SYNGRESS®

1 YEAR UPGRADE
BUYER PROTECTION PLAN

SYNGRESS.COM 1 YEAR UPGRADE BUYER PROTECTION PLAN

VB.NET

Developer's Guide

Develop and Deliver Enterprise-Critical Desktop and Web Applications with VB.NET

- Step-by-Step Instructions for Installing and Configuring Visual Basic .NET and Visual Studio .NET
- Hundreds of Developing & Deploying and Debugging Sidebars, Security Alerts, and VB.NET FAQs
- Complete Coverage of the New Integrated Development Environment (IDE)

Cameron Wakefield
Henk-Evert Sonder
Wei Meng Lee Series Editor

SYNGRESS® WALLET CD
YOUR COMPLETELY PORTABLE SOLUTION!

Global Knowledge
RECOMMENDED READING

SYNGRESS®

1 YEAR UPGRADE
BUYER PROTECTION PLAN

SYNGRESS.COM 1 YEAR UPGRADE BUYER PROTECTION PLAN

VB.NET
Developer's Guide

Cameron Wakefield
Henk-Evert Sonder
Wei Meng Lee Series Editor

KEY	**SERIAL NUMBER**
001	DL84T9FVT5
002	ASD524MLE4
003	VMERL3FG4R
004	SGD34WR75N
005	8LUVCX5N7H
006	NZSJ9NTEM4
007	BWUH5MR46T
008	2AS3R565MR
009	8PL8Z4BKAS
010	GT6Y7YGVFC

PUBLISHED BY
Syngress Publishing, Inc.
800 Hingham Street
Rockland, MA 02370

VB.NET Developer's Guide

Printed in the United States of America

1 2 3 4 5 6 7 8 9 0

ISBN: 1-928994-48-2

Technical Editor: Cameron Wakefield
Series Editor: Wei Meng Lee
Co-Publisher: Richard Kristof
Acquisitions Editor: Catherine B. Nolan
Developmental Editor: Jonathan Babcock
Freelance Editorial Manager: Maribeth Corona-Evans
Cover Designer: Michael Kavish
Page Layout and Art by: Shannon Tozier
Indexer: Robert Saigh
CD Production: Michael Donovan

Distributed by Publishers Group West in the United States and Jaguar Book Group in Canada.

Acknowledgments

We would like to acknowledge the following people for their kindness and support in making this book possible.

Richard Kristof and Duncan Anderson of Global Knowledge, for their generous access to the IT industry's best courses, instructors, and training facilities.

Ralph Troupe, Rhonda St. John, and the team at Callisma for their invaluable insight into the challenges of designing, deploying, and supporting world-class enterprise networks.

Karen Cross, Lance Tilford, Meaghan Cunningham, Kim Wylie, Harry Kirchner, Kevin Votel, Kent Anderson, and Frida Yara of Publishers Group West for sharing their incredible marketing experience and expertise.

Mary Ging, Caroline Hird, Simon Beale, Caroline Wheeler, Victoria Fuller, Jonathan Bunkell, and Klaus Beran of Harcourt International for making certain that our vision remains worldwide in scope.

Anneke Baeten and Annabel Dent of Harcourt Australia for all their help.

David Buckland, Wendi Wong, Daniel Loh, Marie Chieng, Lucy Chong, Leslie Lim, Audrey Gan, and Joseph Chan of Transquest Publishers for the enthusiasm with which they receive our books.

Kwon Sung June at Acorn Publishing for his support.

Ethan Atkin at Cranbury International for his help in expanding the Syngress program.

Contributors

Todd Carrico (MCDBA, MCSE) is a Senior Database Engineer for Match.com. Match.com is a singles portal for the digital age. In addition to its primary Web site, Match.com also provides back-end services to AOL, MSN, and many other Web sites in its affiliate program. Todd specializes in design and development of high-performance, high-availability data architectures primarily on Microsoft technology. His background includes designing, developing, consulting, and project management for companies such as Fujitsu, Accenture, International Paper, and GroceryWorks.com. Todd resides in Sachse, TX.

Mark Horninger (A+, MCSE+I, MCSD, MCDBA) is President and founder of Haverford Consultants Inc. (www.haverford-consultants.com), located in the suburbs of Philadelphia, PA. He develops custom applications and system engineering solutions, specializing primarily in Microsoft operating systems and Microsoft BackOffice products. He has over 10 years of computer consulting experience and has passed 29 Microsoft Certified exams. During his career, Mark has worked on many extensive projects including database development, application development, training, embedded systems development, and Windows NT and 2000 project rollout planning and implementations. Mark lives with his wife Debbie and two children in Havertown, PA.

Tony Starkey is the Lead Software Developer for Lufkin Automation in Houston, TX and is currently in charge of revamping, restructuring, and redesigning, their award-winning, well analysis programs. He also provides consulting services to other companies in the city. Tony specializes in Visual Basic, VBScript, ASP, and GUI design. He has been the head developer on several projects that have seen successful completion through all cycles of software design. Tony holds a bachelor's degree in Computer Science from the University of Houston with a minor in Mathematics. He is a highly respected expert in numerous online developer communities, where he has offered in excess of 3,000 validated solutions to individuals, ranging from the novice to the experienced Microsoft Certified Professional.

Henk-Evert Sonder (CCNA) has over 15 years of experience as an Information and Communication Technologies (ICT) professional, building and maintaining ICT infrastructures. In recent years, he has specialized in integrating ICT infrastructures with secure business applications. Henk's company, IT Selective, works with small businesses to help them develop high-quality, low cost solutions. Henk has contributed to several Syngress books, including the *E-Mail Virus Protection Handbook* (ISBN: 1-928994-23-7), *Designing SQL Server 2000 Databases for .NET Enterprise Servers* (ISBN: 1-928994-19-9), and the upcoming book *BizTalk Server 2000 Developers Guide for .NET* (ISBN: 1-928994-40-7). Henk lives in Hingham, MA with his wife Jude and daughter Lilly.

Jonothon Ortiz is Vice President of Xnext, Inc. in Winter Haven, FL. Xnext, Inc. is a small, privately owned company that develops Web sites and applications for prestigious companies such as the New York Times. Jonothon is the head of the programming department and works together with the CEO on all company projects to ensure the best possible solution. Jonothon lives with his wife Carla in Lakeland, FL.

Prasanna Pattam is an Internet Architect for Qwest Communications. He is responsible for the overall architecture, design, development, and deployment of the multi-tiered Internet systems using Microsoft Distributed interNet Application Architecture. His expertise lies in developing scalable, high-performance enterprise Web solutions for Fortune 500 companies. At Qwest, Prasanna has helped to formalize methodologies, development standards, and best coding practices, as well as to mentor other developers. He has written technical articles for different Web sites and also teaches advanced e-commerce courses. Prasanna holds a master's degree in Computer Science. He resides in Fairview, NJ.

Mike Martone (MCSD, MCSE, MCP+Internet, LCNAD) is a Senior Software Engineer and Consultant for Berish & Associates (www.berish.com), a Cleveland-based Microsoft Certified Solutions Provider, Partner Level. In 1995, Mike became one of the first thousand MCSDs and is certified in VB 3, 4, and 5. Since graduating from Bowling Green State University with degrees in Computer Science and Psychology, he has specialized in developing Visual Basic, Internet, and Office applications for corporations and government institutions. Mike has contributed to several study guides on Visual Basic and SQL 7 in the best-selling certification series from Syngress. He lives in Lakewood, OH.

Robeley Carolina (MCP) is a Senior Engineer with Computer Science Innovations, where his specialties include user interface design and development. He has also served on the faculties of the Florida Institute of Technology and Herzing College, teaching numerous mathematics and computer science courses. Robley holds a bachelor's degree in Mathematics and a master's degree in Management from the Florida Institute of Technology. Robley currently resides in Palm Bay, FL and would like to thank Pamela for her support.

Rick DeLorme (MCP) is a Software Consultant in Ottawa, Ontario, Canada. He currently works for a small company developing logistics applications with Visual Basic 6. He has worked on other large-scale projects such as the Canadian Census of Population where we worked with VB6, MTS, DCOM, and SQL Server. He is currently working towards his MCSD. Rick would like to thank his fiancé Jenn for her encouragement and support.

Narasimhan Padmanabhan (MCSD) is a software consultant with a major software company. His current responsibilities include developing robust testing tools for software. He holds a bachelors degree in Commerce and is an application developer for ERP applications back home in India. He lives with his wife Aarthi and daughter Amrita in Bellvue, WA.

Technical Editor and Contributor

Cameron Wakefield (MCSD, Network+) is a Senior Engineer at Computer Science Innovations, Inc. headquartered in Melbourne, FL (www.csi.cc). CSI provides automated decision support and custom data mining solutions. Cameron develops custom software solutions ranging from satellite communications to data mining applications. He is currently working on a neural network-based network intrusion detection system. His development work spans a broad spectrum including C/C++, Visual Basic, COM, ADO, SQL, ASP, Delphi, CORBA, and UNIX. Cameron has developed a variety of Web applications including online trading systems and international gold futures site. Cameron has passed 10 Microsoft certifications and teaches Microsoft and Network+ certification courses at Herzing College (AATP). Cameron has contributed to a number of Syngress books including *Designing SQL Server 2000 Databases for .NET Enterprise Servers* (ISBN: 1-928994-19-9) and several MCSE and MCSD study guides.

Cameron holds a bachelor's of science degree in Computer Science with a minor in Mathematics at Rollins College and is a member of IEEE. He currently resides in his new home in Rockledge, FL with his wife Lorraine and daughter Rachel.

Series Editor

Wei Meng Lee is Series Editor for Syngress Publishing's .NET Developer Series. He is currently lecturing at The Center for Computer Studies, Ngee Ann Polytechnic, Singapore. Wei Meng is actively involved in Web development work and conducts training for Web developers and Visual Basic programmers. He has co-authored two books on WAP. He holds a bachelor's of science degree in Information Systems and Computer Science from the National University of Singapore.

About the CD

This CD-ROM contains the code files that are used in each chapter of this book. The code files for each chapter are located in a directory. For example, the files for Chapter 9 can be found in Chapter 09/Chapter9 Beta2/Samples/XML/MyData.xsd. The organizational structure of these directories varies. For some chapters, the files are named by a number. In other chapters, the files are organized by the projects that are presented within the chapter.

Chapters 4 and 5 contain sample code. These are not standalone applications, just examples. Chapter 4 contains code samples for performing File I/O, using the System.Drawing namespace for graphics and printing. Chapter 5 contains code samples for working with classes, string manipulation, and exception handling.

Chapter 6 contains the source files for two complete applications: one for performing a simple draw command and one for using C# classes. Chapter 9 contains the source code for several applications demonstrating how to use ADO.NET including: using a Typed Data Set and using data controls. It also contains sample XML and XSD dataset files.

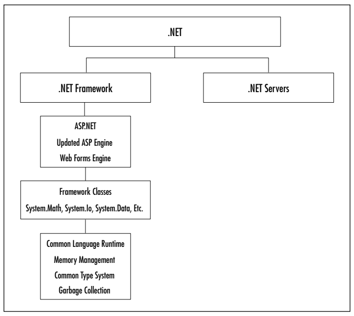
Chapter 10 contains the source code for exercises that demonstrate how to create Web applications. Most of these exercises build on each other. You will build a Web form, then put controls on it. You will see how to use a DataGrid control on a Web form. Then you will see how to use custom controls. Starting with Exercise 10.8, you will create and use a Web service and in Exercise 10.11 you will create a sample application.

Chapter 11 contains a sample calculator application to demonstrate debugging and testing tools built into Visual Basic .NET. Chapter 12 contains a sample Digital certificate for Web applications and a sample configuration file with security policies. And lastly, Chapter 14 contains code for the ICalculator interface.

Look for this CD icon to obtain files used in the book demonstrations.

Contents

.NET Architecture



NOTE

Visualization is still key! Die-hard VB programmers may find themselves having a hard time visualizing all the new concepts in VB.NET (and we all know that proper logic visualization plays a big role in what we do). Something that may help is to think about VB.NET as a completely flexible language that can accommodate Web, console, and desktop use.

Installing Visual Studio .NET

- Phase 1: Installing Windows components
- Phase 2: Installing Visual Studio .NET
- Phase 3: Checking for service releases

Developing & Deploying…

Embrace Your Parameters
VB.NET is insistent upon enclosing parameters of function calls within parentheses regardless of whether we are returning a value or whether we are using the Call statement. It makes the code much more readable and is a new standard for VB programmers that is consistent with the standard that nearly all other languages adopted long ago.

NOTE

When porting Visual Basic applications to Visual Basic .NET, be careful of the lower bounds of arrays. If you are using a for loop to iterate through the array, and it is hard-coded to initialize the counter at 1, the first element will be skipped. Remember that all arrays start with the index of 0.

What Are Collections?

Collections are groups of like objects. Collections are similar to arrays, but they don't have to be redimensioned. You can use the *Add* method to add objects to a collection. Collections take a little more code to create than arrays do, and sometimes accessing a collection can be a bit slower than an array, but they offer significant advantages because a collection is a group of objects whereby an array is a data type.

Creating Dialog Boxes

1. Create a form.
2. Set the **BorderStyle** property of the form to **FixedDialog**.
3. Set the **ControlBox**, **MinimizeBox**, and **MaximizeBox** properties of the form to **False**.
4. Customize the appearance of the form appropriately.
5. Customize event handlers in the Code window appropriately.

Adding Items to a Combo Box at Design-Time

1. Select the **ComboBox** control on the form.
2. If necessary, use the **View** menu to open the **Properties** window.
3. In the **Properties** window, click the **Items** property, then click the ellipsis.
4. In **String Collection Editor**, type the first item, then press **Enter**.
5. Type the next items, pressing **Enter** after each item.
6. Click **OK**.

XML Documents

XML documents are the heart of the XML standard. An XML document has at least one element that is delimited with one start tag and one end tag. XML documents are similar to HTML, except that the tags are made up by the author.

NOTE

Web form controls not only detect browsers such as Internet Explorer and Netscape, but they also detect devices such as Palm Pilots and cell phones and generate appropriate HTML accordingly.

What Are Watches?

Watches provide us with a mechanism where we can interact with the actual data that is stored in our programs at runtime. They allow us to see the values of variables and the values of properties on objects. In addition to being able to view these values, you can also assign new values.

Within the .NET Framework, Three Namespaces Involve Cryptography

1. *System.Security .Cryptography* The most important one; resembles the CryptoAPI functionalities.
2. *System.Security .Cryptography* .X509 certificates Relates only to the X509 v3 certificate used with Authenticode.
3. *System.Security .Cryptography.Xml* For exclusive use within the .NET Framework security system.

WARNING

You should under no circumstance edit the Security.config and Enterprise.config files directly. It is very easy to compromise the integrity of these files. Always use the Code Access Security Policy utility (caspol.exe) or the .NET Configuration tool; these will guard the integrity of the files and will also make a backup copy of the last saved version.

Avoiding Null Propagation

Null propagation means that if Null is used in an expression, the resulting expression is always *Null*. In previous versions of Visual Basic, the Null value disseminated throughout the expression.

From the Series Editor

2001 marks the 10th anniversary of Microsoft Visual Basic (VB). In May 1991, Microsoft introduced Visual Basic 1.0. Microsoft's plan was to use VB as a strategic tool to encourage developers to write Windows applications.

With VB, Windows application development was no longer restricted to a privileged few. Anybody with moderate programming capabilities was able to develop a Windows application by dragging and dropping controls onto a form. In contrast to the more prevalent C and C++ programmers who wrote obscure code, VB programmers concentrated on writing applications that were meant to be prototypes. It is perhaps this ease of use and simplicity of language that gave VB the name of "toy" language. This is not the case anymore.

VB has come a long way. Since version 1.0, it has evolved from a toy language to a full-fledged Object-Oriented programming language. Today, with VB you are able to do almost everything possible with other programming languages. VB is finally a true-blue Object-Oriented language.

Visual Basic, Today and Tomorrow—VB.NET

With the announcement of the Microsoft .NET Framework in 2000, Microsoft has firmly re-iterated its commitment to the Visual Basic language. With language features such as inheritance, structured exception handling, and parameterized constructors, Visual Basic programming has become more elegant, simplified, and maintainable.

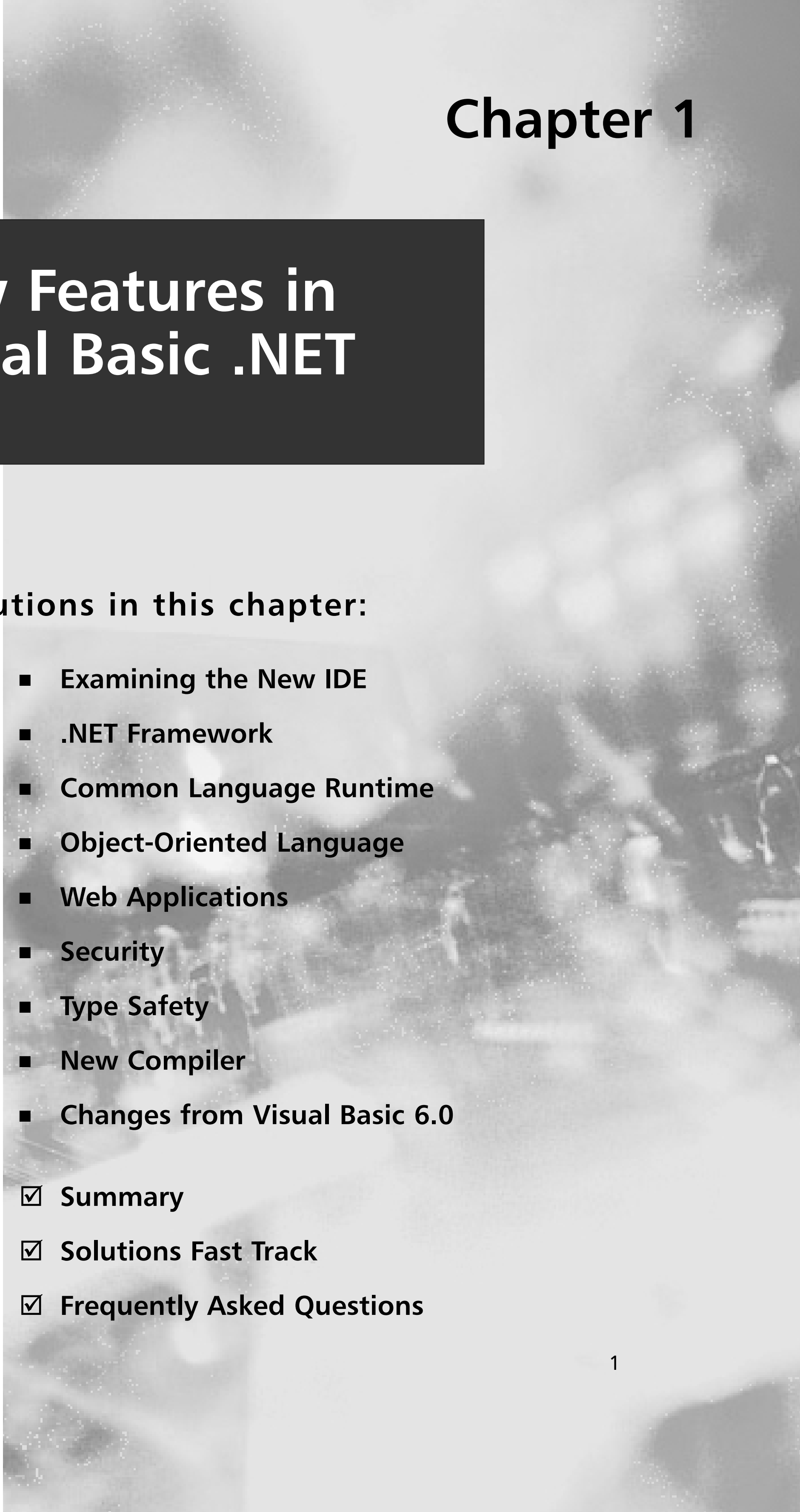
With Microsoft's vision of a programmable Web and its announcement of the .NET Framework and Visual Studio.NET, VB.NET is poised to become the most widely used language for developing Windows and Web applications.

The Syngress .NET Developer Series

VB.NET Developer's Guide, part of the Syngress .NET Developer Series, is written for Visual Basic programmers looking to harness the power of VB.NET's new features and functionality. Developers will appreciate the in-depth explanations of key concepts and extensive code examples. This practical, hands-on book will make you a productive VB.NET developer straight away!

I hope you will enjoy reading the book as much as the authors have enjoyed writing it.

—Wei Meng Lee
Series Editor, Syngress .NET Developer Series



Chapter 1

New Features in Visual Basic .NET

Solutions in this chapter:

- Examining the New IDE
- .NET Framework
- Common Language Runtime
- Object-Oriented Language
- Web Applications
- Security
- Type Safety
- New Compiler
- Changes from Visual Basic 6.0

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

Before we dig into the details of Visual Basic .NET, let's take a look at an overview of all the changes and new features. This new release is a significant change from the previous version. It will take some effort to get used to, but I am sure you will feel that the new features will make it worthwhile. Visual Basic .NET is more than just an upgrade from Visual Basic 6.0. As you would expect, the Integrated Development Environment (IDE) has been enhanced with some new features. All of the Visual Studio development tools will now share the same environment. For example, you will no longer need to learn a different IDE when switching between Visual Basic and Visual C++. Some nice features have been added that many of us have been asking for to ease development.

Visual Studio .NET is now built on top of the .NET Framework. This will be a significant change from Visual Basic 6.0. The .NET Framework takes application development to viewing the Internet as your new operating system. Your applications will no longer recognize hardware as a boundary limitation. This is an evolution of the Windows DNA model. This new framework is built on open Internet protocols for a standardized interoperability between platforms and programming languages. The .NET Framework will also allow the creation of new types of applications. Applications will now run using the Common Language Runtime (CLR). All .NET applications will use this same runtime environment, which allows your Visual Basic applications to run on equal ground with other languages. The CLR allows Visual Basic to provide inheritance and free threading, whose absence created two glaring limitations to Visual Basic Applications. Visual Basic .NET is object-oriented. Everything is now an object, and every object is inherited from a standard base class. Another benefit of the CLR is a common type system, which means that all programming languages share the same types. This greatly increases interoperability between languages.

The Internet has entered a new phase. First, it was used to display static Web pages. Businesses soon found that this did not help them significantly. Next, the Internet evolved to dynamic content and allowing electronic commerce. The next step is to move towards complete applications running on the Internet. Visual Basic .NET promotes these new Web applications. Web services allow objects to be located anywhere on the Internet and to be called from any application across the Internet (no more trying to get DCOM configured). Of course, extending applications across the Internet will increase security risks. The .NET Framework has many security features built-in to it to protect your applications.

Type safety is now enforced. This prevents code from accessing memory locations that it does not have authorization to access. This allows you to define how your objects are accessed. Before code is run, it is verified to be type-safe. If it is not type-safe, it will only run if your security policies allow for it.

Visual Basic has many new changes. This chapter gives you a high-level look at the overall architectural changes. This will help you dig into the details in the following chapters with an eye on the big picture.

Examining the New IDE

Whether you are a developer or a manager, you probably care more about how difficult the transition to this new environment will be than about every new feature. Microsoft shares your concerns. As you explore what VB.NET can offer, you will continually observe an intelligent blending of earlier versions of VB with features adapted from other languages. Nowhere is this clearer than in the IDE. Microsoft has added significant new functionality to make developers work more effectively, without requiring them to learn entirely new ways of doing their jobs.

If you have seen earlier versions of Visual Basic, the IDE for VB.NET will look very familiar. But if you have also worked with InterDev in the past, even more of the new interface will be old hat. That is because the new IDE used for VB.NET has integrated the best ideas from both environments to provide a more effective way of getting work done.

Of course, nothing comes without a cost. Some of the issues involved with this upgrade of VB are discussed later in this chapter and in the chapters to come, and these challenges must certainly be weighed when choosing a development tool. But first, we take a look at some of the specific new features in the IDE and the benefits they provide.

Cosmetic Improvements

Although numerous changes have been made to the IDE, the ones you will probably notice first are the cosmetic changes to existing functionality. Previous versions of Visual Basic have attempted to strike a balance between conserving screen real estate and providing one-click access to as much functionality as possible. Table 1.1 describes some of the ways that these tradeoffs have been addressed in VB.NET.

Table 1.1 Cosmetic Improvements

Feature	Description	Benefit
Multimonitor support	Developers can use more than one monitor for display at the same time.	By executing their code in one window and debugging in another, developers can more accurately simulate the experiences of the end user.
Tabbed forms	A tabbed layout is used to display the child MDI forms within the development environment. The code windows, Help screens, form layout windows, and home page all can be dragged on top of each other and displayed in the same pane.	Though you can't see as much information at once, you have the benefit of taking up less screen real estate.
Toolbox	Instead of displaying the controls in a grid, the controls are presented vertically, with a description next to each.	In previous versions of Visual Basic, you had to hover over the control to display the name of the control. (This was especially frustrating when you developed your own custom controls, because frequently they would all default to the same icon.)
Expandable code	Using an interface similar to Outline mode in Microsoft Word, you can now break your code into sections and conceal or expand each with a single click.	Developers now can keep a higher-level view on their code, allowing them to migrate through their application more efficiently.
Help	Instead of having to press F1, the .NET IDE now observes what you are doing and presents context-sensitive help in its own window.	Accurate guidance is now continuously available to your developers in real time.

