session State	326
Making a Database Connection	326
Using the Pubs Sample Database	327
Getting an XML Dataset	327
Potential Problems with MSDE	329
Looking at the Results	329
Implementing WSDL	330
Viewing WSDL	331

SOAP Too	333
Complex Types	333
PortType	335
Seeing SOAP, WSDL, and the Reference Map	336
UDDI: The Registry	336
Testing a Published Web Service	337
Security Considerations	339
Summary	339
Chapter 14 • Building Asynchronous Applications with Message Queues	341
Queues and Messages	342
Types of Queues	343
Creating New Queues	344
Administering Queues	345
The MessageQueue Class	347
Exploring a Computer's Queues	348
The Message Class	349
Message Properties	350
Creating and Sending Messages	352
Acknowledgments and Time-Outs	358
Requesting Message Acknowledgment	358
Processing Acknowledgment Messages	361
Fault Tolerance and Load Balancing	366
Transactional Messages	371
Processing Orders with Messages	373
Preparing Orders	375
Processing Orders	377
Message Queuing Triggers	382
Defining Rules	382
Defining Triggers	384

The ProcessOrders Console Application	385
Summary	388
Chapter 15 • Practical ADO.NET	391
Accessing Databases	391
The Visual Database Tools	392
The Connection Class	402
The DataAdapter Class	404
The Command Class	409
Working with DataSets	415
Accessing the DataSet's Tables	416
Working with Rows	417
Handling Null Values	418
Adding and Deleting Rows	419
Locating Rows	420
Navigating through a DataSet	421
Using DataViews	426

Insert and Update Operations	428
Updating the Database with the DataAdapter	428
Handling Identity Columns	430
Performing Transactions with the DataAdapter	436
Summary	440
Chapter 16 • Building Middle-Tier Components	441
From Client/Server to Multiple Tiers	441
What Exactly Is a Business Rule?	443
Designing with Middle-Tier Components	445
Remoting the Business Logic	449
Converting the BusinessLayer Class to a Web Service	450
Converting the BusinessLayer to a Remote Service	458
Using COM Components with .NET Clients	461
Using ActiveX Controls in .NET	462
Using COM+ Applications in .NET	465
The COMPlus Component	466
Exporting a Proxy and Testing It	467
Building Serviced Components with .NET	468
Summary	473
Chapter 17 • Exploring XML Techniques	475
Choosing SAX	476
Copying the Sample File	476
Using SAX	476
Deeper into DOM	478
Using Namespaces in XML	480
Explicit Declaration	480
Implicit Declaration	480
The Explosion of Schemes	481
Understanding XSD	481

Using XML Data Types	483
Programmatic XML	487
Edit and Save	488
A Recursive Walk through the Nodes	491
XML and DataSets	493
Persisting with SOAP	495
Mixing and Matching Types	497
Deserialization Trapping	501
More Interchangeability	503
Summary	504
Chapter 18 • Designing Data-Driven Windows Applications	505
Data Binding	505
The NWProducts Application	506
The Application's Interface	507
The Application's Architecture	508
The Application's Code	510

An Invoicing Application	516
The Application's Interface	516
The Application's Architecture	518
The Application's Code	525
Adding a Business Rule	532
The Relations Application	535
The Application's Architecture	535
The Application's Code	536
The Relations1 Project	539
Summary	542
Chapter 19 • Working with Regular Expressions	543
Writing Regular Expressions	544
The RegularExpressions Class	547
Using the Matches Method	550
Using the Match and NextMatch Methods	551
The Split Method	552
The Replace Method	552
The Elements of a Regular Expression	558
Characters and Metacharacters	558
Single Character Metacharacters	558
Ranges of Characters	559
White Space and Metacharacters	560
Quantifiers	560
Anchors	562
Escaping Metacharacters	563
Alternation	563
The RegExEditor Project	564
Advanced Topics in Regular Expressions	567
Grouping and Back-References	568
Regular Expressions with Multiple Captures	573

Lookahead and Lookbehind Assertions	575
Advanced Replacement Operations	578
The RegularExpressions Project	579
The Visual grep Project	582
Summary	588
Chapter 20 • Advanced Graphics	589
The PlotControl	590
The GraphicsPath Object	591
The Control's Members	591
Drawing the Grid	598
A Fractal Generator	602
What Is a Fractal?	602
The Mandelbrot Set	607
The Julia Set	612
The Real Magic of Fractals	616

Complex Number Operations	620
The Transformation $z = z^2 + c$	621
Summary	622
Chapter 21 • Designing the User Interface	623
Making Applications Look Reliable	623
Windows Conventions	626
The Metallic Look	626
FontBold Off	627
Using a Sans-serif Typeface for Headlines	627
Choosing a Type Size	627
Layering	627
Adding Depth	627
Light from the Upper Left	628
Creating Zones	629
Framing	630
Metallic Shading	635
Sliding and Fading Transitions	636
Fade In, Fade Out	637
Sliding	640
Summary	641
Chapter 22 • Using the .NET Compact Framework and Its Emerging Technologies	643
What's Eliminated?	644
Output Lite	644
Solving the Connectivity Problem	644
Using the Simulator	645
Understanding the Mobile Form	646
Navigating to a Second Form	646
More New Features	647
New Technology, New Behaviors	648

The List Controls	650
Mobile Security	652
Debugging via Tracing	654
Custom Tracing	655
Trace Information Sections	656
Providing Friendly Error Messages	656
Device Specificity	658
Using Emulators	660
Custom Device Emulators	660
The Visual Studio 2003 Pocket PC Emulator	661
New Technology, New Problems	662
Summary	663
<i>Index</i>	665

This page intentionally left blank.