Development Accelerators

Of course, not all of the new IDE features are simply cosmetic. The developers of VB.NET have also provided new interfaces to more efficiently use existing functionality. The features discussed in Table 1.2 all have clear predecessors in VB 6.0, but they now allow developers to more efficiently generate their applications.

Table 1.2 Development Accelerators

Feature	Description	Benefit
Menu Editor	Using the in-place Menu Editor, you now can edit menus directly on the associated form.	Previously, you had to choose the Menu Editor item from the Tools menu This change speeds up development and reduces errors associated with using the wrong form.
Solution Explorer	Unlike the Project Explorer provided in previous versions, the Solution Explorer provides a repository to view and maintain heterogeneous development resources.	You can now manage components that did not originate in VB. (The ability to make VB work better with other languages is one of the driving forces behind the .NET initiative.)
Server Explorer	Now you can see the servers available in a client/server or Internet app and directly incorporate their resources into your code.	What was formerly done manually now can be done using drag-and-drop. For example, if you have a stored procedure on a server in SQL, you can browse directly to the stored procedure and make the update on the page directly.
Home Page	The opening screen that appears when you launch VB is now created using DHTML.	You can now do more programming visually, reducing potential for error. For example, if you have a stored procedure in SQL Server, you could browse directly to that stored procedure and drag it onto the needed pane. VB does the rest of the coding automatically.

.NET Framework

The best way to understand what .NET offers is to observe some of the limitations of its predecessors. In this section, we take a very brief and simplified look at the history of Microsoft component interaction and then a short look at the architecture.

A Very Brief and Simplified History

When Windows 3.0 was introduced, the initial method used for communicating across applications was Dynamic Data Exchange, or DDE. DDE was resource-intensive, inflexible, and prone to cause system crashes. Nonetheless, it worked acceptably on single machines, and for many years, many applications continued to use this approach to send messages between applications.

Over the years, Microsoft discouraged the use of DDE, and encouraged the use of the Common Object Model (COM) and Distributed COM (DCOM). COM was used for communication among Microsoft applications on a single machine, whereas DCOM was used to communicate with remote hosts.

Meanwhile, a consortium of allied vendors (including IBM, Sun, and Apple) were proposing an alternative approach to interhost communication called CORBA. Unlike COM, CORBA was much better at passing messages across different operating systems. Unfortunately, the protocol was resource-intensive and difficult to program, and its use never lived up to its promise.

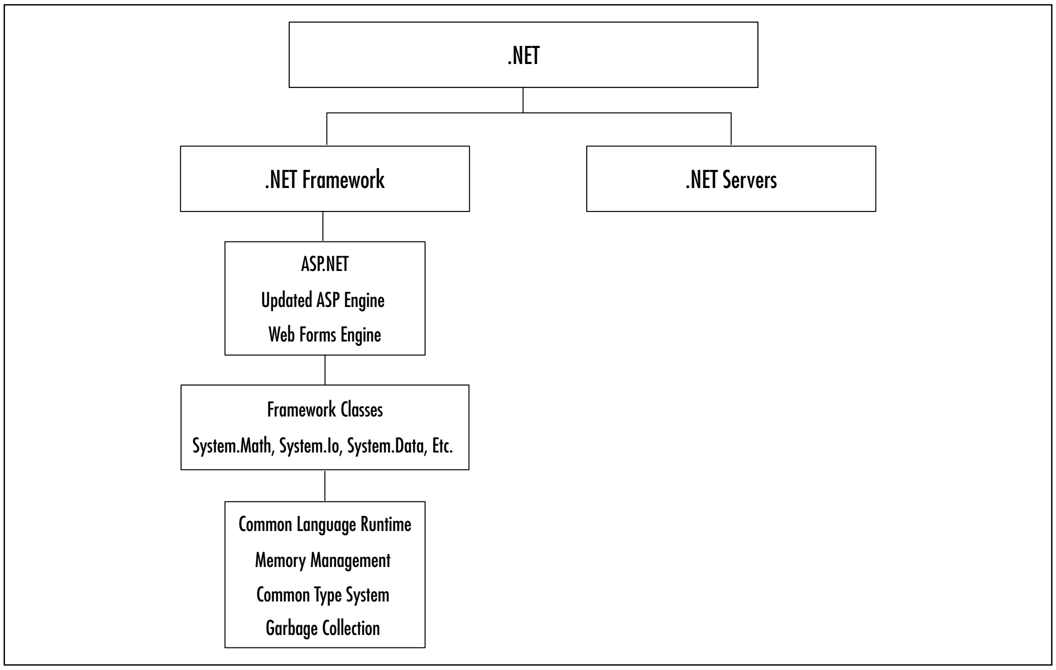
During this time, Microsoft was improving its technology, and they introduced COM+, Microsoft Transaction Server (MTS), and Distributed Network Architecture (DNA). These technologies allowed more sophisticated interactions among components, such as object pooling, events, and transactions. Unfortunately, these technologies required that each of the applications know a great deal about the other applications, and so they didn't work very well when the operating platforms were heterogeneous (for example, Windows apps communicating with Linux).

This brings us to the year 2001 and the .NET initiative, which combines the power of COM with the flexibility of CORBA. Although this technology is primarily associated with Microsoft, its flexibility and scalability means that theoretically it could be usable on other platforms in the future. (Although the .NET Framework runs on all Windows operating systems from Windows 95 on up, another version called the .NET Compact Framework is intended to run on Windows CE.)

.NET Architecture

The .NET Framework consists of three parts: the Common Language Runtime, the Framework classes, and ASP.NET, which are covered in the following sections. The components of .NET tend to cause some confusion. Figure 1.1 provides an illustration of the .NET architecture.

Figure 1.1 .NET Architecture



ASP.NET

One major headache that Visual Basic developers have had in the past is trying to reconcile the differences between compiled VB applications and applications built in the lightweight interpreted subset of VB known as VBScript. Unfortunately, when Active Server Pages were introduced, the language supported for server-side scripting was VBScript, not VB. (Technically, other languages could be used for server side scripting, but VBScript has been the most commonly used.)

Now, with ASP.NET, developers have a choice. Files with the ASP extension are now supported for backwards compatibility, but ASPX files have been introduced as well. ASPX files are compiled when first run, and they use the same syntax that is used in stand-alone VB.NET applications. Previously, many developers have gone through the extra step of writing a simple ASP page that simply executed a compiled method, but now it is possible to run compiled code directly from an Active Server Page.

Framework Classes

Ironically, one of the reasons that VB.NET is now so much more powerful is because it does so much less. Up through VB 6.0, the Visual Basic compiler had to do much more work than a comparable compiler for a language like C++. This is because much of the functionality that was built into VB was provided in C++ through external classes. This made it much easier to update and add features to the language and to increase compatibility among applications that shared the same libraries.

Now, in VB.NET, the compiler adopts this model. Many features that were formerly in Visual Basic directly are now implemented through Framework classes. For example, if you want to take a square root, instead of using the VB operator, you use a method in the *System.Math* class. This approach makes the language much more lightweight and scalable.

.NET Servers

We mention this here only to distinguish .NET servers from .NET Framework. These servers support Web communication but are not necessarily themselves written in the .NET Framework.

Common Language Runtime

CLR provides the interface between your code and the operating system, providing such features as Memory Management, a Common Type System, and Garbage Collection. It reflects Microsoft's efforts to provide a unified and safe framework for all Microsoft-generated code, regardless of the language used to create it. This chapter shows you what CLR offers and how it works—Chapter 4 covers it in much greater detail.

History

For years, the design of Visual Basic has reflected a compromise between power and simplicity. In exchange for isolating intermediate developers from the complexities and dangers of API programming, VB developers accepted certain limitations. The compiled VB code could not interact directly with the Windows API (usually written in C++), but instead they would interface through a runtime module that would handle the dirty work of data allocation and dereferencing.

Because of this situation, a gulf developed between VB and C++ programmers. In fact, many C++ programmers looked down at VB as merely suitable for Rapid Application Development and not as an appropriate tool for serious enterprise development. They also resented having to write wrappers to allow the VB developers to access new Windows APIs. This has all changed in VB.NET. Now, the code created by Visual Basic developers and C++ developers both interface with Windows in the same way—through the CLR. (For that matter, so do other new languages, such as C# or JavaScript.NET.)

Convergence

One of the advantages of VB.NET is that it is now possible to use VB to develop applications that previously needed to be developed in lower-level languages, without losing the traditional advantages of VB development. Whether you are a developer or a manager, your job involves analyzing the tradeoffs of the various tools available to better illustrate the convergence of these two platforms, Table 1.3 compares the ways in which VB and C handle four critical issues, both historically and in the .NET environment.

Table 1.3 VB and C Comparison

	VB 1.0–4.0	VB 5.0–6.0	VB.NET	C++	C#
Runtime Required?	Yes	Yes	No	No	No
Interface Model	COM	COM	CLR	COM	CLR
Memory Leaks?	Few	Few	Very few	Many	Very few
Inheritance Supported?	Yes	No	No	Yes	Yes

Runtime Required? Starting with VB 5.0, Microsoft made the claim that Visual Basic could actually compile to a true executable, but it is probably more accurate to say that the runtime module was just smaller and more transparent to the user. By contrast, C++ has never required a runtime module.

Interface Model With the CLR, the code compiled is no longer the exact code executed, but rather it is translated on the client machine. (Some of the advantages of this approach are described in more detail in the New Compiler section.) In previous versions of VB and C++, the code was compiled to use COM, but in VB.NET and C#, the code is compiled to CLR.

Memory Leaks? One of the traditional advantages of VB is that memory was managed responsibly by the compiled executable, and this advantage remains in VB.NET, although the work is now done in the CLR. (By contrast, poorly written C++ code often created these errors because memory was not deallocated after it was used.)

Inheritance Supported? This is probably the most important advance in VB.NET, and it is covered in the next section. (Starting with Version 5.0, VB supported a rough simulation of inheritance that is also described in the next section.)

Object-Oriented Language

Possibly the most valuable addition in VB.NET is true object orientation. Although approximations of object orientation have been available in earlier versions of Visual Basic, only in VB.NET do developers gain the advantages of true code inheritance, which allows business logic to be more easily and reliably propagated through an organization. In this section, we briefly introduce some principles of object-oriented design and describe the benefits it can provide to VB developers.

Object-Oriented Concepts

One could write an entire book on Object-oriented design (and indeed, many people have) but we will provide an introduction here. The primary advantage of object-oriented (OO) languages compared to their procedural predecessors is that not only can you encapsulate data into structures; you can also encapsulate behavior as well. In other words, a car not only describes a collection of bolts, sheet metal, and tires (properties), but it also describes an object that can speed up and slow down (methods).

OO design frequently requires more up-front work than other environments, and usually the design process starts by enumerating a list of declarative sentences that describe what an object must do. For example, if you were building a car using object-oriented principles, you might describe the requirements as follows:

- The CAR must ACCELERATE.
- The CAR is a type of VEHICLE.
- The CAR has the color RED.

Now we know enough to begin defining the *objects* we need. In general, the nouns in these sentences describe the objects that are required (in this case, the car); the verbs describe the methods that the object must perform, and the adjectives describe the properties contained within the object. Then, after each of these are defined, the code can be developed to support these requirements. This breakdown is summarized in Table 1.4.

Table 1.4 Object-Oriented Terms

High-Level Concept	Part of Speech	Example
Objects	Nouns	Car
Methods	Verbs	Accelerate
Properties	Adjectives	Color=Red

Advantages of Object-Oriented Design

The true advantages to object-oriented design come when you can propagate behavior from one object to another. For example, if you were developing a sedan and a coupe, you might design few differences between the two cars other than the number of doors (four versus two).

This is where *inheritance* comes in. If you already had a sedan designed, you could build a coupe just by *inheriting* all of the behavior of the sedan, except for *overriding* the number of doors. Observe the following VB pseudocode:

```
Public Class Coupe
   Inherits Sedan
   Overrides Sub BuildDoors()
     Doors = Doors + 2
   End Sub
End Class
```

Now, if you add new features to the sedan (such as side air bags, for example), they are automatically propagated to the coupe without adding any additional code.

By contrast, overloading is when you want the methods of a single object to have different behaviors depending upon what parameters you pass to it. Then, VB is smart enough to determine which module to run depending upon the parameter list. The differences between overriding and overloading are summarized in Table 1.5.

Table 1.5 Overriding versus Overloading

Type	Overriding	Overloading
Method Name	Same	Same
Argument List	Same	Different
Behavior	Replaces existing method	Supplements existing method

By combining the new overloading and overriding capabilities of VB.NET, you can create applications that are much more stable and scalable.

Developing & Deploying…

Taking Care with Inheritance

There is a famous story about the Australian army that illustrates the risks involved with careless OO design. They were developing an object-oriented combat training simulation. First, they created a soldier object that could move and shoot. The programmers then wanted a kangaroo object. Because so much of the behavior was the same, they decided to save some time and inherit the soldier as the parent class and added the ability to hop. Unfortunately, because they didn't override the attack method, the next time the virtual soldiers encountered the virtual kangaroos, the kangaroos shot back at them!

History of Object Orientation and VB

Visual Basic has been best described as an *object-based* language, rather than an *object-oriented* one, because it did not support true inheritance from one object to another. Programmers have used different methods to simulate Inheritance since VB 5.0, specifically by using the *Implements* interface. Although this feature didn't actually bring functionality of a parent class, at least it defined a set of methods that would need to be coded. However, there was not an effective way to reuse business logic. This was a clumsy workaround, at best, and is far inferior to the overriding and overloading that are now available.

Namespaces

One final new topic that addresses OO design is that of *namespaces,* which are used in the .NET architecture to keep application resources separated to reduce global conflicts. One of the major design decisions of .NET was to try to reduce the risk of harmful program interaction, while still allowing applications that were intended to work together to share their resources effectively. To achieve this, Microsoft introduced namespaces. Now, when you declare a resource, you also must declare the namespace where that resource will reside. Although the resources will traditionally reside in a local namespace local to the user, it is possible to override that. Of course, you may occasionally need to expose code in common repositories. Although .NET supports this approach, you now need to digitally sign and authorize your code to achieve this. Because of the extra hassles involved, this approach will likely be less used in the future.

Web Applications

In general, a *Web application* is an application that uses resources that are distributed on the client's machine and on one or many Web servers, which may in turn require resources from other servers. This chapter first describes the different ways this has been done in the past and then focuses upon the new resources available to the VB.NET developer.

Web Applications Overview

In the past, four primary approaches were used to develop Microsoft Internet applications:

- **ActiveX documents** You could compile your applications to a VPD, which allowed a nonmodal VB application with an interface that resembled a traditional VB app to be displayed directly in the Internet Explorer interface. Unfortunately, this is not directly supported in VB.NET, so you will probably want to maintain legacy applications using this architecture in VB 6.0.
- **DHTML applications** You could create applications that deployed content to a browser using extensions to HTML that allowed significant data entry and validation to be performed on the client without requiring a round-trip to the server. This approach would require applications that were much smaller and easier to deploy than those created using ActiveX documents. Unfortunately, this approach is not directly supported in VB.NET, so you will probably want to maintain legacy applications using this architecture in VB 6.0.
- **ASP applications** You could create applications that executed primarily on the server, dynamically generating the HTML required to render the interface for the application. Although this approach has been very popular, it can lead to code that can be difficult to maintain.
- **WebClasses** Finally, you could create applications visually that Visual Basic would translate into Internet applications. Although the implementation of WebClasses in VB 6.0 was very limited, WebClasses have evolved into Web forms, which are the preferred approach for developing and deploying Internet applications in VB.NET.

Web Forms

The idea behind ASP applications is that each page is generated dynamically for the user. Because this work is performed on the server, this approach has the huge advantage of being relatively browser- and version-independent—all that the browser has to do is display a static page, and the server does the rest of the work. However, when used by inexperienced engineers, this approach can be difficult to maintain, debug, deploy, and update. Although Web forms may not seem impressive compared to normal VB forms, they compare very favorably to a traditional ASP application.

By contrast, VB.NET supports the use of Web forms, which look similar to ASP pages but have four primary advantages:

- Unlike ASP pages, which are interpreted when they are executed, Web forms are compiled when they are first used, so the performance can be much better.
- Unlike ASP pages, which didn't natively support VB, the full language is now available directly from this environment.
- Building and maintaining the layout of the Web forms is much easier using the built-in VB designers than it was to code them by hand in ASP. (Although ASP has had visual layout tools since InterDev 6.0, these were awkward and rarely used in professional environments.)
- Separating the presentation layer and business layer of the application is much easier, which makes it easier to leverage specialized development resources instead of requiring that all of your developers be skilled in page design.

Web Services

One of the greatest challenges in designing Web applications that communicate with each other is trying to define and determine the required application interfaces. Unless you had a pre-existing strategic relationship with the applications that you were leveraging, you might be unable to integrate your applications, or you might be forced to integrate them in a very inefficient way. For example, some applications can interact only by having one application *pretend* to be a user with a Web browser, navigating among the screens of the target application and screen-scraping the needed information off of the display. The disadvantages of this approach are numerous: You waste server resources by displaying more data than is needed to perform the transfer, and you run the risk of your application breaking whenever the screen layout would change.

This is where Web Services come in. Now, writing server applications that are capable of exposing functionality to non-Microsoft applications is much easier. Features include the following:

- Direct support of industry standard XML for passing information
- Greater platform independence than can be provided through MTS
- Use of HTML to get through firewalls (but note the following warning)

WARNING

Firewalls are explicitly created by network administrators to restrict access on certain ports. You can bypass this by routing your data through HTTP port 80. However, when using this approach, make sure that you consider the security priorities not only of your own organization, but also of the organization you are interfacing with.

Of course, to consume these Web services you need to use the new discovery capability of Web services. This allows an external application to know what methods are available, and what parameters are required to drive them. This is performed by using the protocols HTTP and SOAP. These protocols are described in the following sections.

HyperText Transport Protocol

The *HyperText Transfer Protocol* (HTTP) is the backbone of the Internet. It is most frequently used to transmit Web pages from one computer to another, but it also can be used to transmit other kinds of information.

When you type a URL into a browser, you specify the protocol you use to download the content to your local browser (for example, in http://www.microsoft.com, the protocol is http). This protocol is designed to emphasize reliability over speed, because for Web applications it is more important to wait a little longer to get everything right the first time.

A disadvantage of HTTP is that a separate connection must be created for every resource that is downloaded. It also is not as fast as other protocols (such as FTP) because of this increased overhead. However, more recently, newer versions of Internet servers have done a better job of caching and connection pooling to reduce these disadvantages.

In the .NET architecture, the HTTP protocol is used in conjunction with the SOAP protocol to transmit information and instructions from one Web server to another. The following section describes the SOAP protocol in more detail.

NOTE

Don't confuse HTTP with HTML, which stands for HyperText Markup Language. That is the protocol that defines how Web pages are laid out visually, not how they are transferred from one computer to another.

Simple Object Access Protocol

The *Simple Object Access Protocol* (SOAP) is not nearly as widely used as HTTP, but it is expected to have a large impact in the future. SOAP is a protocol that works on top of HTTP to communicate between servers. Although HTTP simply is used to pass strings of data, SOAP is a way of organizing those strings to represent messages that can be easily parsed and understood either by a computer or by a human analyst. Instead of passing messages in proprietary protocols, it simply sends strings in XML in human-readable form. For example, observe the following excerpt from a simple SOAP message:

```
<SOAP:Body>
  <MyValue>12345</MyValue>
<SOAP:Body>
```

Although HTTP is used to make sure that all the letters and numbers get from point A to point B, the SOAP protocol inserts the hierarchical tags that ascribe meaning to the content.

Other protocols allow servers to communicate with each other. For example, DCOM is used in the Microsoft world, and RMI provides roughly the equivalent functionality for Java applications. However, these protocols work poorly when they span different operating systems.

Of course, this approach has its downsides. SOAP messages will never be as small as those sent using proprietary technologies. For example, in the earlier message, the number 12345 would take either 5 or 10 bytes, (depending on whether or not you were using the international Unicode standard), plus the bytes required to send the XML tags themselves. By contrast, that information could be transmitted in 2 bytes if it was stored as an integer.

Also, the use of SOAP doesn't eliminate the need to have a clear understanding of the contents of the message received. It simply pushes the responsibility for interpretation from the operating system to the programmer.

Security

As applications are extended to the Internet, new risks are extended to the organizations that deploy these applications. The security models for existing client/server applications have been based upon several assumptions. Unfortunately, as the boundaries between client/server, Internet, intranet, and distributed applications have become blurred, some of these assumptions have been challenged. It is no longer safe to focus security efforts upon servers, because the lines between servers and clients have been blurred. It is no longer safe to assume that the effects of an application can be analyzed on a single computer, because more applications now run on and require the resources of multiple machines. And if you are deploying your application to the general public, it is no longer safe to assume you can identify all of the users of your application. Because of this, Microsoft has now introduced a new security tool in .NET to support the developer: **SECUTIL**. This tool makes it easier to extract information about the user identity, after the user has been validated using the Public Key Value (internal users) or the X.509 certificate (external users).

Because of this, users are accountable for their code. In the past, a developer could write their own version of an OCX or DLL, copy it into a Windows system directory and register it, and this would have an impact upon every other application that was dependent upon that resource.

Although this was a handy way to quickly deploy patches, it also infuriated developers whose code failed when used with the new DLL due to dependence upon behavior that was altered in the new versions of the code. By contrast, by using **SECUTIL**, it is possible to identify what code was developed by what developer, which increases accountability.

Type Safety

Although much of VB.NET allows you to eliminate development steps, a few cases exist where you need to take extra precautions in this new environment, and type safety is one of those factors. *Type safety* is the enforcement of variable compatibility when moving data from one variable to another. In this section, we examine the new requirements in VB.NET and the approaches to address this requirement.

Casting

If you have experience with languages such as C++ or Java, then you are probably experienced with casting. If you are an experienced VB developer, then you probably have used casting, but the term may be new to you.

Casting is the process of explicitly converting a variable of one type to a variable of another type, and it is used to reduce bugs caused by moving information into variables using inappropriate data types. For example, observe the following code:

```
Dim A as integer
Dim B as long
A = 20000
B = CLng(A)
```

The variable *B* has been explicitly cast to a Long type, using the **CLng** function. A cast function exists for each type of variable. Some examples of this are provided in Table 1.6. Casting is not new in VB.NET, but it is more important, for reasons discussed in the next section.

Table 1.6 Cast Functions for Variable Types

Cast Function	Action
CLng	Convert to a "Long"
CStr	Convert to a "String"
CInt	Convert to a "Integer"
CDbl	Convert to a "Double"

Data Conversion

When you convert from one variable type to another, it is called *narrowing* if there is a risk of loss of precision, and *widening* if there is no risk of this loss.

In other languages like C++, the developer explicitly tells the compiler what to do when you *pour* data from one variable into another with a risk of data loss. The reason is to provide informed consent—to make sure that you are aware of the risk and accept responsibility for the consequences if the data is too large for the defined container.

Now, in the current version of VB.NET, Microsoft has introduced Option Strict. If you use this option, you must perform an explicit cast for every narrowing assignment. For example, with Option Strict off, the following line would successfully compile:

```
Dim a as integer
Dim b as long
A = 20000
B = a ' Cint excluded
```

But with Option Strict on, this code would generate a compilation error.

Bitwise Operations

VB.NET enforces more precise type usage in other ways as well, and some shortcuts that were used by previous generations of VB programmers are no longer permitted.

In VB.NET, when writing conditional code, the parameter used for the **IF** statement must be of the type Boolean. In previous versions of VB, programmers could take a shortcut, and implicitly cast the integer 0 to the Boolean False. For example, the following line of code would work in VB 6.0:

```
Dim a as integer
A = 0
If (a) then MsgBox "Hello world"
```

This code would, however, fail in VB.NET. To correct the code, you have to make the following change in the third line:

```
Dim a as integer
A = 0
If (CBool(a)) then MsgBox "Hello world"
```

Note that this situation is similar to the relationship between C++ and Java. Java supports only Booleans with **IF**, whereas C++ allowed implicit casting of other variable types.

New Compiler

Although you will normally use the compiler from within the IDE, you also have new flexibility in compiling from the command line with VB.NET. In this section, we take a look at how you can use the compiler, and then we take a look at some of the advantages to the executables created by the new compiler.

Compiling an Executable

You can initiate compilation from the command line, invoking the executable wsc.exe, with the parameters shown in Table 1.7.

Table 1.7 Parameters for WSC Compiler

Tag	Meaning
/t	The type of output code. For example an EXE means a console application, while WINEXE means that it is a Windows application.
/r	References to include (all DLLs that are referenced in the app).
/version	The version number visible when the properties of the executable are viewed (major version, minor version, revision, and build).
The last parameter	The VB file to compile.

Architecture

To understand the operation of the new VB compiler, you need to understand the architecture for the applications that the VB compiler creates.

Previously, the executable created by a language such as C++ would make direct references to registers, interrupts, and memory locations. Although working inside the Microsoft foundation classes could reduce the risk of error, eliminating risk due to inexperience (or malice) was not possible.

That has changed with VB.NET. Now, instead of compiling directly to hardware-specific machine code, the compilation is performed to MSIL (Microsoft Intermediate Language). The syntax of MSIL is similar to machine code, but any EXE or DLL containing MSIL will need to be reinterpreted after it is deployed to the destination machine.

File Management in Previous Versions of VB

In previous versions of VB, each resource that you included in your project would have its own extension and reside in its own file, with an extension that identified the type of resource, as shown in Table 1.8.

Table 1.8 Sample File Extensions in VB 6.0

Resource Type	Extension
Form	.frm
Class Module	.cls
Module	.bas

Although this made it easy to interpret the resource type immediately, it also made it very difficult to manage projects with large numbers of small classes. Another challenge was trying to keep filenames reconciled with class names. This became especially difficult as projects grew and changed in focus.

File Management

In VB.NET, the filename extension restriction has been removed. Now, regardless of which type of resource you create, it will have the same extension (see Table 1.9).

Table 1.9 Some of the File Extensions in VB.NET

Resource Type	Extension
Form	.vb
Class Module	.vb
Module	.vb

You can also concatenate as many resources as you want into a single file, regardless of type. The default behavior (when using the Project | Add Class menu option) is still to create new files, but you can copy this content into a single source file. For example, two distinct classes could be represented in the file MyClasses.vb with the following code:

```
Public Class Beeper
 Public Sub Beep()
  MsgBox("Beep")
 End Sub
End Class

Public Class Booper
 Public Sub Boop()
  MsgBox("Boop")
 End Sub
End Class
```

Changes from Visual Basic 6.0

The following sections detail some options you have to prepare for VB.NET. First, we look at features in VB 6.0 that are gone in VB.NET. Then we look at new features in VB.NET. Finally, we observe features that are present in both versions but with some significant changes. This section doesn't cover every change, but it will provide enough context to illustrate the challenges and opportunities involved with this transition.

Variants

The Variant data type is no longer supported in VB.NET, and it has been merged into the Object type. More specifically, because all variables are now objects, a variant is simply defined as an object.

Variable Lower Bounds

To make the language compatible with the other .NET languages, you no longer can start an array at 1 using the **Option Base** command. All arrays are now forced to begin with array element zero.

Fixed Length Strings

You now cannot create strings of fixed length. In previous versions of VB, you could write the following code to define the string to be exactly 12 characters long:

```
Dim sLastName as String * 12
```

This is no longer supported in VB.NET to ensure compatibility with the other .NET languages.

NULL Propagation

In previous versions of Visual Basic, any expression that had a NULL in it would yield a null. For example, 1 + NULL would yield a NULL in VB 6.0. However, VB.NET does not support NULL propagation. If you are using it to do error handling, you should rewrite your code and use the **IsNull** function.

NOTE

Interestingly, this approach to null propagation is not standard for all Microsoft applications. One major difference between SQL Server 6.5 and 7.0 is that null propagation has been introduced into 7.0 unless you explicitly disable it. In other words, A + NULL + B would equal AB in SQL Server 6.5, but NULL in version 7.0. This was done to comply with the ANSI SQL standard.

Other Items Removed

In addition to those already mentioned, the following features shown in Table 1.10 are no longer supported in VB.NET.

Table 1.10 Language Substitution Strategies

Statement	Old Operation	Approach to Replace
GoSub	Allowed execution of a section of code without leaving the existing function or procedure.	Replace with new modules.
Computed GoTo / **GoSub**	Acted like the **Switch** statement, but selecting one of many sections of code to execute.	Use **Select Case** or **Switch** with custom functions.
DefInt, DefLong, DefStr, and so on	Defined a range of scalar variables of the type specified with a certain range.	Define each of the variables explicitly or rewrite code to support an array.
Lset	Reassign variables of user-defined types.	Copy over components of new types individually.

Function Values

You now can return a value from a function using the command **Return,** instead of needing to assign the value to the name of the function. Not only does this make it easier to terminate the function (instead of having to use two lines to set the value and then **Exit Function**, these two statements can be rolled up into a single command), it also means that you can rename the function without having to change all the references to the function name.

Short Circuits

In many other languages, as soon as an **IF** statement resolves to False, the other parts of code do not execute. For example, observe the following piece of code:

```
If DebitsCorrect("Chase") and CreditsCorrect("Citibank") then
     MsgBox "Transaction processed"
End if
```

In VB6, both the function **DebitsCorrect** and the function **CreditsCorrect** would always execute. However, in one of the new features proposed for VB.NET, if **DebitsCorrect** resolved to False, then **CreditsCorrect** would never execute. This behavior is called *short circuiting* because the code knows that the expression can never resolve to True if the first half resolves to False; it doesn't have to bother to execute the second half of the expression. Unfortunately, this causes greater incompatibility with legacy code, which is why Microsoft has not confirmed whether or not they will include this change in the final release of VB.NET.

Properties and Variables

Of course, many of the day-to-day changes you will notice are evolutionary, not revolutionary. In this section, we look at the impact of changes in how properties and variables are stored and manipulated.

Variable Lengths

Unfortunately, in the history of computer science there has been disagreement over the definition of a byte, which has led to significant confusion for the modern developer. Many early computers used eight bits (binary digits) to describe the smallest unit of storage, but when computers became more powerful and stored data internally in larger structures, some developers still thought that a byte was eight bits, whereas other developers thought that a byte should still represent how the processor stored data, even if it used 16 bits, or more. Because of this situation, the size of the variables in C could change when code was recompiled on other hardware platforms, and other languages that came in the future reflected these incompatibilities. Now, in .NET, the definitions of the variable types have been standardized, as shown in Table 1.11.

Table 1.11 Variable Lengths in Bits

Bit Length	VB 6.0	VB.NET
8 bits	Short	Byte
16 bits	Integer	Short
32 bits	Long	Integer
64 bits	N/A	Long

These definitions bring the standards in line with those used in the rest of the .NET suite of application development tools. Although some of these variable names will be automatically substituted when a VB 6.0 application is imported into VB.NET, you should still examine the finished code to make sure that the new code reflects your application needs. (Also be aware that this will also affect the changes made in the API calls—if it used to be a *Long*, it should now be an *Integer,* and so on.)

Get and Set

Previously, your **Get** and **Let**/**Set** statements had to be coded separately, as two separate blocks of code residing in a class. Of course, it was possible to have a **Get** without a **Let**/**Set** for read-only properties (or vice versa, for write-only properties), but for most properties, this added unnecessarily clumsiness to the organization of the class modules. Now, in VB.NET, these are now grouped together in a single module that is broken down into two sections that support both assigning and retrieving these values.

Date Type

In earlier versions of Visual Basic, variables of the Date type were stored internally as Doubles (with the number of days to the left of the decimal point and the fraction of a day stored to the right). Therefore, many developers chose to store their dates as Doubles instead of as Dates, even after VB introduced the Date type.

This approach had many advantages. (For example, when using heterogeneous databases, it was often more reliable to store data as numbers, and the math was often much easier as well if you reserved the use of Dates for presentation only.) However, in VB.NET, Double and Date are no longer equivalent, so you should use the Date type for date use in VB.NET, or you may get compilation errors. Although Dates are now represented internally using the .NET DateTime format, which supports a greater precision and range of dates, you can still use the **ToOADate** VB.NET function to convert this type back into a Double-compatible format.

Default Properties

Some Visual Basic developers use a shortcut to omit the reference to the default property of an object. For example, if you wanted to assign a value to a text box, instead of writing this:

```
tbFirstName.text = "John"
```

You could instead write this:

```
tbFirstName = "John"
```

Each control had a default property that would be referenced if you omitted the name of the property, and when you created your own objects you could define the default property you wanted to use for it.

However, in VB.NET, because all data types are now represented as objects, a reference to an object that omits any property can be interpreted as the object itself instead of a default property of an object. Therefore, when developing applications in VB.NET, remember to explicitly declare the default properties.

Summary

VB.NET introduces many exciting new features to the VB developer, though these enhancements do cause some minor compatibility issues with legacy code. The new Integrated Development Environment (IDE) incorporates some of the best ideas of VB 6.0 and InterDev to make it easier and more intuitive to quickly create applications using a wider variety of development resources. The code developed in the IDE can then be compiled to work with the new .NET Framework, which is Microsoft's new technology designed to better leverage internal and external Internet resources. The compiler writes the code to Common Language Runtime (CLR), making it easier to interact with other applications not written in VB.NET. It is now possible to use true inheritance with VB, which means that a developer can more efficiently leverage code and reduce application maintenance. Not only is the CLR used for stand-alone VB applications, it is also used for Web Applications, which makes it easier to exploit the full feature set of VB from a scripted Web application. Another way in which security is enhanced is through enforcement of data type compatibility, which reduces the number of crashes due to poorly designed code. Exploiting the new features of VB.NET is not a trivial task, and many syntax changes were introduced that will cause incompatibilities with legacy code. But, many of these are identified, emphasized, and in some cases automatically updated by the IDE when a VB 6.0 project is imported into VB.NET.

Solutions Fast Track

Examining the New IDE

☑ The improvements in the new IDE can be broken down into two categories: those that conserve development time and those that conserve screen real estate.

☑ Among the cosmetic improvements in the new IDE are multimonitor support, tabbed forms, a better layout for the toolbox, expandable code, and live interactive help.

☑ Among the development improvements in the new IDE are an integrated menu editor, an enhanced solution explorer, a server explorer that permits the developer to directly access resources on remote hosts, and a dynamically configurable IDE home page.

.NET Framework

- ☑ The .NET Framework is made up of three parts: the Common Language Runtime, the Framework classes, and ASP.NET.
- ☑ The Common Language Runtime provides the interface between your code and the operating system.
- ☑ The Framework classes offload much of the work done by the VB into language-independent development libraries.
- ☑ ASP.NET provides direct access to the full VB language from a scripting platform.

Common Language Runtime

- ☑ In .NET, the compiler no longer reduces the source code into a file that can be directly executed.
- ☑ Instead, the code is compiled into CLR, a Common Language Runtime that has an identical syntax regardless of the .NET compiler used to generate it.
- ☑ By executing CLR instead of compiled code, the operating system can reduce the number of system crashes caused by the execution of erroneous or malicious code, while also increasing opportunities for cross-platform compatibility.

Object-Oriented Language

- ☑ Previous versions of Visual Basic did not offer true object-oriented inheritance of code from a parent class to a child class.
- ☑ In VB.NET, propagating code from one module to another is now possible, while only overriding the behavior that needs changed in the child class, thus improving maintainability.
- ☑ Because of the CLR, not only can a VB developer inherit a class from another VB module, he can also inherit from a module developed in another language, such as C#.

Web Applications

- ☑ Web applications are the successor of Web forms in VB 6.0.
- ☑ Using Web applications allows a developer to separate the presentation layer of an application from the business layer and data layer.
- ☑ Web applications can be more efficient than traditional ASP applications, because the ASPX pages are compiled when they are first run.

Security

- ☑ Microsoft has now introduced a new security tool in .NET to support the developer: **SECUTIL**. This tool makes it easier to extract information about the user identity, after the user has been validated using the Public Key Value (internal users) or the X.509 certificate (external users).
- ☑ By using **SECUTIL**, it is possible to identify what code was developed by what developer, which increases accountability.

Type Safety

- ☑ To reduce the security and application risks associated with careless variable assignment, VB.NET is more restrictive than VB 6.0 when copying data from one variable to another.
- ☑ If you assign a variable residing in one variable to another, and the second variable cannot store numbers as large as the first variable, it is now necessary to explicitly cast the variable to the new type. (In VB 6.0, in most cases the conversion happens automatically.)

New Compiler

- ☑ The Compiler in VB.NET compiles the code not into code that can be directly executed by the OS or by a runtime module, but rather to CLR syntax.
- ☑ The code generated by the new compiler is more reliable (many errors are screened out at runtime), more secure (security holes have been closed), and more interoperable (new CLRs could potentially be generated for other platforms in the future).

Changes from Visual Basic 6.0

- ☑ Although there are substantial syntax changes between VB 6.0 and VB.NET, in most cases clear substitutes are available to the developer to accomplish the same thing.
- ☑ Many of these substitutes are automatically substituted when a VB 6.0 project is imported into VB.NET. However, you should still inspect and heavily test your application after any such conversion.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: If we're deploying other .NET applications across our organization, do I need to update my applications builtin VB 6.0?

A: No. Traditional COM based applications will continue to be supported for at least the next several years, and CLR applications will interface cleanly with legacy code. However, if you want to start gaining some of the advantages of CLR, you may consider writing a wrapper application in VB.NET that is used as the new interface to your application. Remember that (unlike some other Microsoft products) you can have VB 6.0 and VB.NET on your computer at the same time and use them to support different families of products.

Q: When should I use ASP.NET, and when should I use MTS?

A: If you need to support distributed transactions, you may want to stick with MTS because ASP.NET will not support that feature in its initial release. Conversely, if you need to use XML to pass data, it may be easier to do this with ASP.NET than with MTS (though, of course, you could write your own tools in MTS to accomplish the same thing). Over time, ASP.NET will probably replace most of the need for MTS.

Q: I am creating a project that will have 40 classes. Should I put all of these classes into their own file or into one big file?

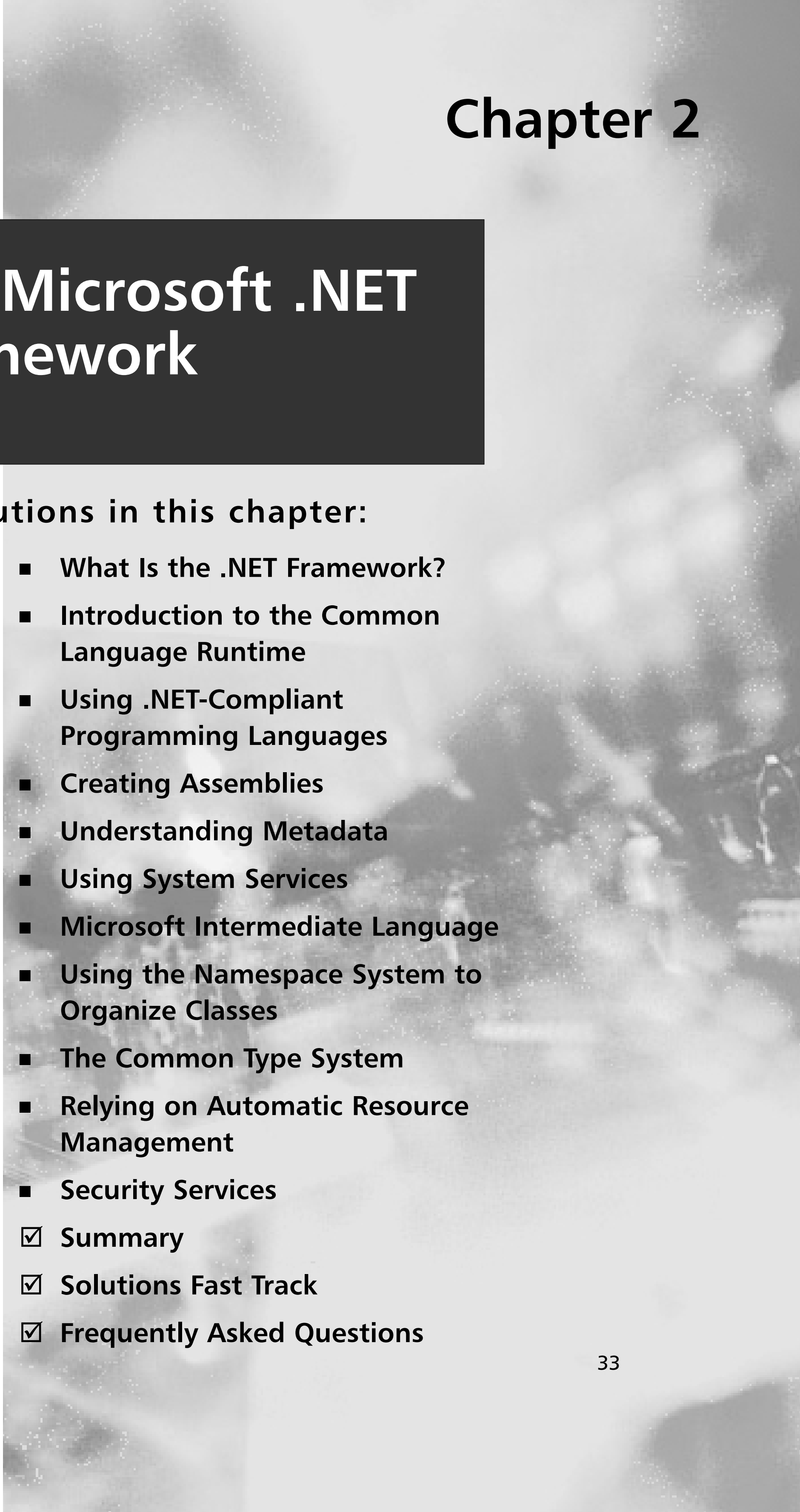
A: This may depend upon how your work team is organized. If you have a small project team, you may want to aggregate many of your resources together into a single file. However, if you have a large, decentralized team, you may want to keep the old style of separating the classes into many different files, because this would work better with traditional version control software. Just because Microsoft has added a feature doesn't mean you have to use it.

Q: On my project team, we've set up inheritance, but we're having some problems. My team is inheriting objects created by another team, but whenever the other team changes the behavior of their objects, our code breaks. What can we do?

A: You can organize your object model in many different ways, but one popular approach is to use abstract classes. These classes are not directly instantiated, but they are intended to serve only parents of other classes. Instead of inheriting directly from the objects created by the other team, you may want to work with them to define a subset of functionality that won't change, put that in an abstract class, and then both inherit instead from that shared object.

Chapter 2

The Microsoft .NET Framework



Solutions in this chapter:

Introduction

Chapter 1 provided an overview of Visual Basic .NET applications; let's now look more closely at the various components of the .NET Framework. The .NET Framework includes a number of *base classes*, which you will use to begin. The Framework includes abstract base classes to inherit from as well as implementations of these classes to use. You can even derive your own classes for custom modifications. All the classes are derived from the *system object*. As you can imagine, this gives you great power and flexibility. Some of this power was previously available in Visual C++, but now you can have this same power within Visual Basic. All applications will share a common runtime environment called the Common Language Runtime (CLR). The .NET Framework now includes a common type system. This system allows all the languages to share data using the same types. These features facilitate cross-language interoperability.

To use .NET, you are required to learn some new concepts, which we discuss throughout this chapter. A Visual Basic .NET application is wrapped up in an *assembly*. An assembly includes all the information you need about your application. It includes information that you would find currently in a type library as well as information you need to use the application or component. This makes your application or component completely self-describing. When you compile your application, it is compiled to an intermediate language called the Microsoft Intermediate Language (MSIL). When a program is executed, it is then converted to machine code by CLR's just-in-time (JIT) compiler. The MSIL allows an application to run on any platform that supports the Common Language Runtime without changing your development code.

Once the code has been prepared, .NET's work is still not done. .NET continues to monitor the application and performs *automatic resource management* on the application to clear up any unused memory resources and provide security measures to prevent anyone from accessing your assembly.

In these few paragraphs, we've introduced the major new concepts found within .NET: the CLR, the assembly unit (and its contents), what makes .NET interoperable, and how .NET is "smart" in terms of automatic memory management and security.

What Is the .NET Framework?

The *.NET Framework* is Microsoft's latest offering in the world of cross-development (developing both desktop and Web-usable applications), interoperability, and, soon, cross-platform development. As you go through this chapter, you'll see just how .NET meets these developmental requirements. However, Microsoft's developers did not stop there; they wanted to completely revamp the way we program.

In addition to the more technical changes, .NET strives to be as simple as possible. .NET contains functionality that a developer can easily access. This same functionality operates within the confines of standardized data types and naming conventions. This internal functionality also encompasses the creation of special data within an assembly file that is vital for interoperability, .NET's built-in security, and .NET's automatic resource management.

Another part of the "keep it simple" philosophy is that .NET applications are geared to be copy-only installations; in other words, the need for a special installation package for your application is no longer a requirement. The majority of .NET applications work if you simply copy them into a directory. This feature substantially eases the burden on the programmer.

The CLR changes the way that programs are written, because VB developers won't be limited to the Windows platform. Just as with ISO C/C++, VB programmers are now capable of seeing their programs run on any platform with the .NET runtime installed. Furthermore, if you delegate a C programmer to oversee future developments on your VB.NET program, the normal learning curve for this programmer will be dramatically reduced by .NET's multilanguage capabilities.

NOTE

Visualization is still key! Die-hard VB programmers may find themselves having a hard time visualizing all the new concepts in VB.NET (and we all know that proper logic visualization plays a big role in what we do). Something that may help is to think about VB.NET as a completely flexible language that can accommodate Web, console, and desktop use.

Introduction to the Common Language Runtime

CLR controls the .NET code execution. CLR is the step above COM, MTS, and COM+ and will, in due time, replace them as the Visual Basic runtime layer.

To developers, this means that our VB.NET code will execute on par with other languages, while maintaining the same, small file size.

The CLR is the runtime environment for .NET. It manages code execution as well as the services that .NET provides. The CLR "knows" what to do through special data (referred to as *metadata*) that is contained within the applications. The special data within the applications store a map of where to find classes, when to load classes, and when to set up runtime context boundaries, generate native code, enforce security, determine which classes use which methods, and load classes when needed. Since the CLR is privy to this information, it can also determine when an object is used and when it is released. This is known as *managed code*.

Managed code allows us to create fully CLR-compliant code. Code that's compiled with COM and Win32API declarations is called *unmanaged code*, which is what you got with previous versions of Visual Basic. Managed code keeps us from depending on obstinate dynamic link library (DLL) files (discussed in the Ending DLL Hell section later in this chapter). In fact, thanks to the CLR, we don't have to deal with the registry, graphical user identifications (GUIDs), AddRef, HRESULTS, and all the macros and application programming interfaces (APIs) we depended on in the past. They aren't even available options in .NET.

Removing all the excess also provides a more consistent programming model. Since the CLR encapsulates all the functions that we had with unmanaged code, we won't have to depend on any pre-existing DLL files residing on the hard drive. This does not mean that we have seen the last of DLLs; it simply means that the .NET Framework contains a system within it that can map out the location of all the resources we are using. We are no longer dependent upon VB runtime files being installed, or certain pre-existing components.

Because CLR-compliant code is also *Common Language Specification* (CLS)-compliant code, it allows CLR-based code to execute properly. CLS is a subset of the CLR types defined in the Common Type System (CTS), which is also discussed later in the chapter. CLS features are instrumental in the interoperability process, because they contain the basic types required for CLR operability. These combined features allow .NET to handle multiple programming languages. The CLR manages the mapping; all that you need is a compiler that can generate the code and the special data needed within the application for the CLR to operate. This ensures that any dependencies your application might have are always met and never broken.

When you set your compiler to generate the .NET code, it runs through the CTS and inserts the appropriate data within the application for the CLR to read. Once the CLR finds the data, it proceeds to run through it and lay out everything it needs within memory, declaring any objects when they are called (but not before). Any application interaction, such as passing values from classes, is also mapped within the special data and handled by the CLR.

Using .NET-Compliant Programming Languages

.NET isn't just a single, solitary programming language taking advantage of a multiplatform system. A runtime that allows portability, but requires you to use a single programming model would not truly be delivering on its perceived value. If this were the case, your reliance on that language would become a liability when the language does not meet the requirements for a particular task. All of a sudden, portability takes a back seat to necessity—for something to be truly "portable," you require not only a portable runtime but also the ability to code in *what* you need, *when* you need it. .NET solves that problem by allowing any .NET compliant programming language to run. Can't get that bug in your class worked out in VB, but you know that you can work around it in C? Use C# to create a class that can be easily used with your VB application. Third-party programming language users don't need to fret for long, either; several companies plan to create .NET-compliant versions of their languages.