You'll find a variety of useful utilities throughout the book, including DES and RSA encryption systems in Chapter 6 that you can plug into your projects to protect the privacy of any kind of data. The chapter on Office automation, Chapter 4, demonstrates how to add a variety of utilities to your VB.NET programs: extracting statistical information such as word or paragraph counts; evaluating math expressions; spell checking; automatically retrieving e-mail; filtering and displaying e-mail; searching directories and subdirectories for specific files; importing data from or exporting data to Word, Excel or Outlook; and sending faxes.

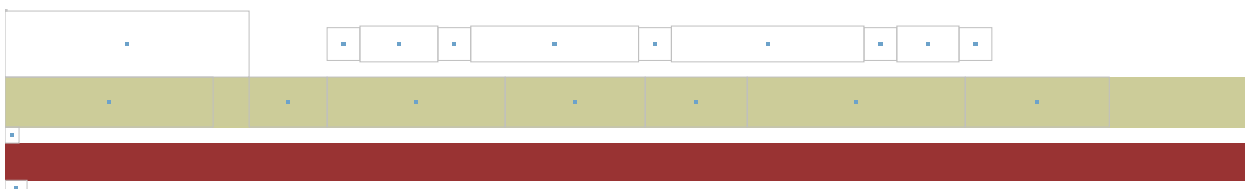
Chapter 5 takes you on a tour of the Windows and .NET multi-layered security maze—all the various secret keyholes and locks that must be correctly set before even a single function can execute. You'll see how to manage code-based (.NET), role-based (Windows), and miscellaneous (IIS, database-specific, and so on) security systems. Did you know that you can specify security with such great specificity that you can grant or deny permission for each individual procedure—or even for each individual line of code—within your VB.NET projects?

Chapter 8 explores the new technology called *reflection* with which you can extract information from self-describing assemblies (libraries of code that contain descriptions of their contents). You're shown how you can use this information in specialized, though practical, applications, and how you can even go so far as to *emit* (write code-generating code).

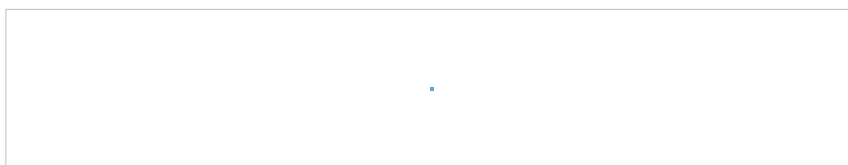
Forms design is the topic of Chapter 21: What you can do to make the appearance of your VB.NET programs more professional, polished, and ergonomic. This chapter covers one of the most overlooked aspects of program design—the *design* itself properly so called, the actual *look* of the finished application on its surface—where your programming logic rises and becomes visible to your users. Among the techniques explained in this chapter are light sources, metallic surfaces, fonts, layering, depth, framing, shading, gradients, and transitions. In Chapters 11, 13, and 18 you'll find largely hype-free coverage of programming data-driven Web applications, Web services, and XML, respectively.

Although you will find a book on every aspect of ADO.NET in the market, we've included two chapters on the topic of database programming: an overview of the ADO.NET object model with simple examples and a chapter with practical data-driven applications. One of the examples is an invoicing application, which is a fundamental component of every business. Besides sending XML data to Japan, or exchanging data with a database residing on a satellite, a company may need to sell products and services, and for this you'll need a functional application for preparing orders and invoices. It's a humble task, but too important to be skipped in a practical chapter on data-driven applications. You have certainly read a lot about the middle tier of a data-driven application. If you understand what a middle tier is, you can skip our discussion on developing middle-tier components. If you need a simple example of a business rule, how to implement business rules as middle-tier components, and how to deploy middle-tier components so that you can change the business rule without touching the code of the application that has already been deployed to the users' workstations, then explore our examples in Chapters 16 and 18.

We've also included a few useful tools, which you can use in your projects with little or no customization. The PRNTextBox and PRNListView controls are enhanced versions of the TextBox and ListView controls that provide methods to print their contents. Regular expressions are not the bread and butter of the typical VB developer, and this is the reason most books totally ignore this topic. To demonstrate regular expressions in Chapter 19, we've included the RegExEditor: a simple text editor that allows you to search text files using general search patterns, such as e-mail addresses, dollar amounts, and so on. In Chapter 20 we've included two graphics applications, one for plotting functions—a practical control you can use in any number-processing application—and a fractal application—a program that generates fascinating patterns for your amusement. All of the utilities are available at www.sybex.com.



You are here: home



We offer the only comprehensive approach to eBooks that integrates with the time-honored missions and methods of libraries and librarians. Our vision is one of enhancing the role of librarians as stewards of knowledge, supporting their crucial role in serving millions of people every day who seek information.