Currently, the only .NET-compliant languages are all of the Microsoft flavor; for more information, check these out at http://msdn.microsoft.com/net:

- C#
- C++ with Managed Extensions
- VB.NET
- ASP.NET (although this one is more a subset of VB.NET)
- Jscript.NET

In addition, the following are being planned for .NET compliance. To obtain more information on these languages, visit the following URLs:

- Dyalog APL (www.dyadic.com, or directly at www.dyadic.com/msnet.htm)
- CAML (http://research.microsoft.com/Projects/SML.NET)
- Cobol (www.adtools.com/info/whitepaper/net.html)
- Eiffel (www.eiffel.com/announcements/2000/pdc)
- Haskell (www.haskell.org/pipermail/haskell/2000-November/000133.html)
- Mercury (www.cs.mu.oz.au/research/mercury/information/dotnet/mercury_and_dotnet.html)
- ML (http://research.microsoft.com/Projects/SML.NET)
- Mondrian (www.haskell.org/pipermail/haskell/2000-November/000133.html)
- Oberon (www.oberon.ethz.ch/lightning)
- Oz (reported by Microsoft as under development)
- Pascal (www2.fit.qut.edu.au/CompSci/PLAS//ComponentPascal)
- Perl (http://aspn.activestate.com/ASPN/NET/index)
- Python (http://users.bigpond.net.au/mhammond/managed_python/ManagedPython.html)
- Scheme (http://rover.cs.nwu.edu/~scheme)
- SmallTalk (reported by Microsoft as under development)

NOTE

Don't see your language on the lists in this section? Don't worry; it doesn't mean it's not going to happen! Several developers have mentioned waiting until .NET enters Beta 3 phase before writing a CLR compiler for their languages. If you don't think the particular programming language you're interested in will do it, write to the developers and let them know you want your language in .NET.

These developments will enhance your ability to work with multiple languages. For example, a COBOL advancement that might not exist in VB.NET doesn't mean that a VB.NET programmer can't take advantage of it. You can easily find a workaround for the issue using the COBOL solution as an example or simply convert the code to VB.NET.

Creating Assemblies

When you have multiple languages, how do they all work together to execute? Most other programming languages do not use Portable Executable (PE) format for their executables. With the .NET environment comes something new: a logical approach to executables named *assemblies*. The CLR handles the entire executing of an assembly. The assembly owns a collection of files that are referred to as *static assemblies*, which the CLR uses. Static assemblies can be resources used by the assembly, such as image files or text files that the application will use. The actual code that executes is found within the assembly in Microsoft Intermediate Language (MSIL) format. In other words, an assembly is roughly the equivalent of a VB 6.0 COM component. An assembly has three options that need to be set when you create it:

- Loader optimization
- Naming
- Location

The *loader optimization* option has three settings; *single domain*, *multidomain*, and *multidomain host*. The single-domain setting is the default and is used most in client-side situations. The JIT code is generally smaller when the single-domain setting is used, compared with the other two settings, and there is no noticeable difference between memory resources. The exception is if the application winds up being used as part of a multidomain or multidomain host setup, where it will actually hurt more than it'll help—such as within a client/server solution.

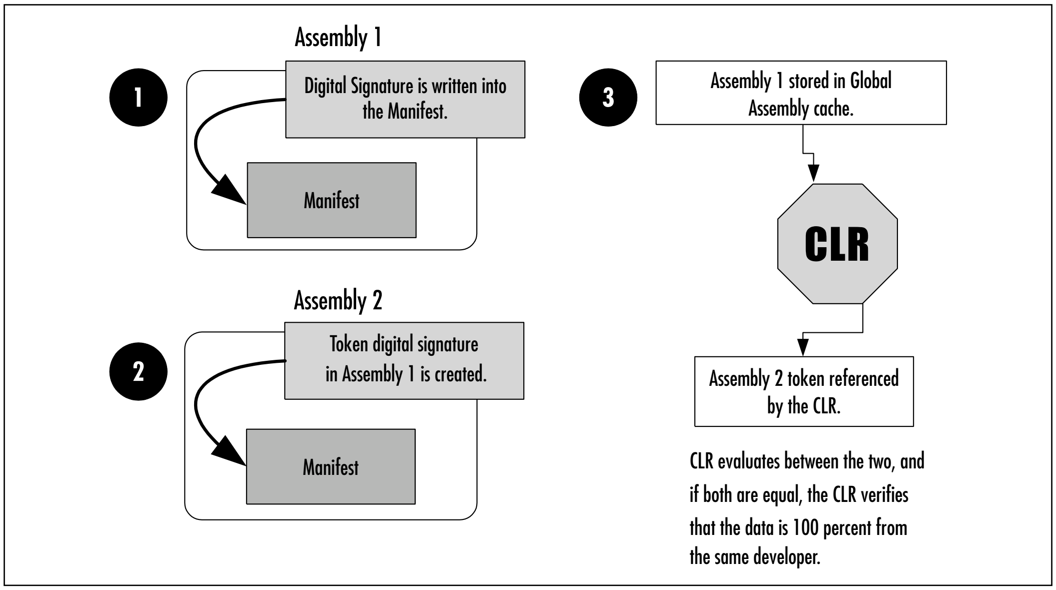
The multidomain and multidomain host settings apply to the same concept of multidomain usage. The only difference between the two is how the CLR will react with the code; in multidomain, the code is assumed to be the same across the domain. In multidomain host, however, each domain hosts different code. Let's say that you have an application development in which all the domains have the assembly filename, but each one has different code hosted to see how they can still interact. You would get the best performance using the multidomain host optimization routine.

You will receive many benefits by setting the assembly to be useable by multiple applications. Fewer resources will be consumed, since the type (object) will be loaded and mapped already, therefore the type won't need to be recreated each time it's needed. However, the end result of the JIT code is increased some, and access to static items are slower, since the static references are referenced indirectly.

The *name* of the assembly can impact the scope and usage by multiple applications. A single-client use application uses the name given to it when created, but there is no prevention for name collision. So, in order to help prevent name collisions in an assembly in a multiassembly scenario, you can also give the assembly a *shared name*. Having a shared name means that the assembly can be deployed in the *global assembly cache*, which you can think of as a global repository of assemblies.

A shared name is made up of the textual name of the assembly (the name you created for it) and a digital signature. Shared names are unique names due to the pairing of the text name and digital signature. This system, in turn, helps prevent name collision and keeps anyone using the same textual name from writing over your file, since the shared name is different. A shared name also provides the required information that's needed for versioning support by the CLR. This same information is used to provide integrity checks to give a decent level of trust. (For full trust, you should include a full digital signature with certificates.) Figure 2.1 illustrates how the shared-name process works.

Figure 2.1 The Shared-Name Process


Assembly 1
1
Digital Signature is written into the Manifest.
Manifest
Assembly 2
2
Token digital signature in Assembly 1 is created.
Manifest
3
Assembly 1 stored in Global Assembly cache.
CLR
Assembly 2 token referenced by the CLR.
CLR evaluates between the two, and if both are equal, the CLR verifies that the data is 100 percent from the same developer.

From the shared-name diagram, you can see that the shared name is first created into the primary assembly (Assembly 1), then the reference of the primary assembly is stored as a token of the version within the referencing assembly's (Assembly 2's) metadata, and it is finally verified through the CLR.

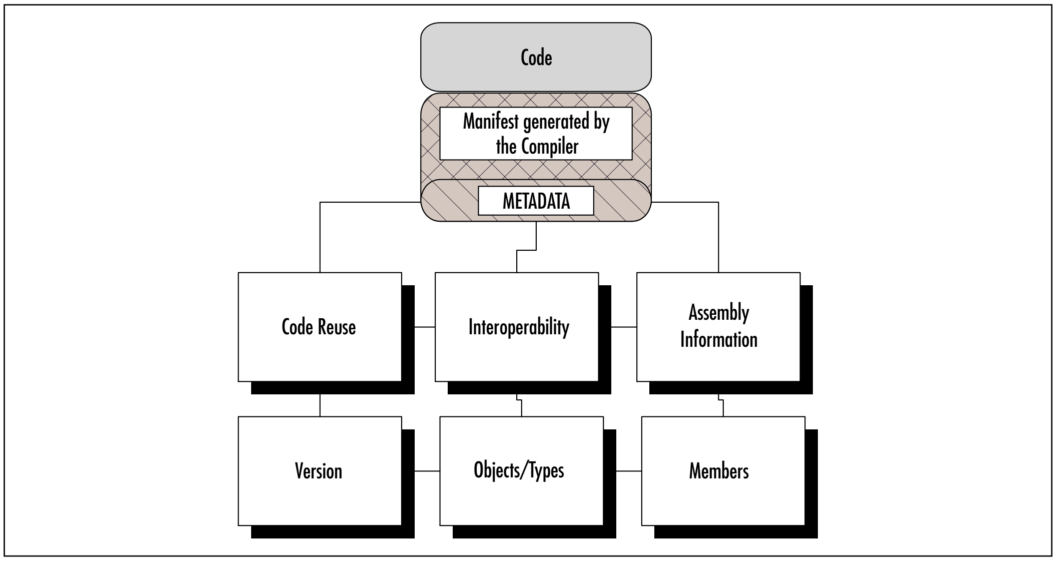
Once created, an assembly has the following characteristics:

- **Contains code that the runtime executes** PE MSIL code is not executed without the manifest present. In other words, if the file is not formatted correctly, it will not run.
- **Only one entry point** An assembly cannot have more than one starting point for execution by the runtime. For example, you cannot use both WinMain and Main.
- **Unit of side-by-side execution** An assembly provides the basic unit needed for side-by-side execution.
- **Type boundary** Each type declared within an assembly is recognized as a type of the assembly, not as a solitary type initiated into memory.
- **Security boundary** The assembly evaluates permission requests.
- **Basic deployment unit** An application made up of assemblies requires only the assemblies that make up its core functions. Any other assemblies that are needed can be provided on demand, which keeps applications from having the bloated setup files commonly associated with VB 6.0 runtime files.
- **Reference scope boundary** The manifest within the assembly dictates what can and can't occur in order to resolve types and resources. It also enumerates assembly dependency.
- **Version boundary** Being the smallest versionable unit in the CLR, all the types and resources that it has are also versioned as a unit. The manifest describes any version dependencies.

Figure 2.2 displays a typical assembly. The assembly has been dissected to display the code, the manifest area, the metadata within the manifest, and the information stored within the metadata.

As you can see, all the benefits that CLR gives us are located within the assembly but reside within the manifest.

Figure 2.2 A Typical Assembly



Using the Manifest

Apart from the MSIL, an assembly contains metadata within its manifest. We will go into detail about metadata and its uses in upcoming sections, but for now just remember that the metadata is all the relevant information that the CLR needs to properly run the file, and the manifest stores the metadata. Thanks to the manifest, assemblies are freed from depending on the registry and breaking DLLs (the cause of DLL Hell). Basic metadata includes the items listed in Table 2.1.

Table 2.1 Basic Attribute Classes

Basic Attribute Class	Description
AssemblyCompanyAttribute	Contains a string with the company name and product information.
AssemblyConfigurationAttribute	Contains current build information, as in "Alpha" stage.
AssemblyCopyrightAttribute	Copyright information that is stored as a string.
AssemblyDefaultAliasAttribute	Name information and alias information.

Continued

Table 2.1 Basic Attribute Classes

Basic Attribute Class	Description
AssemblyDescriptionAttribute	Provides a description of the modules included within the assembly.
AssemblyInformationalVersionAttribute	Any extra version information; this is not used by the CLR for versioning purposes.
AssemblyProductAttribute	Product information.
AssemblyTitleAttribute	Title of the assembly.
AssemblyTrademarkAttribute	Any trademarks of the assembly.

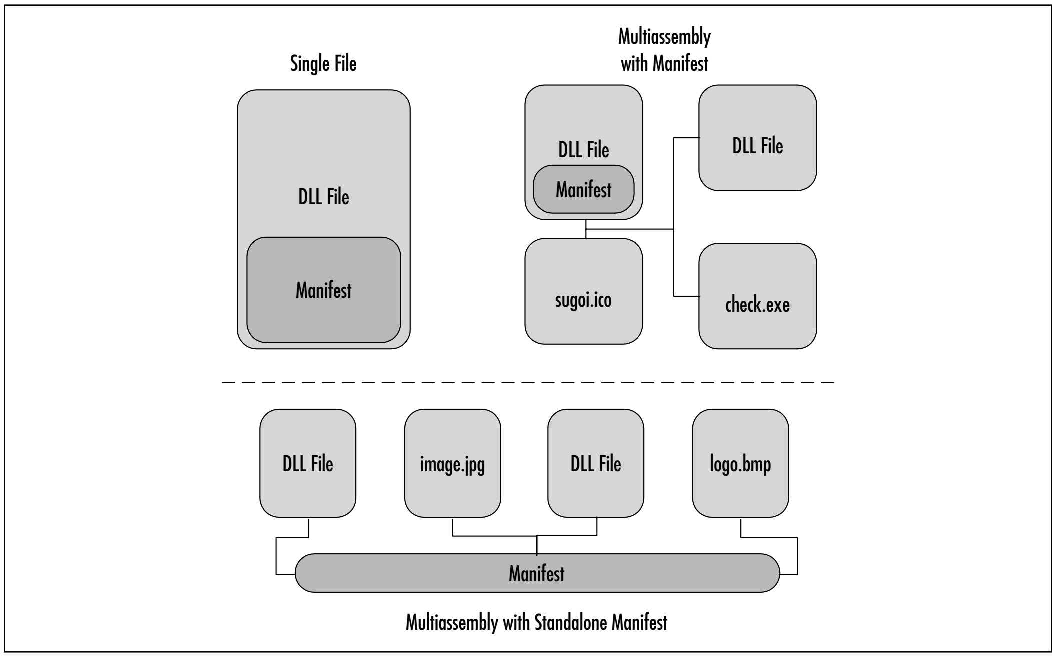
There are also custom attributes that you can set into the Manifest (see Table 2.2).

Table 2.2 Custom Attributes

Custom Attributes	Description
AssemblyCultureAttribute	Contains information on the "cultural" settings, such as base language or time zone.
AssemblyDelaySignAttribute	Tells the CLR that there is some extra space that might be empty to reserve space for a future digital signature.
AssemblyKeyFileAttribute	Contains the name of the file that contains the key pair for a shared name.
AssemblyKeyNameAttribute	If you use the CSP option, the key will be stored within a key container. This attribute returns the name of the key container.
AssemblyOperatingSystemAttribute	Information on the operating system(s) supported by the assembly.
AssemblyProcessAttribute	Information on the CPU(s) supported by the assembly.
AssemblyVersionAttribute	Returns the version of the assembly in the standard major.minor.build.revision form.

In regard to the third assembly option, *location*, a manifest's location on the assembly can also be altered, based on the type of assembly deployment. An assembly can be deployed as either a single file or multiple files. A single file-assembly is pretty much like a standard DLL file, because its manifest is placed directly within the application. Once again, the assembly is not that different from the standard executable or DLL; what changes is how it's run. In a multifile assembly, the manifest is either incorporated into the main file (such as the main DLL file) or as a standalone (see Figure 2.3).

Figure 2.3 Manifest Location within an Assembly



NOTE

Depending on what you are doing, you might want to use a standalone manifest for any multifile assembly. A standalone manifest provides a consistent access location for the manifest and ensures that it will be there when needed. However, constantly referencing the assembly can be a small memory overhead, so its advantage is apparent with larger, multifile assemblies.

Compiling Assemblies

Creating assemblies isn't as hard as it might seem. Compilers are available for all the currently supported .NET languages within the software development kit (SDK). For Visual Basic applications, the compiler is named VBC.EXE (Visual Basic Compiler). Any code you need to run through VBC needs to be saved with the .VB extension. The good thing about this is that you don't need to stick to Visual Studio to create your applications. You can use any text editor you want; as long as you save code with the .VB extension, VBC will compile it for you.

Assembly Cache

The cache on which the CLR relies is called the *machinewide code cache*. This cache is further divided into two subsections: the *global assembly cache* and the *download cache*. The download cache simply handles all the online codebases that the assembly requires. The global download cache stores and deals with the assemblies that are required for use within the local machine—namely, those that came from an installer or an SDK. Only assemblies that have a shared name can be entered into the global assembly cache, since the CLR assumes that these files will be used frequently and between programs.

Even though a file will be used often, however, it could still be sluggish. Since the CLR knows that to enter the global assembly cache, the assembly must be verified, it assumes that it is already verified and does not go through the verification process, thus increasing the time it takes to reference the assembly within the global assembly cache. One integrity check is performed on it prior to entry into the global assembly cache; this integrity check consists of verifying the hash code and algorithms located within the manifest. Furthermore, if multiple files attempt to reference the assembly, a single dedicated instance of the assembly is created to handle all the references, which allows the assemblies to load faster and reference faster across multiassembly situations.

A file that's located in the global assembly also experiences a higher degree of end-user security, since only an administrator can delete files located within the global assembly cache. In addition, the integrity checks ensure that an assembly has not been tampered with, since assemblies within the global assembly cache can be accessed directly from the file system.

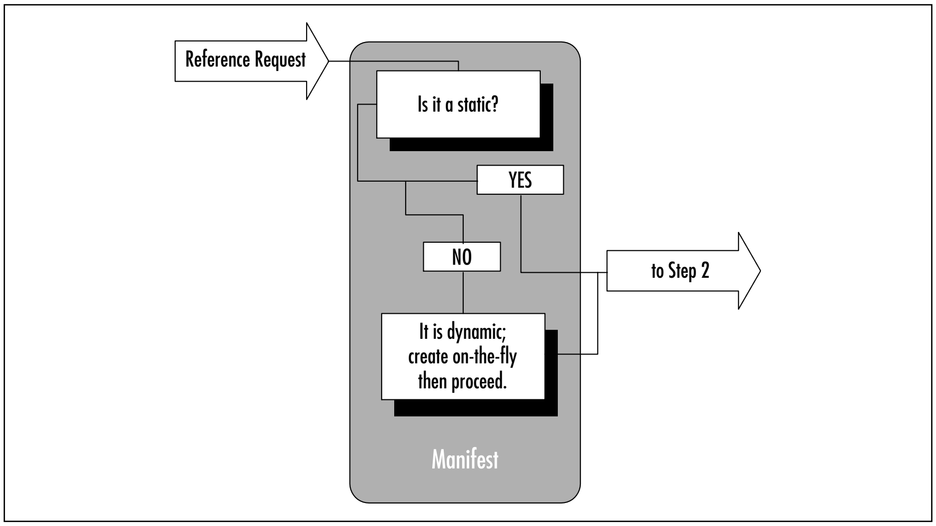
Locating an Assembly

Once the assembly is created, finished, and deployed, its scope is basically private. In other words, the assembly will *not in any way, shape, or form* interfere with any other assemblies, DLL files, or settings that are *not* declared in the assembly's manifest. It's all part of CLR's automation; it used to be that only VB coders had protection from memory leaks or other sorts of problems by inadvertently creating a program that went too far out of its area, but now the CLR handles all that.

Now a single assembly is easy to run, and easy for the CLR to locate. However, when you're dealing with multiple files, you might ask yourself, "Wait—if the assembly is so tightly locked, how can multiple assemblies interact with each other?" It's a good question to ask, because most programmers working with .NET create multifile assemblies, and so we need to understand the process the CLR takes to locate an assembly. It goes like this:

1. **Locate the reference and begin to bind the assembly(ies).** Once the request has been made (through *AssemblyRef*) by an assembly in a multiassembly to reference *another* assembly within the multiassembly, the runtime attempts to resolve a reference in the manifest that tells the CLR where to go. The reference within the manifest is either a static reference or a dynamic reference. A *static reference* is a reference created at build time by the compiler; a *dynamic reference* is created as an on-the-fly call is made. Figure 2.4 displays Step 1.

Figure 2.4 Step 1 of the Location Process



2. **Check the version policy in the configuration file.** The CLR checks to see if there's a configuration file. For client-side executables, the file usually resides in the same directory with the same name, but has a *.CFG extension. For Internet-based applications, the application must be explicitly declared in the HTML file. A standard configuration file can look like the following example:

```
<?xml version = "1.0">
<Configuration>
  <AppDomain
    PrivatePath="bin;etc;etc;code"
    ShadowCOpy="true"/>
  <BindingMode>
     <AppBindingMode Mode="normal"/>
  </BindingMode>
  <BindingPolicy>
    <BindingRedir Name="TestBoy"
                   Originator="45asdf879er423"
                   Version="*" VersionNew="7.77"
                   UseLatestBuildRevision="yes"/>
  </BindingPolicy>
  <Assemblies>
    <CodeBaseHit Name="s_test_mod.dll"
                  Originator="12d57w8d9r6g7a3r"
                  Version="7.77"
                  CodeBase=http://thisisan/hreflink/test.dll/>
  </Assemblies>
</Configuration>
```

The document element of this XML file is Configuration. All this node does is tell the CLR that it's found a configuration file type and that it should look through it to see if this type is the one it needs. The first node contains the *AppDomain* element that has the *PrivatePath* and *ShadowCopy* attributes. *PrivatePath* points to a shared and *private path* to the bin(s) directory(ies). The path is the location of the assemblies that you need and the location of the global assembly cache.

Keep in mind that the *PrivatePath* attribute is relative to the Assembly's root directory and/or subdirectories thereof, and anything outside of that needs to be either in the global assembly cache or linked to using the *CodeBase* attribute of the *Assemblies* attribute. *ShadowCopy* is used to determine whether or not an assembly should be copied into the local download cache, even if it can be run remotely.

The next node contains *BindingMode*. *Binding mode* refers to how the assemblies within the application should bind to their exact versions. *BindingMode* contains the *AppBindingMode* element, which declares the *BindingMode* to be safe or normal. A safe binding mode indicates that this assembly is of the same Assembly version as the others when the application is deployed. No Quick Fix Engineering (QFE) methods are applied, and any version policies are ignored; these characteristics apply to the entire application. *Normal mode* is simply the normal binding process in which the QFE is used and version policies are applied.

NOTE

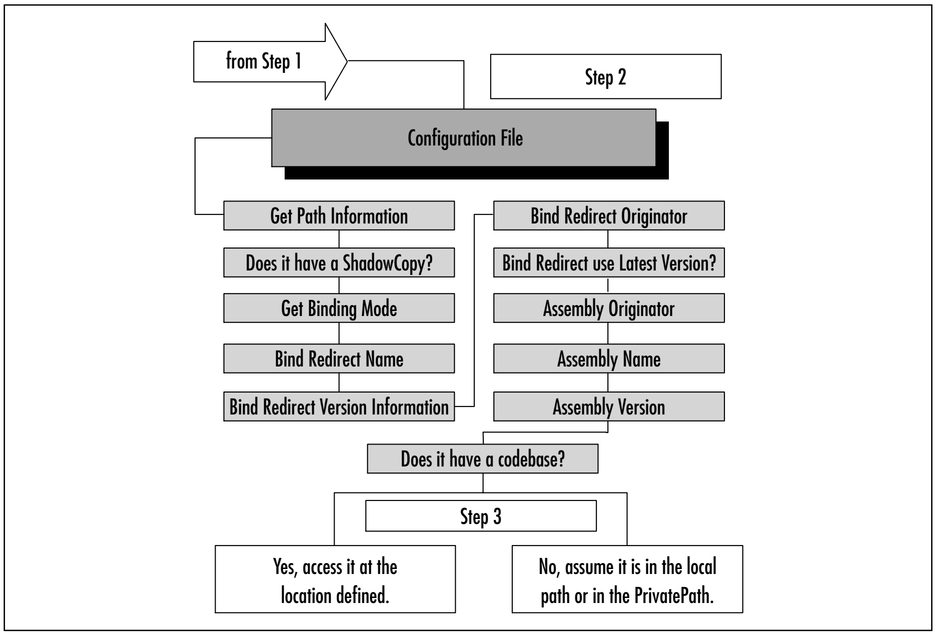
The reference that's checked against from the *AssemblyRef* contains the following information from the assembly it's asking for: text name, version, culture, and originator if it has a shared name. Of the references listed, the location process can work without all of them except the name. If it can't find culture, version, or originator (which only shows up on shared names), it will try to match the filename and then the newest version.

BindingPolicy stores the *BindingRedir* element, which deals with the attributes that tell the CLR which version to look for. This type of element applies only to assemblies that are shared. The *Name* attribute is the assembly's name, *Originator* contains an 8-byte public key of the assembly, and *Version* can either explicitly state which version the assembly should be redirected to or uses a wildcard (*) to signify that all versions should be redirected. *VersionNew* contains the version to which the CLR should be redirected, and *UseLatestBuildVersion* contains a yes/no value that states whether or not the QFE will automatically update it.

Assemblies stores the tags that the CLR can use to locate an assembly. The tags in this element are always attempted before a thorough search.

Name and *Originator* contain the same information that they contain in the *BindingPolicy*. *Version* contains only the current version of the assembly, and CodeBase contains the URL at which the assembly can be located. Figure 2.5 illustrates Steps 2 and 3.

Figure 2.5 Steps 2 and 3 of the Location Process



WARNING

Even though you *can* use partial references, doing so not only kills the whole concept of version support—it can also cause you to use the *wrong* file at times. For example, let's say that you've created a whole new set of classes and need to benchmark the differences. If you are using partial references, it's more than likely that the new version will be picked over the old version. Be precise, even if it's tedious to do so!

3. **Locate the assembly via probing or codebase.** When the information stored in the Configuration file is retrieved, it is then checked against the information contained in the reference and determines whether or not it should locate the file at the specified URL codebase or

via location probe. In the case of a codebase, the URL is referenced and the file's version, name, culture, and originator are retrieved to determine a match. If any of these fails, the location process stops. The only exception is if the version is equal to or greater than the version needed. If it is greater or equal to and all the other references check out, the location process proceeds to Step 4. If no URL is listed for a codebase, the CLR will probe for the needed assembly under the root directory.

Probing is a bit different and more thorough than looking at the URL but definitely more lax in verifying references. When probing begins, it checks within the root directory for a file with the assembly name ending with *.MCL, *.DLL, or *.EXE. If it's not found in the root, it continues to check all the paths listed in the PrivatePath attribute of AppDomain of the configuration file. The CLR also checks a path with the name of the assembly in it. Again, if an error is found, the location process stops, however if it's found and verified, it proceeds to Step 4.

4. **Use the global assembly cache and QFE.** The global assembly cache is where global assemblies that are used throughout multiple programs are found. All global assemblies have a shared name so that they can be located through a probe. *Quick fix engineering*, or QFE, refers to a method in which the latest build and revision are used. It's done this way to allow greater ease for software vendors to provide patches by recreating just one assembly instead of the whole program. If the assembly was found and the QFE is off, the runtime double-checks in the global assembly cache with a QFE for the particular assembly; if a greater revision/build is found, that version takes the place of the one found while probing.
5. **Apply the administrator policy.** At this point, any versioning policies are applied (versioning policies are stored in the admin.cfg file of the Windows directory) and the program is run with the policies applied. The only major impact this policy has occurs if an administrator policy initiates a redirect to a version. If this happens, the version must be located in the global assembly cache before the redirect occurs. The runtime assumes that since the redirect is administrative, the user manually and consciously set it and that the user already has supplied the necessary file in the global assembly cache.

Private Assembly Files

Private assembly files are normally single applications, that reside in a directory without needing to retrieve any information or use resources from an assembly that is located outside its own folder. This does not mean that the private assembly can't access the standard namespaces, rather it simply means that they do not use or require any other external applications to properly function. These types of assemblies are useful if the assembly will be constantly reused and does not rely on any other assembly. Private assembly files are not affected by versioning constraints.

Shared Assembly Files

Shared assembly files are generally reserved for multiassembly applications and store commonly used components, such as the graphical user interface (GUI) and/or frequently used low-end components. These assemblies are stored in the global assembly cache, and the CLR does enforce versioning constraints. Examples of a shared assembly are the built-in .NET Framework classes.

A shared assembly, as you might have guessed, is the exact opposite of a private assembly. A shared assembly does stretch outside the bounds of its directories and requires resources that are found within other assemblies. Shared assemblies are utilized heavily when dealing with modular applications. For example, a GUI that is used between several applications can be stored as a shared assembly or a commonly used database routine.

Understanding Metadata

When you create your assembly, two things happen: Your code is transformed into MSIL, and all the relevant information contained in the code (types, references, and so on) are noted within the manifest as metadata. The CLR then inserts the metadata into in-memory data, and uses it as a reference in locating what is needed according to the program. This road map provides a large part of interoperability, since the CLR doesn't actually need to know what code it's programmed in; it simply looks at the metadata to find out what it needs and where it's going. The metadata is responsible for conveying the following information to the CLR:

- Security permissions
- Types exported
- Identity

- External assembly references
- Interface name
- Interface visibility
- Local assembly members

The Benefits of Metadata

The items in metadata are placed within in-memory data structures by the CLR when run. This allows metadata to be used more freely with faster access time. This system enhances the self-describing functions of .NET assemblies by having readily available all the items that the assembly requires. This also allows for other objects (per the metadata, of course) to interact with the assembly.

Metadata also allows interoperability by creating a layer between the assembly's code and what the CLR sees. The CLR uses the metadata extensively, thus removing the burden of operability from the CPU/language. The CLR reads, stores, and uses the metadata through a set of APIs, most notably the managed *reflection* and *reflection emit* services. The layer abstraction causes the runtime to continue optimizing in-memory manifest items without needing to reference any of the original compilers and enables a snap-in type of persistence that allows CLR binary representations, interfacing with unmanaged types, and any other format needed to be placed in-memory.

You might have been surprised when you saw that the metadata allows unmanaged types to show up; however, this does not impact the CLR in any way. Unmanaged metadata APIs are not checked nor do they enforce the constraints present. However, the burden of verifying unmanaged metadata APIs is placed solely on the compiler.

NOTE

PEVerify is a command-line tool enclosed with the .NET Runtime SDK that checks for you the CLR Image within the PE's manifest during development. Use it if you wind up migrating VB 6.0 code and have doubts as to its portability or performance.

Identifying an Assembly with Metadata

Metadata identifies each assembly with the following: name, culture, version, and public key. The *name* used is the textual name of the assembly or the name you gave it when you created it. The *culture* simply references the cultural settings used such as language, time zone, country/region, and other localization items. The *public key* used is the same one generated by the assembly.

Types

In unmanaged code (i.e., VB 6.0), we referred to types as *objects*. Types, like objects, contain data and logic that are exposed as methods, properties, and fields. The big differences between the two lie in the properties and fields; *properties* contain logic in order to verify or construct data, whereas *fields* act like public variables. *Methods* are unchanged. Types also provide a way to create two different representations with different types by looking at the two different types as part of the same interface—in other words, they have similar responses to events.

Currently two types are available to .NET users: value types and reference types. *Reference* types describe the values as the location of bits and can be described as an object, interface, or pointer type. An *object type* references a self-describing value, an *interface type* is a partial description that is supported by other object types, and the *pointer type* is a compile-time description of a machine-address location value.

When dealing with classes, the CLR uses any method it deems fit, according to the Common Type System. Metadata has a special mark for each class that describes to the CLR which method it should use. Table 2.3 lists the layout rules that metadata marks for each class.

Table 2.3 Class Layout Rules

Class	Layout Rules
AutoLayout	CLR has free reign over how the class is laid out; this shows up more often on the inconsequential classes.
LayoutSequential	CLR guides the loader to preserve field order as defined, but offsets are based on the field's CLR type.
ExplicitLayout	CLR ignores field sequence and uses the rules the user provides.

Defining Members

Members are the methods, fields, properties, events, and nested types that are found within a type. These items are descriptions of the types themselves and are defined within the metadata. This is one of the reasons that access of items through metadata is so efficient.

Fields, arrays, and values are subvalues of a value representation. Field subvalues are named, but when accessed through an index they are treated as array elements. A type that describes the values composed of array elements creates a true array type with values of a single type. Finally, the compound type is a value of a set of fields that can hold fields of different types.

Methods are operations that are associated with a particular type or a value within the type. For security purposes, methods are named and signed with the allowed types of arguments and return values. Static methods are methods that are tied directly to the type; virtual methods are tied to the value of the type. The CLR also allows the *this* keyword to be null within a virtual method.

Using Contracts

The signature that methods use is part of a set of signatures referred to as a *contract*. The contract brings together sets of shared assumptions from the signatures between all implementers and users of the contract, providing a level of check and enforcement. They aren't real types but rather are the requirements that a type needs to be properly implemented. Contract information is defined within the class definition.

Class contracts are one of the most common. They are specified within a class definition and in this case defined as the class type along with the class definition. The contract represents the values and other contracts supported by the type and allows inheritance of other contracts within other types.

An *interface contract* is defined within an interface. Just like the class definition, an interface definition defines both the interface contract and the interface type. It can perform the functions that a class contract can, but it cannot describe the representation of a value, nor can it support a class contract.

A *method contract* is defined within a method definition. Just like a normal method, it's an operation that's named and specifies the contract between the method and the callers of the method. It exerts the most control over parameters, specifying the contract for each parameter in the method that it must support and the contracts for each return value, if there is one.

A *property contract* is defined within a property definition. The property contract specifies the method contract used for the subset of operations that handle a named value, including the read/change operations. Each property contract can be used only with a single type, but a type can use multiple property contracts.

An *event contract* is defined in an event definition. It specifies method contracts for the basic event operations (such as the activation of an event) and for any operations implemented by any type that uses the event contract. Like the property contract, each event contract can be used only with a single type, but a type can use multiple event contracts.

Assembly Dependencies

An assembly can depend on another assembly by referencing the resources that are within the scope of another assembly from the current assembly scope. The assembly that made the reference has control over how the reference is resolved, and this gives the assembly mapping control over the reference onto a particular version of the referenced assembly. When you depend on an external assembly, you can choose to let the CLR assume that the files are present in the deployed environment or will be deployed with the corresponding assemblies. Such an assumption can be pretty large or problematic, but the CLR is smart enough to know what to do if it's not there.

Unmanaged Assembly Code

There are two things that you can do as far as unmanaged code goes—you can export COM components to the framework or you can expose .NET components to COM.

To export a COM into .NET, you will need to import the COM type library, but remember that a COM library file can be either the standard TLB file, a DLL file, or an EXE file. Convert the code into metadata by using either Visual Studio.NET or the Type Library Importer tool. Visual Studio.NET will automatically convert the COM library into a metadata type library while the Type Library Importer tool uses a command-line interface that lets you adjust a couple more parameters than Visual Studio.NET. Define your newly created COM metadata type in your assembly and compile it with the /r flag pointing to the dll containing the unmanaged types. Most programmers suggest that an assembly that works with COM be deployed into the Global Assembly Cache.

If the need should arise to expose .NET components to COM you can, but it is *not recommended* since you will lose all of the features the .NET framework has given your code. In fact, if you can avoid it completely for now, do so and just upgrade your code to .NET or rewrite it completely.

First determine which types are needed for the export. The classes you are planning to export must match the following criteria:

- Must have a Public Constructor
- All methods, properties, and events must be public
- Classes need to implement interfaces implicitly
- All managed types must be public

Since .NET won't expose anything that is not public, it will not export anything that is not public. If you have an error with an exported .NET component that has a missing class, file name, or run-time initialization error, you may want to go back to your .NET source and figure out if you have fulfilled all of the above requirements.

The tricky part now is using the *System.Runtime.InteropServices* namespace. There are 3 COM classes within this namespace that are used to set the values needed for your particular COM export and the rest of the classes give your assembly COM-like attributes. Once your assembly has been properly checked and assembled, compile it and export it using the TypeLibraryExport.Exe tool.

Now that you've prepared the file, you will need to register the exported assembly(ies) with COM. RegASM.exe (Register Assembly) is a command-line tool that can register the assembly(ies) needed into the Microsoft System Registry so your export will have its own CLSID. Once the exported item has been registered, you can proceed to use this new object within your application.

Reflection

The concept of *reflection* is available to the user via the *System.Reflection* namespace. In essence, reflection reflects the composition of other .NET code back to us. It can discover everything that is vital within the assembly, such as the classes, events, properties, and methods exposed by the assembly. We can then use this information to clone an instance of that assembly so that we can use the classes and methods defined there.

NOTE

You might have used reflection in VB 6.0 via TypeLib.DLL; however, TypeLib was limited in that it had to create the "clones" using the IDL description provided by COM, which can give inaccurate or incomplete clones. Since all the information for "cloning" is available directly from the manifest, we don't have to worry about that anymore.

Using reflection can theoretically provide access to nonpublic information such as code, data, and other information that is normally restricted due to isolation. .NET provides a built-in check system of rules to determine just what you can get using reflection. If you really have to use nonpublic information, you need to use *ReflectionPermission*. *ReflectionPermission* is a class located within Object.CodeAccessPermission namespace and gives access to all the nonpublic information when requested by a reflection. This class can theoretically also give someone the ability to view your code, so *do not use this class* if you can avoid it! You definitely will not want to use this ability on Internet applications. By default and without needing permission, reflection can access or perform the following:

- Public types
- Public members
- Module/assembly location
- Enumerate assemblies and modules
- Enumerate nonpublic types (have to be in the same location as the assembly using reflection)
- Enumerate public types
- Invoke public, family access (of calling code class), and assembly access (of calling-code class) members

Attributes

More a C++ concept than a VB one, an *attribute* allows you to add descriptive declarations that behave similarly to keywords. You can use attributes to annotate types, methods, fields, properties, and other programming elements. They are stored within the metadata and can help the CLR understand the description of your code. Attributes can describe the way that data is serialized, describe security characteristics, or limit JIT compilation for debugging purposes. Perhaps one of the most versatile of the metadata items, attributes can even add descriptive elements to your VB code to affect its runtime behavior. A simple attribute may be used like this:

```
Public Class <attribute()> ClassName
```

In this example, the class *ClassName* is described by the attribute *attribute()*. This means that when the CLR hits this class, it will alter its behavior according to what *attribute()* says.

Ending DLL Hell

Everyone knows what DLL Hell is: It's the situation that occurs when an older or newer DLL file overwrites the previous copy after the installation of a new application (usually a newer DLL that is not backward compatible). Registry settings are changed; some are added, some are removed, and some are altered. GUIDs could change and, at the blink of an eye, all these things create a situation where one DLL file prevents your application from working. In order to prevent DLL Hell, the .NET Framework takes the following steps:

- Application isolation is enforced.
- "Last known good" system from Windows NT systems is enforced.
- Side-by-side deployment is permitted and backed up by isolation.
- File version information is recorded and enforced.
- Applications are self-describing.

Side-by-Side Deployment

Side-by-side execution allows two different versions of the same assembly file to run simultaneously. This is an advantage of the isolation provided to each assembly. Side-by-side deployment removes the dependency on backward compatibility that often causes DLL Hell. Side-by-side execution can be running either on the same machine or in the same process.

Side-by-side deployment in the same process can be the most strenuous to code for; you have to write the code so that no processwide resources are used. The extra work pays off in that you can run multiple components and objects in the same thread, allowing for greater process flexibility and usage.

Side-by-side deployment on the same machine puts less stress on the code writer but still has its quirks. The biggest point to look out for when coding this way is to write in support for multiple applications attempting to use the same resource; you can work around this by removing the dependency on the resource and allowing each version to have its own cache.

Versioning Support

Versioning is the method .NET uses with assemblies that have a shared name; it tells the CLR the version of the particular assembly. Each assembly has two types of version information available: the compatibility version and the informational version. The *compatibility version* is the first number, which the CLR uses to determine identities. The *informational version* allows for an extra string description of the assembly that the CLR doesn't really need.

The version number looks like your typical version—a four-part number that describes, in order, the major build version, the minor build version, the build, and the revision. If there are any changes to the major or minor versions, the assembly is used as a separate entity and isolated. The build and the revision signify a build compatible to the present assembly, which means that this new version contains a bug fix or patch.

The major and minor numbers are used to perform incompatibility checks. In other words, compatibility is weighed against the major and minor numbers, and any difference in either of these two numbers tells the runtime that it is a new release with many changes and should be treated accordingly. The build number tells the runtime that a change has been made, but does not carry a high incompatibility risk. It's been my experience that relying on the build number at times is very bad practice, especially if the minor change involves your types. In fact, whenever you change anything, such as how a class is referenced, you should treat it as a major/minor revision unless you *absolutely* take all the necessary steps to make the class backward compatible.

When you do create a backward-compatible class, try to create it as a bug fix or patch and define the change in the QFE. That way, the runtime assumes backward compatibility is in place, since there should be no major changes (again, such as class references), and uses it accordingly unless it is explicitly told not to use it by a configuration file.

Using System Services

System services combine everything that the runtime makes available, such as exception (error) handling, memory management, and console input/output (I/O). Some of the topics discussed here might not be new to some VB programmers, especially those who have had some exposure to Java or C/C++.

The big change that VB programmers can look forward to is how exception handling is approached. The way we used to do it involved thorough use of the debugger and then praying for the executable to not throw an arcane runtime error. Now we can actually catch any errors thrown and handle them properly. This also means that we have a better method for tracing error messages.

Memory management really hasn't changed significantly; only the way it's implemented has changed. Instead of programmers having full control over object instantiation and destruction, the CLR takes over that task. However, we do have the ability now to create standard command-line programs—something that VB never had before.

Exception Handling

.NET introduces the implementation of a try/catch system through its new Exception object. Some of you may be already familiar with this concept from previous JAVA work. A simple try/catch statement can look like the following.

```
Try
{
Thiswillcrash();
}
Catch(error_from_Thiswillcrash)
(
//react to the error thrown by Thiswillcrash()
}
```

So, in essence, a *try/catch* set will place the function or sub within a *try* wrapper that will monitor any error messages. If an error message matches *error_from_Thiswillcrash* then the *catch* wrapper generates the appropriate response to the error. This will give programmers more flexibility in determining errors and how they want to handle the error instead of letting Windows do it and hoping for the best.

Within a DLL file you have a standard file read and file write system. However, instead of just generating a failure error if the file that needs to be read is not found, you would rather just display a message that says "this file is being created" and then creates the file without the user even knowing that an error occurred. A simple way of doing a try/catch for this situation may appear like the following:

```
Try
{
FileReadDisplay();
}
Catch(File_not_found_error)
{
//display message "This file is being created"
//create file that matches needed defaults
//display message "A new default file has been generated.
//Please reset your defaults."
}
```

The try/catch system is part of the *Exception* class. While it's a pretty neat ability to finally have in VB, the *Exception* class also brings with it some extra goodies for debugging, including *StackTrace*, *InnerException*, *Message*, and *HelpLink*.

StackTrace

Stacks haven't changed over the years; a *stack* is still a special type of data structure in which items are removed in the reverse order in which they are added (last in, first out, or LIFO). This means that the most recently added item is the first one removed. *StackTrace* allows you trace the stack for errors. It is most useful in dealing with constant errors along loops and within a try/catch statement. *StackTrace* is useful when it is defined before a try statement and when it ends after the catch statement.

InnerException

An *InnerException* can store a series of exceptions that occur during error handling. You can then format the series of exceptions into a new exception that contains the series. It's almost like a waterfall view, because an exception is thrown, which in turn throws another exception. Using *InnerException*, the first exception would be stored within the last exception and so on, giving the developer an ample road map to locating the starting point of an error.

Message

Message stores a more in-depth error description. This is extremely useful when used in conjunction with *InnerException*.

HelpLink

Using *HelpLink*, you can set a specific URL or URN within a try/catch block to point to an article or help file that has more details on the error generated.

Garbage Collection

Memory usage and clean-up have always been valuable features of VB, mainly due to VB's preventive method of initializing and destroying its objects. Garbage Collection is .NET's method for handling object creation and destruction as well as cleanup and preventive maintenance. Garbage Collection does not rely on reference counting, as VB 6.0 and previous versions do; it has its own unique system for detecting and determining which objects are no longer in use. In this sense, .NET is smart enough to know when a file is being used and when it needs to be removed. We delve into a full overview of Garbage Collection in the Relying on Automatic Resource Management section later in this chapter.

Console I/O

We finally have the ability to create console programs in VB! Much of this ability comes from .NET's Microsoft Intermediate Language (MSIL) system. Console applications are those little programs that pop up a DOS box and run from the command line. Command-line applications can be used in middle-tier situations, in testing a new class, or even for creating DOS-based functionality for a utility tool. We have this ability thanks to the *System.Console* namespace. (We discuss namespaces later in this chapter.) Here's a brief example of a simple command-line VB application:

```
Import System.Console


Sub Main()
   Dim readIN as String
   WriteLine("This is a line!")
```

```
    ReadIN = ReadLine()
    WriteLine(ReadIN)
End Sub
```

The console would print *This is a line!* with a carriage return at the end automatically, giving us one line to write whatever we want. After a carriage return is detected, what we wrote is stored within the variable *ReadIN* and then displayed via *WriteLine*.

Microsoft Intermediate Language

Once your assembly is in managed code, the CLR in turn translates the code to the MSIL. MSIL is a type of bytecode that gives .NET developers the necessary portability, but it is also key to the system's interoperability, since it provides the JIT compiler with the information it needs to create the necessary native code. MSIL is platform independent.

MSIL also creates the metadata that is found within an assembly. Both the MSIL and metadata are stored within an extended and modified version of the PE (which is more a combination between PE's syntax and the Common Object File Format, or COFF, object system). MSIL's flexibility allows an assembly to properly define itself and declare all it needs for self-description.

The Just-In-Time Compiler

Without the *just-in-time* (JIT) *compiler*, we wouldn't have any functioning .NET programs. The JIT turns the MSIL code into the native code for the particular platform on which it's running. Each version of .NET for each individual platform also includes a JIT for that specific platform architecture. For example, an x86 version of .NET can compile .NET code from a non-x86 architecture because the JIT on the x86 machine translates the MSIL into x86-specific code, since the MSIL contains no platform-specific code.

JIT's method of code compilation is literally just in time—it compiles the MSIL code as it's needed. This method guarantees faster program loading time and less overhead in the long run, since JIT compiles what is needed when it's needed. MSIL, when created and referenced, creates a stub to mark the methods within the class being used. JIT compiles just the stubbed code and replaces the stubs within the MSIL to the location of the compiled code address.

There are currently two flavors of JIT: normal JIT and economy JIT. Economy JIT is geared toward intensive CPU/RAM usage systems, such as Windows CE platforms. Economy JIT differs from normal JIT in that, in order to make the best of the intensive CPU/RAM usage situation, it replaces the stubs in the MSIL with the actual compiled code, not a reference to its address. Microsoft currently claims that economy JIT is less efficient than normal JIT for this reason. However, a decent benchmark exam of these two compilers has yet to be done.

Using the Namespace System to Organize Classes

We've already seen an example of namespaces in the previous code example, but what are they? *Namespaces* are references that we place within the code that point to the location of the object or class that we need to use within the .NET Framework. In the previous code example, we used the *System.Console* namespace. This naming scheme is used only for organizational purposes, but it is vital that you understand it.

A namespace is basically a hierarchical system created to organize intrinsic classes that provide the basic functions that come with .NET. Each class is kept within a namespace that suits its use; for example, Web-related classes are kept within the *System.web* namespace. Each namespace can contain namespaces, providing more functionality for each namespace. The system namespace is the root namespace on all .NET machines.

VB 6.0 users are already familiar with this concept from COM as the PROGID (the name of the component and class within COM) *component.classname*. VB 6.0 users are also familiar with COM's limitations, such as PROGID naming not allowing more than one level in depth and that its name was global to the computer. .NET, however, allows for multiple namespaces, classes, interfaces, and other valid types declared within it. The following example displays a sample namespace that contains multiple assemblies and an assembly that is stored within a namespace:

```
MyNamespace.namespace.class
MyNamespace.enum
MyNamespace.interface.class
MyNamespace.Namespace.class
```

Here we have the *MyNamespace* base namespace with multiple namespaces that in turn contain all the needed operations, functions, and procedures to provide necessary services. Each namespace can have classes that have the same name; for example, *Assembly3* and *Assembly5* can both have a *count* class. However, within a single namespace there cannot be any duplicate class names. Namespaces can also be local or global; local namespaces can be seen only by the current application, and global namespaces can be seen on the entire machine.

The Common Type System

The *Common Type System* (CTS) gives the CLR a description of the types that are supported and used and how they are presented in metadata. The type in CTS represents the type system, which is one of the more important parts of .NET for cross-language support. The type provides the rules and logical steps that a language compiler employs to define, reference, use, and store information. If you are using any CLR-compliant compiler outside of the .NET Framework, it must use the CTS system to properly create the assembly. The type system that the CTS uses contains classes, interfaces, and value types.

A class is now contained within a type. In fact, the term *type* is sometimes used (although sometimes erroneously) with the same meaning as *object* to reflect .NET. The term still has the same functionality as in any other object-oriented programming (OOP) language. It can define variables, hold the state of objects, perform methods and events, and create, set, and retrieve properties. Every time an instance of a .NET class is created, it is treated as an object; you can use it in the same style that you would use objects in VB 6.0, by accessing its properties, events, and fields. Table 2.4 displays the characteristics of a class. Table 2.5 displays the characteristics of the members.

Table 2.4 Class Characteristics

Class	Characteristics
Sealed	Class derivations are prohibited.
Implements	Interface contracts are fulfilled by this class.
Abstract	This class can't be instantiated on its own; in order to use it, you must derive a class from it—just like abstract classes in C/C++.
Inherits	This means that the class being defined will inherit the characteristics (i.e., properties, fields, methods) of the class that is written next to it. You can use the same characteristics or override them.
Exported	This class can be viewed outside the assembly.
Not-Exported	This class cannot be viewed outside the assembly.

Table 2.5 Member Characteristics

Members	Characteristics
Private	Defines accessibility as permitted only within the same class or a member of a nested class within the same class.
Family	Defines accessibility as permitted within the same class as the member and subtypes that inherit it.
Assembly	Defines accessibility as permitted only from within the assembly in which the member is implemented.
Family or Assembly	Defines accessibility as permitted only by a class that qualifies as a family or an assembly.
Public	Defines accessibility as permitted from any class.
Abstract	A nonimplemented member; as with C/C++, you have to derive a class from it in order to implement it.
Final	A method with the final statement cannot be overridden; this helps prevent any unintentional overrides that can damage functionality.
Overrides	Used by virtual methods; it replaces the predefined implementation from the derived class.
Static	A method that is declared static exists without needing to be instantiated and can be referenced through all class instances.
Overloads	An overloaded method has the same name as another method and the same code, but its parameters, order of parameters, or calling convention may be different. This is useful for adding last-minute functionality to a method that you might only need once.
Virtual	Used to create a virtual method in order to have the functionality provided by Overrides.
Synchronized	Limits usage of implementation to one thread at a time.

NOTE

The Virtual Execution System is tied in with the CTS concept. In fact, it's a special execution engine that was created just to ensure that the tenants of the CTS are implemented.

Developing & Deploying…

Abstract Classes?

If you've never used C/C++, abstract classes might be a foreign concept for you. An *abstract class* can be defined as a *skeleton* class that has no actual code within it—simply a declaration of what a class that can be derived needs to have within its structure to be considered a derivative of the skeleton. In other words, the flesh on the bones is added later.

Abstract classes are useful when you need to create some sort of base class that needs to be reused but have no need for it later—similar to a blueprint. For example, take the abstract class fruit_eater:

```
Abstract class fruit_eater
{
   Private Me_eat As Integer
   Me_eat = 1

   Public Property Eat() As Integer
    Get
       Return Me_eat
    End Get
   End Property

End Class

Public class monkey_boy
  Inherits fruit_eater

  Public Property me_do_eat() as String
     If Eat = 1 Then
         'code goes here to tell you that monkey_boy eats
fruit!
     End If
```

Continued

```
      End Property


    End Class
```

Using the abstract class fruit_eat, we set a requirement that class monkey_boy must have to say that monkey_boy eats fruit. This can be further expounded to another class, animal_kingdom, that can use fruit_eat to organize between herbivores and carnivores within its kingdom of wild animals and monkey_boys.

Type Safety

Type safety limits access to memory locations to which it has authorization. So, if we have Object A trying to reference the memory location of Object B that is within the memory area of Assembly C, Object A will not be allowed access. Even if Object A tries to access a memory location that is accessible by its assembly and does not have permission, it will be denied. An optional verification process can be run on the MSIL to verify that the code is type safe. It's optional because it can be skipped based on permissions given to the code.

Type-safe code tells the runtime that it can go ahead and isolate the code, since it's not going to need anything outside its boundaries. Even if the trust levels are different within a type safe code, it can execute on the same process. Code that is not type-safe might cause crashes in the runtime or even shut down your whole system, so be careful with it. Remember, we're working with a beta runtime, and it can be touchy!

Relying on Automatic Resource Management

We are now getting to the nuts and bolts of .NET. So far, we've discussed enhancements and changes in semantics. However, memory management in .NET is radically different. Previously, we used the deterministic finalization system, in which we declare that the code ran on the class initialization and termination plus had control over where a class was terminated. Deterministic finalization had its drawbacks, because if the programmer forgot to declare the class empty (null, in some cases) or simply forgot to run the termination event, we'd have a memory leak or worse when control over the project terminated.VB's system of destroying classes once the class count reached zero caused some problems when the last instance of a class was referenced by the last instance of another class, and neither class would technically reach zero, so no cleanup was done.

This outdated memory management system is referred to as *reference counting*. A count is kept within each object, usually in its header, of how many references there are for the object. Each application (or client, as it is referred to in COM circles) that is referencing an object states when it is referencing the object and when it is releasing the object. As new objects are instantiated, the count (or number of objects in the count) is incremented and decremented when the object is either overwritten or recycled.

The burden of doing the actual cleanup of the object, however, was not on the application. All the application did was merely issue the destroy command to the object, and the object then had to free itself from the reference count. When an object was not properly deallocated (destroyed), we had an instance of a memory leak. Reference counting also had a limited growth size, because objects became bloated (made bigger artificially) in order to store the reference count, and of course cyclic objects generated the previously mentioned nonzero reference count.

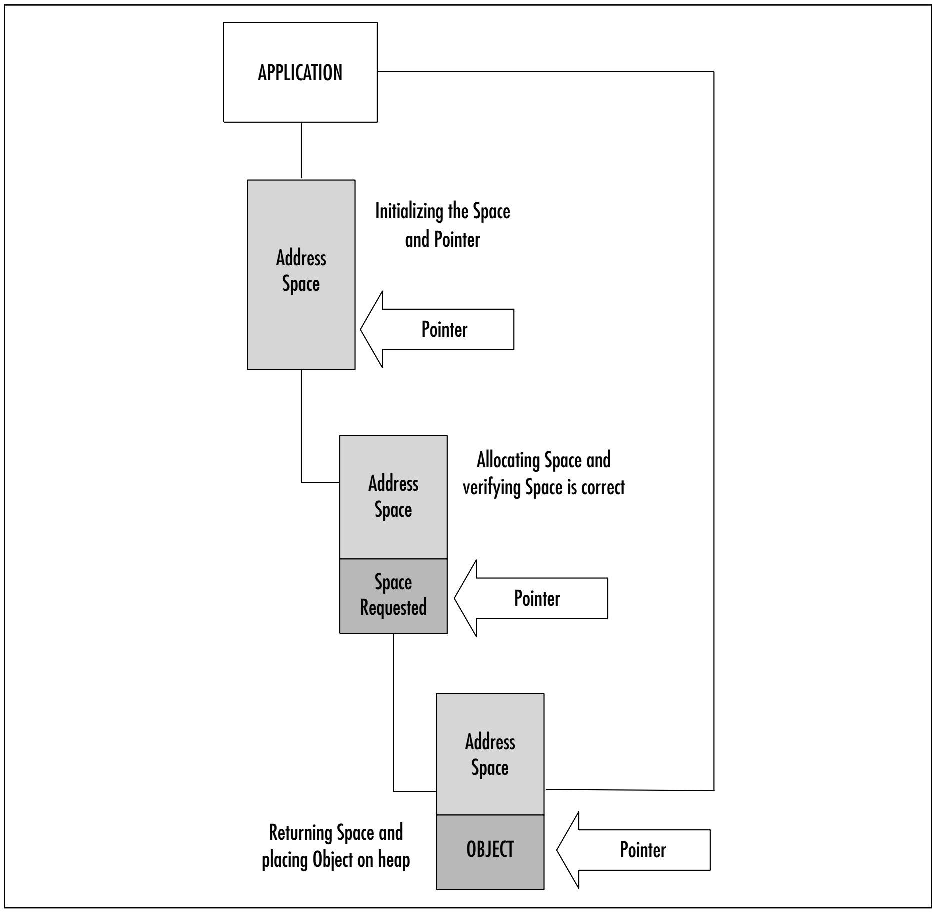
.NET replaces all this with *automatic resource management*. The runtime is now smart enough to know when and how to handle memory allocation, deallocation, and usage. A major drawback is that we can't control when an object or a class is terminated, and therefore we have no knowledge of when the termination takes place. This is a very valid point and, quite honestly, the only noticeable drawback because it won't release the memory and so we encounter a dead reference. However, most of the time this won't matter, because Garbage Collection will eventually get to it. Now let's see how .NET handles memory and how this relates to Garbage Collection.

The Managed Heap

When a program is run in .NET, the runtime creates the region of address space it knows it needs but does not store anything on it. This region is the *heap* (also referred to as the *free store* or *freestore*). .NET controls the heap and determines when it's time to free an object. Figure 2.6 presents an illustration of the following pointer interaction process:

1. A pointer is created for the allocated space (heap) that keeps track of the next available free area on the allocated space that the runtime can use for storage.
2. As the application creates new objects, the runtime checks to see if the space currently being pointed to can handle the new object. If it can't, it dynamically creates the space.
3. Then the object is placed on the heap, its constructor is called, and the new operator returns the address block of our newly created object.

Figure 2.6 Pointer Interaction with a Managed Heap


APPLICATION
Address Space
Initializing the Space and Pointer
Pointer
Address Space
Allocating Space and verifying Space is correct
Space Requested
Pointer
Address Space
Returning Space and placing Object on heap
OBJECT
Pointer

NOTE

When an object/type is over 20,000 bytes, a special *large heap* is created to store them. This special heap does not go through compression when Garbage Collection is called. Compression occurs during the generation process, described in a later section in this chapter.

Garbage Collection and the Managed Heap

As mentioned, .NET handles the managed heap by using Garbage Collection. In its purest sense, Garbage Collection is an algorithm designed to determine when the life cycle of an object has ended. In order to determine if an object is at or near its end, Garbage Collection analyzes the root of the object. Roots (also known as *strong references*), much like the actual roots found in nature, act as road maps to where vital resources, such as objects, are stored. Global or static pointers, local variables that are on a thread stack, and CPU registers containing pointers to the heap are all considered roots. All the roots that are visible are stored in a list created and updated by the JIT and CLR.

Once Garbage Collection starts, it assumes that all the roots available to the heap are null. This makes the Garbage Collection begin a verification process in which it goes through each root recursively and starts to make a graph that contains all the references available and any linked references (i.e., Object A references Object B). This step is repeated once more to make sure that everything is in place by assuming that if it's a duplicate object, it's already on the list and thus a legitimate object, meaning that the graph it just built is correct. The final step of this verification process is that Garbage Collection starts to trace the root of each object to determine if the root is coming from the program that is going to use the current address space. Any objects without roots are considered null or no longer in use and are treated as garbage, which is an accurate assumption since no two applications share the same address space, and are promptly removed from the heap. You can also manually invoke Garbage Collection. It's not necessary to do that since Garbage Collection works automatically, but it's useful for those times that you find an object that needs to be destroyed immediately (such as an object that needs to be reset by destroying it and recreating it immediately). You can manually invoke Garbage Collection as follows:

```
System.GC.Collect()
```

This code automatically starts Garbage Collection. However, it eventually creates overhead if used repeatedly, so it's best to use it sparingly. Roots also provide the fix to memory leaks and stray resources. The runtime uses the roots to determine when an object or resource is no longer in use, enabling Garbage Collection to clean them up. Now that we know how Garbage Collection works, let's take a look at just what the Garbage Collection namespace offers (see Table 2.6).

Table 2.6 The Garbage Collection (GC) Namespace

Property/Method Type	Method	Description
Properties—public static	MaxGeneration	Lists the generations that the system can support.
	TotalMemory	This method displays the total byte space of alive objects and can occasionally overlap objects that will be released soon. This method is used frequently for high-usage areas, especially the areas that contain expensive and/or limited resources, such as CE.
Methods—public static	Collect	An example of an overloaded method; it forces a collection of all available generations. Can be useful in building your own garbage collection system for your particular application by analyzing available generations. You can then use this information to force any objects into a disposal.
	GetGeneration	Another overloaded method; it returns the specific generation that an object is in.

Continued

Table 2.6 Continued

Property/Method Type	Method	Description
	KeepAlive	A method that assists in migrating VB 6.0 code to VB.NET. Using KeepAlive, you can tell GC that this object does not get recycled, even if there are no roots to it from the rest of the managed cod by sending GC a "fake" alive response.
	RequestFinalizeOnShutdown	This method is an implemented workaround to a bug in the beta1 Framework; the .EXE engine usually shut downs without calling a finalize routine. This method causes all finalization that needs to be done on shutdown.
	SuppressFinalize	This method simply tells the system to not finalize a object. Very useful for helping GC "skip" prefinalized objects (objects that have been manually finalized) and thus keeps GC from wasting time on something that's not there.
	WaitForPendingFinalizers	A really buggy implementation of a good idea. This method suspends the current running thread until all finalizers in the queue are run. However, since running a finalizer almost always kicks in a GC, this method causes a circular

Continued

Table 2.6 Continued

Property/Method Type	Method	Description
		loop that will keep waiting for finalizers as new finalizers are created. This method would be much more useful if it could target generations instead.
Methods—public instance (all of these methods are inherited from System.Object namespace)	Equals	Checks to see if the object being evaluated is the same instance as the current object.
	GetHashCode	Returns the hash function for a specific type.
	GetType	Returns the type from an object.
	ToString	Returns a string to represent the object.
Methods—protected instance (all of these methods are inherited from System.Object namespace)	Finalize	Allows cleanup before GC gets to it. However, the CLR can decide to "ignore" this command, as when the root is still active or it's considered a constantly used resource.
	MemberwiseClone	Creates a copy of the current object's members.

We can use the methods and properties inherent to the Garbage Collection namespace to formulate a workaround to Garbage Collection having full control over the disposal of objects. (Remember, the runtime controls the memory allocation through Garbage Collection; that includes the destruction of objects.) An example of this code would be:

```
Imports System


'class/module/assembly code here to do whatever you want
```

```
'please note that this is just an example and is non-functioning.
' there is a very good functional example of this similar process
' available in the .NET SDK Samples under the GC/VB folder of the
' SAMPLES directory.

'now that we have the objects / resources set, let's create a typical
' Dispose class.

Public Class DisposeMe
Inherits Object

   Public Sub Dispose(objName as String)
   'objName would be received by previously using the
   'ToString Public Instance Method and storing the value in a string.

   Finalize

   GC.SuppressFinalize(objName)
   End Sub

   Protected Overrides Sub Finalize()
    ' no clean-up code needed; this will cause Finalize to be run
   End Sub
End Class


'note the use of SuppressFinalize to keep the GC from repeating itself.
```

Congratulations! We've just resolved one of the basic problems of Garbage Collection. With this example, we can successfully control manual termination of objects and resources. It's best to reserve this type of workaround for intensive resources.

Debugging…

Don’t Use a Raw Finalize Method!

Garbage Collection allows a small emulation of the Class_Terminate event via the *finalize method*. However, the finalize method does *not* supercede the authority of the Garbage Collection/CLR, and it may not be instantly implemented if the Garbage Collection/CLR assume that the resource/object is still needed or in use. It could very well be a couple of calls too late before it’s shut down. This is especially frustrating when you need to remove an object for program flow. Finalized objects:

- Are promoted to older generations causing unnecessary heap usage
- Have longer initialization times
- Are out of your control as to when and where they are actually terminated
- Cause any other objects that are associated with them to be finalized, adding more strain to the heap
- Can prolong the lifetime of other objects that are referenced from the finalized object

For these reasons, it is better to avoid using finalize by itself. If you determine that you must use it, make sure that you avoid all actions that could interfere with the finalize code, such as creating an instance of the finalized object after you run the finalize method, thread synchronization operations, and any exceptions from the finalize method.

Resurrection is a side-effect of finalization. Sometimes we’ll be presented with a situation in which an object has been finalized but there is still a pointer to it, meaning that Garbage Collection assumes it’s alive when it’s been already finalized. A typical scenario is to finalize an object in order to create a new instance of the same object; if the first object is still there in finalization, the pointer points to the old object, and the object, while in finalized stage, never gets cleaned out properly because it’s got a reference from the application. It’s important that if you finalize something, you set a flag or a check routine to make sure that it’s gone before you try to do anything else concerning that object type.

Assigning Generations

Garbage Collection uses an ephemeral garbage collector, which describes the lifetime of an object in generations. Using this system, the garbage collector makes the following logical assumptions:

- Newer objects have shorter lifetimes.
- Older objects have longer lifetimes.
- Newer objects are created around the same time and have strong relationships.
- Compacting a portion of the heap is faster than compacting it completely.

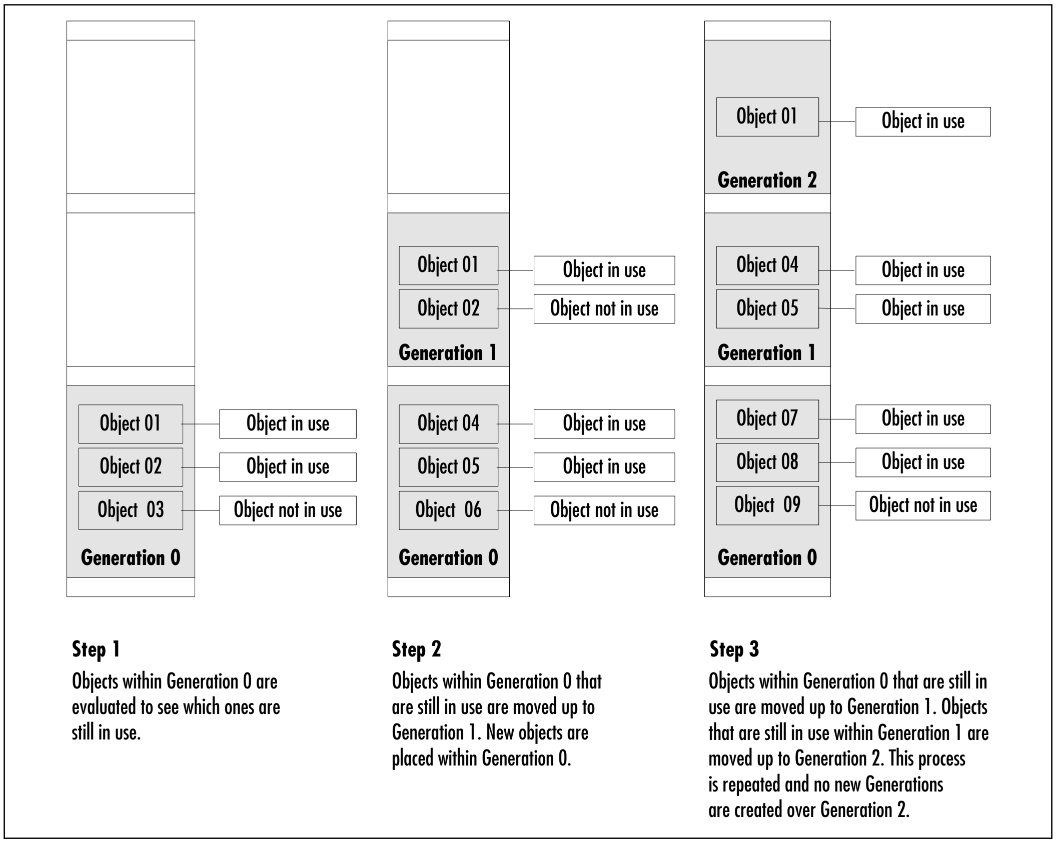
Let's look at a new heap. Once the heap is created and the first set of objects are instanced, they are created and set as *Generation 0*. As a new set of objects is created, Garbage Collection checks to see which objects from Generation 0 still exist (see Step 1 in Figure 2.7). Those that do exist are compacted, moved above Generation 0, and become *Generation 1* (see Step 2 in the figure). As the new Generation 0 enters the same process, so does Generation 1. Any remaining members of Generation 1 become *Generation 2*, and those that survived Generation 0 become 1 (see Step 3 in the figure). Then the new Generation 0 is created. At this point, the process continues, but there can be no higher generation than 2; any survivors from any subsequent Generation 1 members are placed in Generation 2 with the previous Generation 1 members that survived. This also means that a complete heap compacts portions at a time, thus increasing overall speed.

Objects within Generation 0 are checked more frequently than the other two generations due to .NET's philosophy that new objects are more likely to be the first to be removed. In other words, the longer an object is alive, the more likely it is to stay alive.

Utilizing Weak References

Another innovation that stems from the roots concept is *weak references*; a weak reference is a weak link to an object in memory that has been or is in the finalization process. It acts like a root will be collected by Garbage Collection the next time it runs. A *strong reference*, on the other hand, represents the primary object creation. Without a strong reference, you can't really create a weak one.

Figure 2.7 Generations


Object 01
Object in use
Object 02
Object in use
Object 03
Object not in use
Generation 0
Object 01
Object in use
Object 02
Object not in use
Generation 1
Object 04
Object in use
Object 05
Object in use
Object 06
Object not in use
Generation 0
Object 01
Object in use
Generation 2
Object 04
Object in use
Object 05
Object in use
Generation 1
Object 07
Object in use
Object 08
Object in use
Object 09
Object not in use
Generation 0
Step 1
Objects within Generation 0 are evaluated to see which ones are still in use.
Step 2
Objects within Generation 0 that are still in use are moved up to Generation 1. New objects are placed within Generation 0.
Step 3
Objects within Generation 0 that are still in use are moved up to Generation 1. Objects that are still in use within Generation 1 are moved up to Generation 2. This process is repeated and no new Generations are created over Generation 2.

Weak references can provide a workaround when you are dealing with memory-intensive objects and avoid the cost of constantly recreating and reinitializing objects. Imagine an object that traverses a database and stores a set of sorted fields. If the database is small enough, it can rest in memory without problem. However, if the database is large, we run the risk of over loading our resources every time we have to create a new one. Using a weak reference, we can bypass having to create a new object and redoing the sort by keeping the items we need on standby. You can then recreate the strong reference by pointing to the weak reference.

Security Services

Security services are not to be confused with the security concepts offered by .NET. Security services provide a type of check and balance within code, metadata, and MSIL. Security services ensure that the CLR gets what it expects, that it's getting it through either the same developer or a trusted source, and that future references to items usually denied access to due to isolation can be granted access.

In .NET, the Virtual Execution System (VES) handles all the security checking. Type safety is enforced through the VES by matching the same strong types in metadata with the corresponding MSIL (local variables and stack slots). You can look at it as a technical diagram; it draws a very strong line pointing from the metadata to the MSIL and makes sure that everything matches up to the correct declaration and memory space.

The VES also covers versioning safety. Since the VES lines everything up, it also goes ahead and verifies that all the information that's being checked also passes the version check. The VES also makes sure that the CLR will see what it gets—in other words, that the CLR will work within the assumptions it made about the code.

However, in order to make an assumption about the code, the CLR must be sure that the code is a proper executable. Again, the VES intervenes by providing the only three methods that a code can use to become executable: *class loader*, *legacy-code-based platform invoke*, and, for migration purposes, an *unmanaged COM interop*. Using the legacy-code platform invoke and the unmanaged COM interop can cause some performance issues, so it's best to avoid them altogether when writing or migrating code and to stick to the class loader. The class loader connects implementations to the information about the implementation within a metadata. The VES also uses the class loader to determine who is trying to access a type and thus takes the advantage to determine accessibility.

In addition, the VES has access through the CTS, to the permissions that are stored within metadata to access methods. It checks each type against the permissions and marks each type that has permission with a stub in the loader (the JIT and the linker also use VES to do the same) that tells the CLR to enforce the permissions to which the stub points. This is called *declarative security*.

NOTE

Even though the CLR is impressive in terms of detection algorithms, it has a drawback in that it's still simply a logical system. It can't tell when someone might trick it (although the CLR is very stringent, thus making it hard to trick). To prevent that, we can use *imperative security*; that is, we can set the rules in our code.

Framework Security

Code access security and *role-based security* are the two types of security provided by the.NET Framework itself. They are mechanisms that are geared toward a keep it simple mentality regarding how to decide what a user can do. The keep-it-simple idea is based on consistency, and providing easy transitions from code-based to role-based security and back. The fundamentals that give the .NET security its robustness are *permission*, *principals*, and *security policy*.

Code access security, as you might have noticed, provides varying degrees of trust for an application. It can change these degrees according to the information that the assembly provides, such as developer, version, and the like, since this information is stored in the code. When the process of determining if a particular code can access, the runtime checks the current call stack of the code looking for the permission, however if it can't find permission, it throws an exception.

Role-based security makes an authoritative decision based on the principal value from the current thread making the request. The role(s) listed within the principal value are then evaluated, and the action/ability requested is given or denied.

Financial software programmers and database coders might be already familiar with the concept of role-based security. Usually, in these situations, when a client requests access to a certain part of the system or resource, a check is run to determine from what role the client making the request comes. Let's say that a member of the group Alpha is trying to access a resource located with a member of the Omega group. Alpha starts the connection and Omega picks off the first principal from the connection thread. The principal is then analyzed for roles, and

Omega determines that the Alpha workgroup does not have permission for all the resources—just two of them. Omega allows the connection but limits Alpha's request to the two resources. If Alpha tried to obtain a resource outside those two, the request would be denied.

Granting Permissions

Permission is the basic building block of security. Some view permission logically as a response given to a query in order to gain access, while others look at it as a key fitting into a lock. Both views are equally correct. Permissions in .NET are used via requests, grants, and demands.

A code can *request* permissions to see if it can access a file. If it doesn't fall under those permissions, you could have a function *grant* permission to the code that's making the request. If a code with the permissions ready comes along, you might want to implement an added layer of permission called *demand*. In other words, while the code might have the basic permissions needed in order to satisfy the need, the code can also *demand* that (a) specific permission(s) be present. Both code access security and role-based security have a list of permissions (see Table 2.7).

Table 2.7 Code Access Security and Role-Based Security Permission Lists

Code Access Security Permissions	Description
DnsPermission	Provides access to a Domain Name System.
EnvironmentPermission	Provides access to the ability of read/write/query environment variables. Write access also includes the ability to create, remove, and write.
FileDialogPermission	Provides access to files acquired via a file dialog box.
FileIOPermission	Provides access to perform low-level (through stream) read, write, append, or create directories.
IsolatedStoragePermission	Provides access to an area that is attributed to a specific user within a part of the code identity.
ReflectionPermission	Used in conjunction with System.Reflection to have permission to find out information about a type at runtime.
RegistryPermission	Provides access to registry and the read, write, create, delete registry functions; applies to keys and values. If you truly want to make people

Continued

Table 2.7 Continued

Code Access Security Permissions	**Description**
	who use your .NET code happy, use the .NET and don't use the registry anymore. This permission is really more a migration step.
SecurityPermission	Provides the ability to do actions that are normally not allowed, such as calling into unmanaged code and skipping the verification process. Use this with caution; it can lead to holes in your system that can be used to access other parts of it.
SocketPermission	Doesn't really grant any ability; either accepts or creates any attempted connections at a given transport address. Using this permission in conjunction with SecurityPermission for executables can cause some bad things to happen.
UIPermission	Provides the ability to use the functionality provided by the user interface.
WebPermission	Just like SocketPermission, it either accepts or creates any attempted connections from/to a Web address.

Role-Based Security Permissions	**Description**
PrincipalPermission	Demands that the identity of an active principal match. (See the Principal section for more information.)

Gaining Representation through a Principal

Have you ever wanted a go-between to plead your case to the program to get access? A *principal* provides just that function. Depending on the situation, a principal provides the permission level needed on your behalf to enter. The CLR lets the principal in, but it's not letting *you* in, because the CLR only allows you to do what the principal is supposed to.

A generic principal is your run-of-the-mill representation that you can use to find out what someone that's not unauthenticated can see. Although this is not practical in an everyday program, it is very useful for testing and debugging situations and is extremely helpful when trying to determine situations in which a permission shows up that you didn't plan for.

Custom principals are created on the fly by an application to suit a current need or requirement. They extend the basic usability of a generic principal but are dependant on having the proper authentication modules and types given to them by the application. This dependency gives the custom principal an element of security since it can't work without being given what it needs to work.

NOTE

A special class of principal—the *Windows Principal*—represents strictly Windows users. It uses this impersonation to get roles that are available for that particular user.

Security Policy

The rules that the CLR follows are referred to collectively as the *security policy*. The local administrator determines these configurable rules. Once an assembly is attempting to load, the security policy is checked to see what permissions the CLR can grant the assembly. It determines various possibilities and then, if it passes, provides the needed permissions or simply does not allow the program to run.

Three levels specify security policy: the local machine policy, the application domain policy, and the user policy. The runtime uses all three of these policies to filter out the final security policy that will be placed on the assembly and thus determines its permissions. Both the user and the application domain policy specify the set of permissions that are allowed, and then this set of permissions is compared to the machine policy. The permissions that are not filtered out become the security policy.

Application Domains

An application in .NET runs in a domain that's managed by a host. This host can be a shell host (launches .EXEs from a shell), a browser host (runs code from the site), a server host (ASP.NET; runs code that handles requests on a server), and a custom-defined host. When one of these create the application domain, for example, the shell host—which would be Windows—sets the policy that the code must deal with under that domain. The policy generated cannot be added to but can be made more flexible by the host.

After an application domain policy is set, the new policy applies only to assemblies that are loaded after the creation of the new policy. Any previous policy holders will have their previous policy covered and won't have to use the new one unless reloaded. Once the main assembly is loaded and the first reference to another assembly is made, the loader kicks in, places the assembly into the appropriate application domain, and then returns the information (referred to as evidence) that proves it can be trusted (will return versioning information to verify) to the runtime. Table 2.8 displays the evidence that is/can be returned.

Table 2.8 Evidence

Application Directory	Where the Application Resides
Custom	An evidence created by the user or system defined; great for making 100 percent that sure it's the correct evidence.
Hash	Returns the hash encrypted in MD5 or SHA1.
Publisher	The AuthentiCode signature provided by the code.
Site	Location of origin.
Strong Name	Assembly's strong name.
URL	URL of origin.
Zone	Zone of origin—for instance, Internet Zone. Matches the zones listed in your Properties box for IE under the Security tab.

Summary

VB.NET is the first true version of VB released with a complete redesign after VB 4.0 came out. All the limitations that Visual Basic programmers have found in the past, such as being limited to windowed applications, are now completely gone. Visual Basic programmers can now take part in the console programming world and use the tricks associated with that world to create better programs and optimize batch files.

With the interoperability that .NET provides, programmers can use any language to overcome any of VB's language shortcomings. Any custom class written in any language, such as LISP, can be used and referenced by an assembly written in VB.NET and vice versa; C/C++ developers who would like to use some of VB's more robust functionality for windowed applications can now simply build the GUI out of VB and the implementation in C/C++ with no problems whatsoever.

The fuel for this new interoperability comes from .NET's CLR and MSIL. The CLR compiles any MSIL-generated code for our use without having to worry about what compiler was used to create it. The new deployment system, *assembly*, creates a standard way of looking at deployable files and removes our dependence on the registry and DLLs by including a road map of what it needs within the metadata. To top everything off, the burden of providing security is removed (somewhat) from the developer and placed in the hands of the CLR.

Solutions Fast Track

What Is the .NET Framework?

☑ .NET provides developers with new possibilities for creating applications.

☑ The CLR changes the way that programs are written, in the sense that VB developers won't be limited to the Windows platform.

Introduction to the Common Language Runtime

☑ The CLR is the heart of the .NET Framework. It provides a lot of the functionality that .NET uses.

☑ CLR will provide the function of translating the application from its internal code to code within the native environment.

☑ Managed code will be able to get the most of the new .NET features from the CLR.

Using .NET-Compliant Programming Languages

☑ Programming for .NET is not limited to the Microsoft standard languages. Any compiler that follows the Common Type System and other requirements for .NET can be created for any programming language.

☑ .NET's new interoperability allows us to use each language's strengths to counteract weak areas.

☑ Different programming languages will have the same method of communication within each other, ensuring true interoperability.

Creating Assemblies

☑ The new deployable unit for .NET is an assembly. It is more like a logical DLL file than a true executable file.

☑ All the information that the CLR needs to properly run an assembly is located within the assembly itself.

☑ Each assembly file consists of the internal code, the manifest area, and the metadata contained within the manifest area.

Understanding Metadata

☑ Metadata contains the map that .NET uses to layout objects in memory and how they are used.

☑ The manifest area within the assembly contains the metadata.

Using System Services

☑ More control is given to exception handling through the try/catch system.

☑ The automatic resource management system for .NET is smart enough to know when objects are in use and when they need to be removed.

This takes the burden off the programmer, but the programmer can always opt to declare when an object should be removed.

☑ Console applications are now within the reach of VB programmers through the intrinsic *System.console* namespace.

Microsoft Intermediate Language

☑ MSIL is the bytecode that the just-in-time (JIT) compiler utilizes to create native code for the assembly file.

☑ MSIL is platform-independent.

☑ The code within a .NET application is converted to MSIL.

Using the Namespace System to Organize Classes

☑ A namespace provides an organizational hierarchical system for classes.

☑ Each class that specifies to a specific function is stored within its respective namespace.

☑ The System namespace is the root namespace of all namespaces in .NET.

The Common Type System

☑ The Common Type System is the way that types are supported within the runtime.

☑ The CTS also specifies how types can interact with each other and how they are displayed as metadata.

☑ The CTS provides the rules that types must follow in order to work with .NET.

Relying on Automatic Resource Management

☑ The managed heap system replaces the reference count system.

☑ The object cleanup is referred to as Garbage Collection. .NET controls when Garbage Collection runs and when an object is removed.

☑ The burden of object cleanup is placed more within .NET than on the developer.

Security Services

☑ Permissions are the rights needed to use a resources. There are many different types of permissions that can be used in any event and are primarily used within code access security.

☑ The principal acts as a go-between for you to get the permissions needed. There is only one type of principal. Principals are used within role-based security.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: I've heard that there has been a significant change in VB.NET since Beta 1. Will this affect the Framework?

A: No. The changes being done to VB.NET are actually changes to allow backward compatibility with VB 6.0 semantics. They do not truly affect the portability or the CLR.

Q: What are the changes to VB.NET in Beta 2?

A: The changes are as follows:

- VB.NET will default TRUE values to –1 again instead of 1. Just as in VB, VB.NET also applies to explicit (using Cint() and so on) and implicit conversions (giving an integer the value of the Boolean).
- And/Or/Not/Xor will return to being bitwise operators instead of pure logic. This removes BitAnd, BitOr, and BitNot operators from VB.NET.

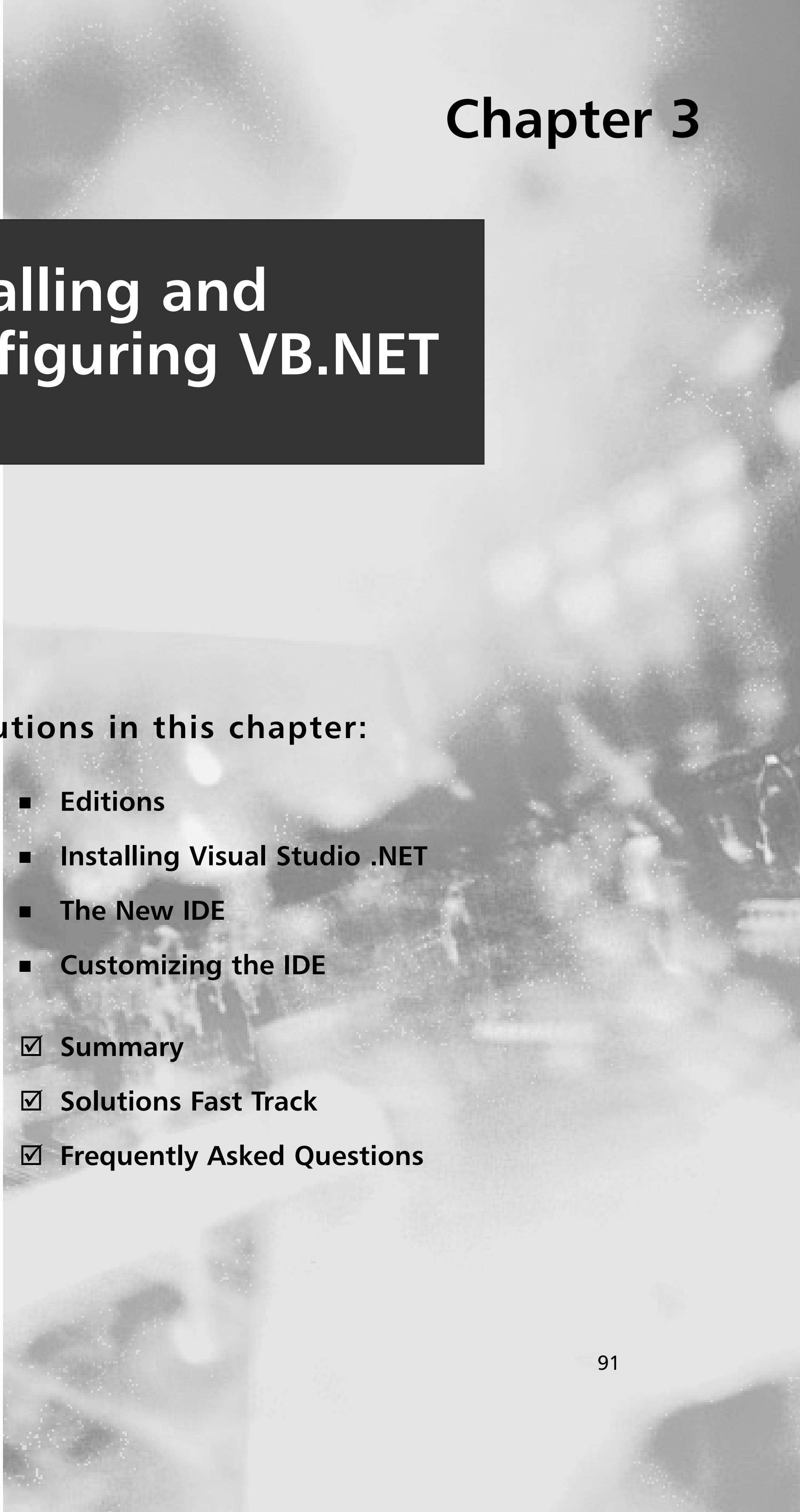
- And/Or/Not/Xor will be returned to VB 6.0 order of operator precedence.
- AndAlso and OrElse will be introduced to create the "short-circuit" behavior once used by And/Or/Not/Xor.
- Arrays will now be declared using an upper bound, as in VB 6.0.

Q: Do I have to use Visual Studio.net or a Microsoft-endorsed editor to create my VB.NET files?

A: No. With the implementation of VBC.EXE, you can use any editor you want to write the code, without suffering any bugs or problems.

Q: Is it better to learn and rewrite my existing VB 6.0 applications in VB.NET or to make the necessary changes to my VB 6.0 application to run on VB.NET?

A: That's a subject of debate. It all depends on the size of your code. Naturally, smaller programs will be easier to convert to VB.NET; even if you do convert to .NET, you might still miss out on the advantages VB.NET has over VB 6.0. On the other hand, learning and rewriting a complete program in VB.NET can be time consuming. Keep these considerations in mind when deciding what you should convert and what you should rewrite.



Chapter 3

Installing and Configuring VB.NET

Solutions in this chapter:

Introduction

Prior to beginning Visual Basic .NET installation, you should make some preliminary checks first. You must verify that you meet the system requirements for installation. When you install Visual Studio .NET, it will also install the MSDN for Visual Studio .NET, which contains valuable information on .NET development. You can also install sample projects that help you learn .NET. If you aren't sure whether you need a component during installation, you can always add components later.

The Integrated Development Environment (IDE) has some changes, but it should be familiar to those of you who have used Visual Basic 6.0 and Visual Studio. All projects, regardless of the programming language, will be developed in the same IDE now. When you start Visual Studio .NET, you no longer choose between tools such as Visual Basic or Visual C++; you just start Visual Studio. To keep in line with the new Internet strategy, Visual Studio starts with a home page. It contains links for various items, and you can customize it to your liking. You will see some new project options available. If you have used Visual Interdev 6.0, you are already familiar with the task list that is now available. The tabbed child windows feature makes navigation between windows easier. The new IDE makes development much easier, as we will see.

A new feature of the IDE is that it can be customized to your liking. You can customize the home page for the links you prefer, create a profile that will contain some preset defaults for different types of programmers, and choose from several windows layouts and keyboard schemes. In this chapter, you will learn how to install Visual Studio .NET, explore the new features of the IDE, and learn how to customize the IDE to fit your needs.

Editions

Currently the Beta 2 version of Visual Studio .NET includes only the components that will be found in Visual Studio .NET Professional Edition. Microsoft plans to release at least two other editions, named Visual Studio .NET Enterprise Architect and Visual Studio .NET Enterprise Developer. Visual Studio .NET Enterprise Developer will include a host of tools to assist developers with the process of building custom applications to use on the .NET platform including modeling features, core reference applications, and testing capabilities. Visual Studio .NET Enterprise Architect will include tools to simplify the job for architects of XML-based Web services. Neither of the Enterprise editions are widely available as of the printing of this book.

Installing Visual Studio .NET

You can install Visual Studio .NET on Windows 2000 and Windows NT 4.0. You can execute code in Windows 98 and higher. Be aware that this product is still under development; installing it on a production or development machine is not advisable. There is also no guarantee that the applications built using Visual Studio .NET Beta 2 will work the same way in the released version. Also, it is not advisable to create and deploy production applications using Visual Studio .NET Beta 2. Visual Studio .NET Beta 2 is designed for evaluation and academic purposes and fit for installation only on test machines.

Visual Studio .NET Beta 2 should successfully install and interoperate with existing Microsoft products including Visual Studio 6.0 and Visual Interdev. However, certain issues might arise, including security issues. Make sure that you read the release notes in Readme.htm, located in the root of Visual Studio CD1. You can look for the latest information in the Beta 2 Web site at http://beta.visualstudio.net.

Visual Studio .NET Beta 2 requires that a specified number of Windows components be present on the machine before it is installed. The first step in the installation process is to install the following Windows components:

- Windows 2000 Service Pack 2
- Microsoft Windows Installer 2.0
- Microsoft FrontPage 2000 Web extensions client
- Setup runtime files
- Microsoft Internet Explorer 6.0 and Internet tools
- Microsoft Data Access Components 2.7
- Microsoft .NET Framework

Some of these system components, such as the .NET Framework, are still in beta stages. Visual Studio .NET requires that the user be an Administrator on the local machine. Given that the user is required to log on as an Administrator, potential security issues may arise that could be exploited maliciously. Because this is a beta version of the product, the installation might not complete successfully (or be aborted midway), and in these situations the password could remain in the registry. If this happens, the administrator password becomes easily accessible. The minimum hardware requirements for installing Visual Studio are listed in Table 3.1.

Table 3.1 Minimum Hardware Requirements for Installing Visual Studio

Hardware Type	Minimum Requirement	Recommended
Processor	Pentium 2 processor with a speed of 450 MHz	Pentium 3 processor with a speed of 600 MHz
Memory	128MB	256MB
Hard Disk Space	3GB	3GB
Video Settings	800 x 600, 256 colors	High Color 16-bit
CD-ROM	Required	Required

Exercise 3.1: Installing Visual Studio .NET

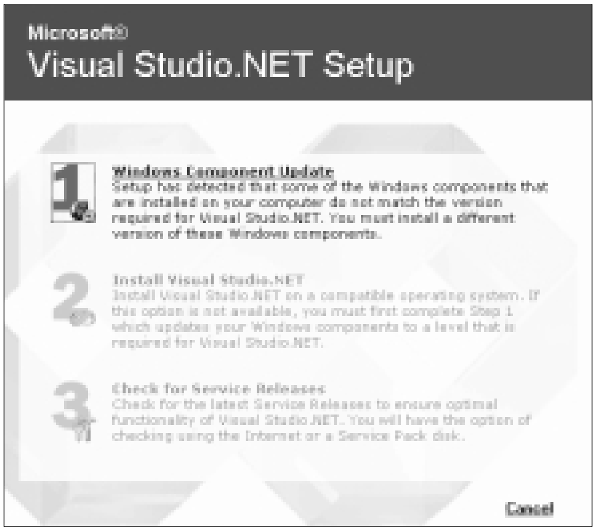
The three phases for installing Visual Studio .NET are as follows:

- Phase 1 involves installing Windows components.
- Phase 2 involves installing Visual Studio .NET.
- Phase 3 involves checking for service releases.

Installing Visual Studio is not a difficult task. In this exercise, we walk through the steps necessary for installation:

1. To start the installation, insert the **Visual Studio .NET CD-ROM**. If installation does not start automatically, double-click **setup.exe** to start the installation. Setup launches the initial screen shown in Figure 3.1.
2. Click **Windows Component Update** to bring up the End User License Agreement screen, shown in Figure 3.2.
3. Click the **I accept the agreement** button to accept the user agreement, and the screen shown in Figure 3.3 appears. This screen lists the required Windows components for running Visual Studio .NET.

Figure 3.1 Installing Windows Components



Microsoft®
Visual Studio.NET Setup

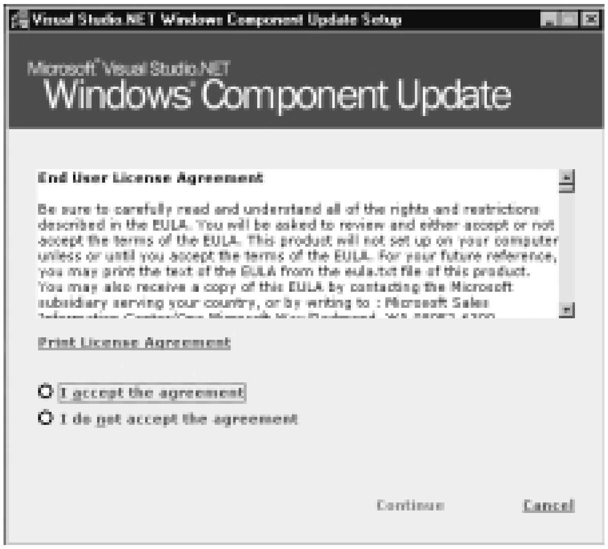
1 **Windows Component Update**
Setup has detected that some of the Windows components that are installed on your computer do not match the version required for Visual Studio.NET. You must install a different version of these Windows components.

2 **Install Visual Studio.NET**
Install Visual Studio.NET on a compatible operating system. If this option is not available, you must first complete Step 1 which updates your Windows components to a level that is required for Visual Studio.NET.

3 **Check for Service Releases**
Check for the latest Service Releases to ensure optimal functionality of Visual Studio.NET. You will have the option of checking using the Internet or a Service Pack disk.

Cancel

Figure 3.2 End User License Agreement



Visual Studio.NET Windows Component Update Setup

Microsoft® Visual Studio.NET
Windows® Component Update

End User License Agreement

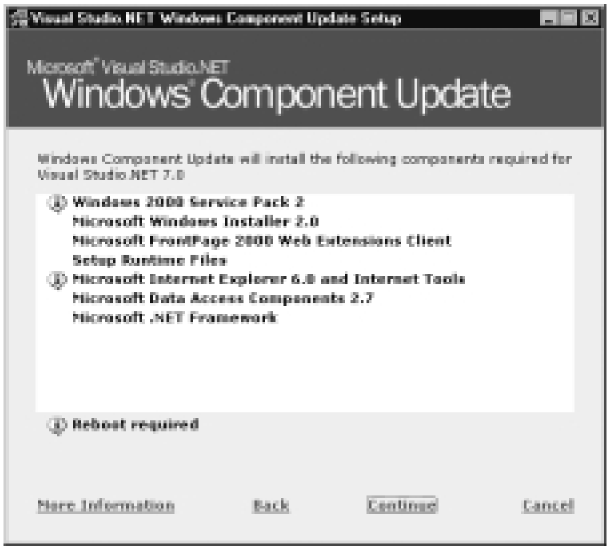
Be sure to carefully read and understand all of the rights and restrictions described in the EULA. You will be asked to review and either accept or not accept the terms of the EULA. This product will not set up on your computer unless or until you accept the terms of the EULA. For your future reference, you may print the text of the EULA from the eula.txt file of this product. You may also receive a copy of this EULA by contacting the Microsoft subsidiary serving your country, or by writing to : Microsoft Sales

Print License Agreement

○ I accept the agreement
○ I do not accept the agreement

Continue Cancel

Figure 3.3 Windows Components


Visual Studio.NET Windows Component Update Setup
Microsoft Visual Studio.NET
Windows Component Update

Windows Component Update will install the following components required for Visual Studio.NET 7.0

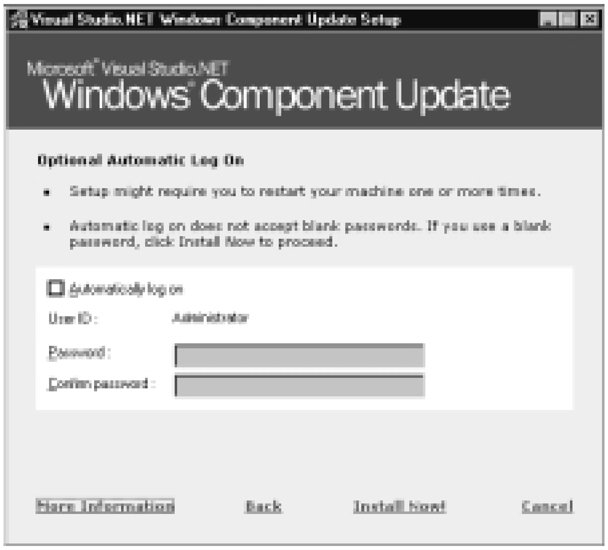
Windows 2000 Service Pack 2
Microsoft Windows Installer 2.0
Microsoft FrontPage 2000 Web Extensions Client
Setup Runtime Files
Microsoft Internet Explorer 6.0 and Internet Tools
Microsoft Data Access Components 2.7
Microsoft .NET Framework

Reboot required

More Information
Back
Continue
Cancel

4. Click **Continue**, and the screen shown in Figure 3.4 appears. Installing windows components requires rebooting the machine several times. Setup gives you an option to enter your password to do an unattended install. Setup uses the password to automatically log the user in after every reboot. Checking the **Automatically log on** check box enables the two text boxes. Type the password in the first text box. Retype the password for confirmation in the **Confirm Password** textbox.
5. After you specify the password, click **Install Now!** to begin the installation of Windows components. The setup program installs the components shown in Figure 3.3 and automatically reboots the system when necessary. This marks the end of the first phase of installation. Figure 3.5 shows the screen that appears after all the necessary Windows have been successfully installed.
6. The next step is to start installing Visual Studio .NET, which constitutes the second phase of the entire installation procedure. After you click the **Done** hyperlink, setup shows you the same screen you saw in Figure 3.1, but this time the second link is enabled, and the first and third hyperlinks are disabled. Figure 3.6 shows you the beginning of the second phase of installation.

Figure 3.4 Automatic Logon



Visual Studio.NET Windows Component Update Setup

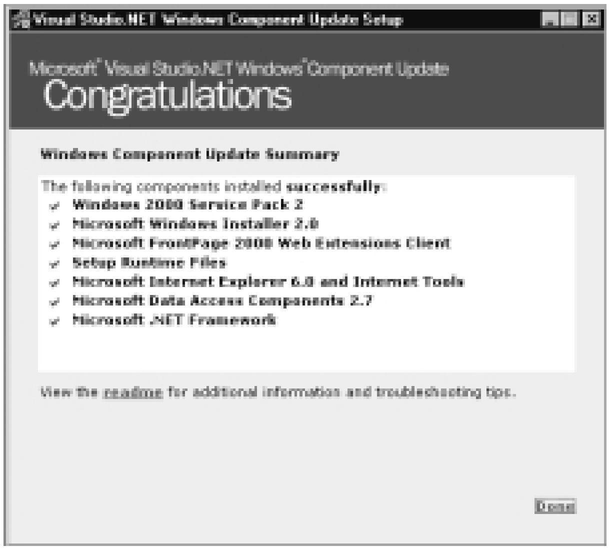
Microsoft® Visual Studio.NET
Windows® Component Update

Optional Automatic Log On

- Setup might require you to restart your machine one or more times.
- Automatic log on does not accept blank passwords. If you use a blank password, click Install Now to proceed.

☐ Automatically log on
User ID: Administrator
Password:
Confirm password:

Figure 3.5 Windows Component Update Summary



Visual Studio.NET Windows Component Update Setup

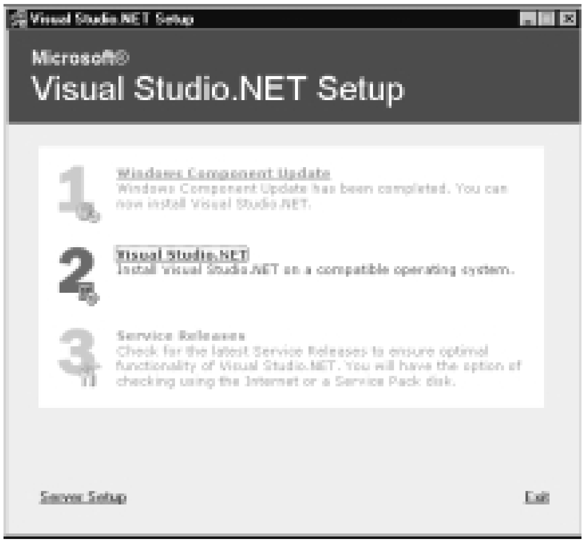
Microsoft® Visual Studio.NET Windows® Component Update
Congratulations

Windows Component Update Summary

The following components installed **successfully**:
- ✓ Windows 2000 Service Pack 2
- ✓ Microsoft Windows Installer 2.0
- ✓ Microsoft FrontPage 2000 Web Extensions Client
- ✓ Setup Runtime Files
- ✓ Microsoft Internet Explorer 6.0 and Internet Tools
- ✓ Microsoft Data Access Components 2.7
- ✓ Microsoft .NET Framework

View the readme for additional information and troubleshooting tips.

Figure 3.6 Second Phase of Installation



Visual Studio.NET Setup

Microsoft®
Visual Studio.NET Setup

1 Windows Component Update
Windows Component Update has been completed. You can now install Visual Studio.NET.

2 Visual Studio.NET
Install Visual Studio.NET on a compatible operating system.

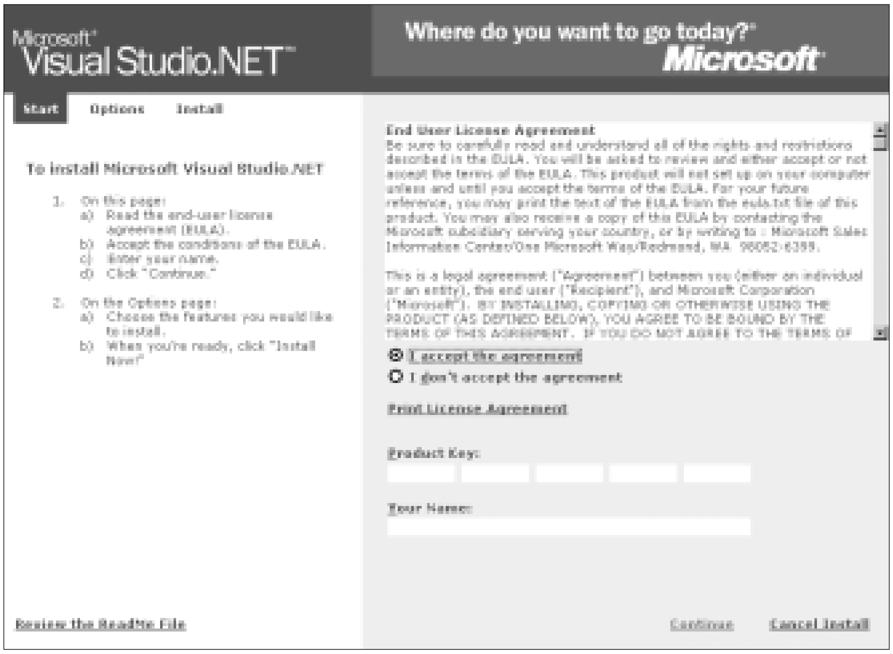
3 Service Releases
Check for the latest Service Releases to ensure optimal functionality of Visual Studio.NET. You will have the option of checking using the Internet or a Service Pack disk.

Server Setup

Exit

7. Click **Visual Studio .NET**, and the setup program copies the files necessary for installation and displays the screen shown in Figure 3.7.

Figure 3.7 Beginning Visual Studio .NET Setup



To install Microsoft Visual Studio.NET

1. On this page:
   a) Read the end-user license agreement (EULA).
   b) Accept the conditions of the EULA.
   c) Enter your name.
   d) Click "Continue."
2. On the Options page:
   a) Choose the features you would like to install.
   b) When you're ready, click "Install Now!"

Review the ReadMe File

End User License Agreement
Be sure to carefully read and understand all of the rights and restrictions described in the EULA. You will be asked to review and either accept or not accept the terms of the EULA. This product will not set up on your computer unless and until you accept the terms of the EULA. For your future reference, you may print the text of the EULA from the eula.txt file of this product. You may also receive a copy of this EULA by contacting the Microsoft subsidiary serving your country, or by writing to : Microsoft Sales Information Center/One Microsoft Way/Redmond, WA 98052-6399.

This is a legal agreement ("Agreement") between you (either an individual or an entity), the end user ("Recipient"), and Microsoft Corporation ("Microsoft"). BY INSTALLING, COPYING OR OTHERWISE USING THE PRODUCT (AS DEFINED BELOW), YOU AGREE TO BE BOUND BY THE TERMS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THE TERMS OF

[✓ I accept the agreement] I don't accept the agreement

Print License Agreement

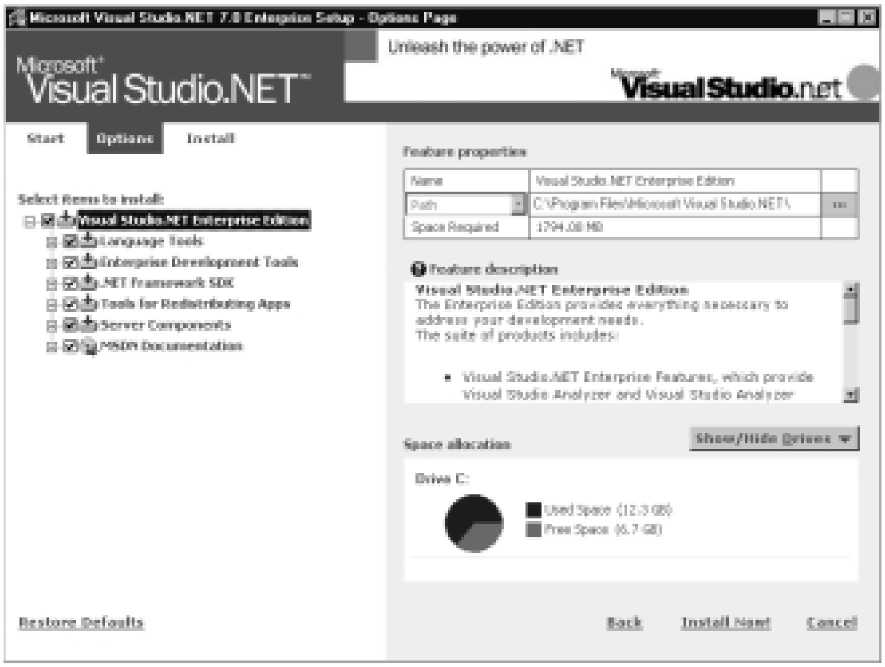
Product Key:

Your Name:

Continue Cancel Install

8. After entering the product key and your name, click the **I accept the agreement** button. Click **Continue** to continue to the next part of the current phase, which is selecting the features you want to install. Figure 3.8 shows the available selections.

Figure 3.8 Selecting VS.NET Features



Select items to install:

- ☑ Visual Studio.NET Enterprise Edition
  - ☑ Language Tools
  - ☑ Enterprise Development Tools
  - ☑ .NET Framework SDK
  - ☑ Tools for Redistributing Apps
  - ☑ Server Components
  - ☑ MSDN Documentation

Name	Visual Studio.NET Enterprise Edition
Path	C:\Program Files\Microsoft Visual Studio.NET\
Space Required	1794.08 MB

Visual Studio.NET Enterprise Edition
The Enterprise Edition provides everything necessary to address your development needs.
The suite of products includes:

- Visual Studio.NET Enterprise Features, which provide Visual Studio Analyzer and Visual Studio Analyzer

Drive C:
■ Used Space (12.3 GB)
■ Free Space (6.7 GB)

9. After you select the features to install, click **Install Now!** to start the installation. The last phase of the installation, which is checking for service releases, kicks in after the Visual Studio .NET installation is complete. This involves checking for any service packs. Because this is a beta release, this option is of little significance.

Installing on Windows 2000

Internet Information Server (IIS) and FrontPage Server Extensions must be present on the Windows 2000 machine before you can install Visual Studio .NET Beta 2. IIS is installed by default on Windows 2000 Server and Advanced Server but not on Windows 2000 Professional. So make sure that IIS is configured before you install Visual Studio .NET on a machine running Windows 2000 Professional.

FrontPage Server Extensions are configured on a Windows 2000 machine only if the operating system is installed on the NTFS file system. You must install FrontPage Server extensions if the Windows 2000 operating system is installed on a FAT16 or FAT32 file system. After making sure that the required components are installed, insert the Visual Studio .NET Beta 2 CD to begin the installation.

The New IDE

Visual Studio .NET, like Visual Studio 6.0, lends itself to automation by exposing a very rich programming model. The new programming model supported by Visual Studio .NET goes beyond the extensibility model supported in Visual Studio 6.0, which has two extensibility models. One was used to automate the Visual Basic 6.0 environment, and the other was to automate the Visual C++ environment. Microsoft Visual Basic 6.0 extensibility model allowed the developer to automate mainly the project environment. The Visual C++ environment allowed the developer to exploit only the document and text editor.

Visual Studio .NET not only brought together all the development environments but also added a host of objects to the extensibility model. It provides direct access to developers and tool writers to the underlying components and events that drive the IDE. The developer can customize the look and feel of the IDE, enhance its functionality, and integrate the IDE with other Microsoft applications.

You can customize the Visual Studio .NET IDE in two ways: with built-in customizations and user-defined customizations. Built-in customization takes the form of the customizable toolbox, customizable toolbar, and so on. User-defined customizations take the form of known features such as add-ins, wizards, macros, and so on. These features are some of the programmable components of the IDE. The following sections cover these components in detail.

Integrated Development Environment Automation Model

The automation capabilities of Visual Studio .NET give the developer absolute control of the IDE. The developer can customize the IDE to his specific needs, automate repetitive tasks, and virtually control the way the IDE works. To enable this flexibility, the new IDE programming model consists of numerous objects. These objects provide direct access to various windows such as the command window, output window, and tasklist window, as well as the code editor and the tasklist events. The various objects are grouped under the following categories depending on their functionality:

- **Add-in objects** Add-ins are program modules that are created to perform repetitive tasks within the IDE. Add-ins are discussed in the following sections.
- **Project collection objects** The Project collection objects store details of a project that is created in the IDE. The project collection objects can contain Visual Basic projects, C# projects, or Visual C++ projects.
- **Commands objects** A command object represents a command in the Visual Studio environment.
- **Build objects** The Build objects allow a programmer to control the build environment of Visual Studio .NET.
- **Events objects** The Events object is responsible for providing access to all events that are raised within the IDE. Thus, the programmer can use this object for performing custom processing based on the occurrence of an action.
- **Debugger objects** The Debugger object is used to manipulate the debugger, such as setting the next breakpoint, querying the breakpoints hit, the status of the current program being debugged, and so on programmatically.
- **Properties objects** The Property object is a single instance in a collection of Property objects.
- **Window configuration objects** The Window configuration object holds information on the layout and the way in which windows within the IDE are configured.
- **Code objects** The Code objects are essentially a collection of objects that allows a programmer to manipulate the contents in the code editor.

Each of these high-level objects consists of a set of objects, collections, and interfaces, each catering to a specific functionality. The top level Events object contains the following objects:

- **BuildEvents** The BuildEvents object provides events that are fired when a solution is built.

- **CommandBarEvents** The CommandBarEvents object causes a click event to occur when you click on a control in the command bar.
- **CommandEvents** The CommandEvents object provides command events for automation clients.
- **DocumentEvents** The DocumentEvents objects provides events that fire whenever an action is performed on a document. The events that are fired are DocumentClosing event, DocumentOpened event, DocumentOpening event, and DocumentSaved event.
- **Development Tool Environment (DTE) Events** The DTEEvents object provides events that are fired depending on the changes happening to the environment. The events that are fired are ModeChanged event, OnBeginShutdown event, OnMacrosRuntimeReset event, and OnStartupComplete event.
- **FindEvents** The FindEvents object fires a single event that occurs when you do a Find operation on files. It fires the FindDone event.
- **OutputWindowEvents** This object fires three events whenever any change happens to the output window. The events are PaneAdded event, PaneClearing event, and PaneUpdated event.
- **SelectionEvents** Whenever you make changes to a selection, a single event in the SelectionEvents object is fired. The event name is OnChange event.
- **SolutionEvents** The SolutionEvents object fires eight different events when changes are made to a solution. The events are AfterClosing event, BeforeClosing event, Opened event, ProjectAdded event, ProjectRemoved event, ProjectRenamed event, QueryCloseSolution event, and the Renamed event.
- **TaskListEvents** The TaskList events object provides events that respond to changes made to the TaskList. The events are TaskAdded event, TaskModified event, TaskNavigated event, and TaskRemoved event.
- **WindowEvents** The WindowEvents object provides events that are fired when changes are made to the windows in the environment. The events are WindowActivated event, WindowClosing event, WindowCreated event, and WindowMoved event.

- **VBProjectEvents**, **CsharpProjectEvents**, and **VCProjectEvents** These are late-bound properties of the Events object. They are available when a project is opened in Visual Studio .NET.

For example, you can use the BuildEvents object to do processing whenever a build process begins or whenever a build process ends. The SolutionEvents object provides the AfterClosing, BeforeClosing, Opened, ProjectAdded, ProjectRemoved, ProjectRenamed, QueryCloseSolution and the Renamed events. These events provide a flexible way for programmers to customize the Visual Studio .NET IDE to suit their requirements.

The integration of Visual Basic into the IDE means that both Visual Basic and Visual C++ can now use the same extensibility model. This is unlike the previous versions of Visual Studio where Visual Basic 6.0 had its own extensibility model that neither had as many as objects nor fired as many events.

Debugging…

Exception Handling

Visual Basic .NET introduces a new type of exception handling called *structured exception handling,* besides supporting unstructured exception handling. Unstructured exception handling is implemented with the help of **On Error Goto**, and the new structured exception handling involves the use of **Try**, **Catch**, and **Finally** statements.

Structured exception handling provides a more powerful and a comprehensive way to handle errors. It uses a predefined construct that allows you to code, filter errors, and perform cleanup operations. The **Try** block contains code that can potentially raise errors, the **Catch** block has code that will trap the exceptions, and the **Finally** block is the final step in setting up an exception handler. If an error occurs in the **Try** block during execution of code, Visual Basic .NET evaluates each of the **Catch** statements to match the exception that was generated. If a match is found, the control is transferred to the first line of the **Catch** statement that matches this exception. If no **Catch** statement is found, the control is transferred to the outer **Try…Catch…Finally** block, if one was present. If no external block was found, then the control is transferred to the calling procedure, and a matching **Catch** statement is searched for. If that is also not present, then a message box containing the error is

Continued

displayed. Alternately, you can handle errors more gracefully by specifying a **Catch** statement without any reference to an exception. In this case, this **Catch** statement becomes a generic error handler and will be called for all unhandled exceptions. The **Finally** statement is the last statement to be executed in a structured exception-handling scenario. This block normally contains code that releases connections to databases, closing files, and so on.

Add-Ins

The easiest way for developers to customize the development environment is to use DTE extensions called add-ins. The term DTE extension refers to a collection of tools—such as add-ins, wizards, and so on—that extend the power of the development environment. Add-in is the generic term for a program that is created to perform tasks within the IDE, often in response to events. An add-in is typically used to automate repetitive tasks and extend the functionality of the development environment.

An add-in is a compiled application that is loaded and used by the IDE. They can be invoked through the Add-in Manager, command window, during IDE startup, or during the Visual Studio .NET startup from a command line. An add-in is represented as a COM object or a .NET assembly that implements the IDTExtensibility2 interface. The IDTExtensibility2 is an interface object that provides five methods acting as events in a Visual Studio .NET environment. They are fired when add-ins are loaded and unloaded in an environment, when an environment is shut down, and so on. The five methods are as follows:

- **OnAddInsUpdate Method** This event is fired when an add-in is loaded or unloaded in an environment.
- **OnBeginShutdown Method** This event is fired when the environment is shut down.
- **OnConnection Method** This event is fired when an add-in is loaded in the environment.
- **OnDisconnection Method** This event is fired when the add-in is unloaded from the environment.
- **OnStartupComplete Method** This event is fired when the environment is ready.

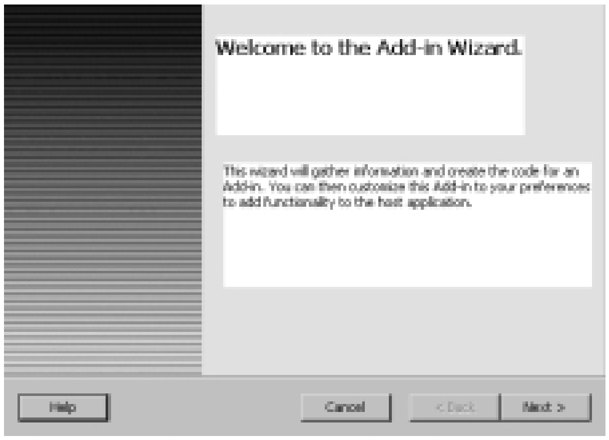
You can create a Visual Studio .NET add-in by using Visual Basic, Visual C++, or C#. Add-ins created using Visual Studio .NET can be hosted in a variety of Microsoft applications.

Visual Studio .NET provides an Add-in Wizard that helps you create an add-in template. Once created, the add-in appears in the Add-in Manager. The Add-in can then be configured to load during startup and/or when invoked from the command line. Exercise 3.2 lists the various steps involved in creating the add-in using the Add-in Wizard.

Exercise 3.2 Creating an Add-In Using the Add-In Wizard

1. The Add-in Wizard is invoked when you choose the Visual Studio .NET add-in template. You can find the template when you choose **New Project** from the **File** menu and choose **Extensibility projects** project type. After entering a name for the add-in and clicking **OK**, the Add-in Wizard starts—Figure 3.9 shows its initial screen. The wizard collects information from the user and creates the basic code for an add-in.

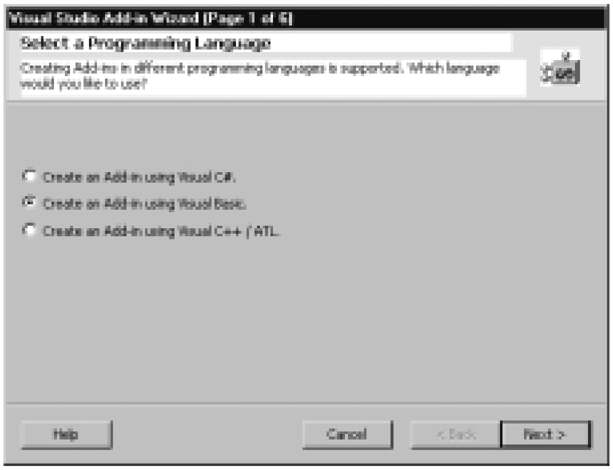
Figure 3.9 The Initial Add-In Wizard Screen


Welcome to the Add-in Wizard.

This wizard will gather information and create the code for an Add-in. You can then customize this Add-in to your preferences to add functionality to the host application.

2. You can create add-ins in any of the languages supported by Visual Studio .NET: Visual Basic, C#, and Visual C++. Figure 3.10 prompts the user to choose the programming language with which the add-in will be created.

Figure 3.10 Choosing the Programming Language


Visual Studio Add-in Wizard (Page 1 of 6)

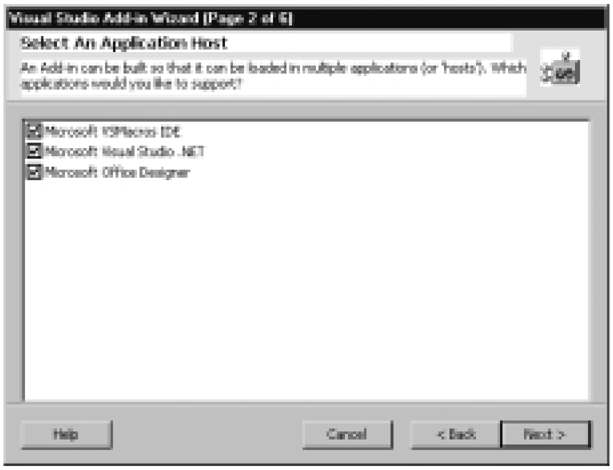
Select a Programming Language

Creating Add-ins in different programming languages is supported. Which language would you like to use?

( ) Create an Add-in using Visual C#.
[✓ Create an Add-in using Visual Basic.]
( ) Create an Add-in using Visual C++ / ATL.

3. Figure 3.11 lists the various hosts in which you can load the add-in. An application host is an application that supports the execution of an add-in. So, in this example, all of the applications listed here can execute the add-in.

Figure 3.11 Selecting an Application Host


Visual Studio Add-in Wizard (Page 2 of 6)

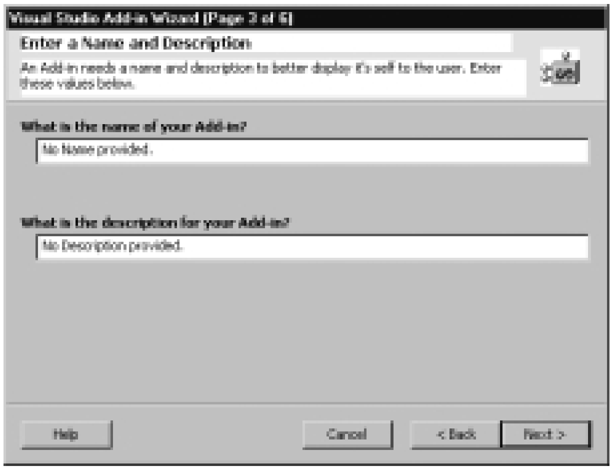
Select An Application Host

An Add-in can be built so that it can be loaded in multiple applications (or 'hosts'). Which applications would you like to support?

☑ Microsoft VSMacros IDE
☑ Microsoft Visual Studio .NET
☑ Microsoft Office Designer

4. Figure 3.12 prompts the user to enter a name and a description for the add-in. The name that you enter here appears when you click the Add-in Manager submenu from the Tools menu. You can also type in a short description.

Figure 3.12 Entering a Name and Description



Visual Studio Add-in Wizard [Page 3 of 6]

Enter a Name and Description

An Add-in needs a name and description to better display it's self to the user. Enter these values below.

What is the name of your Add-in?

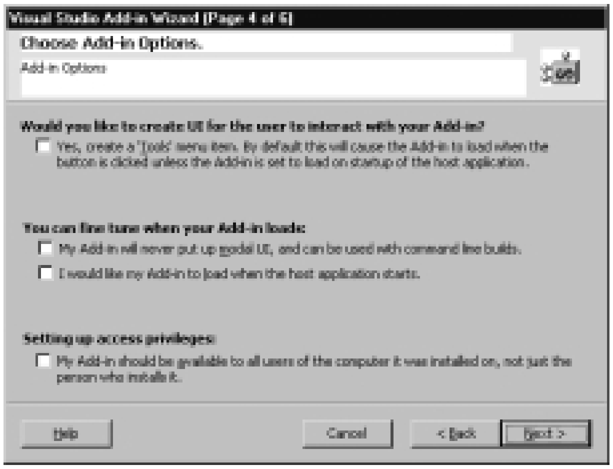
No Name provided.

What is the description for your Add-in?

No Description provided.

5. Figure 3.13 displays the configurable options for the add-in. You can specify whether you want the wizard to automatically create an entry in the Tools menu for this add-in; specify that this add-in does not have a user-interface and can be invoked from the command-line; specify that the add-in must be loaded whenever the IDE starts; and finally, define the accessibility of the add-in. You can use the last option to specify whether the add-in is available to all users or only to the user who installs it.

Figure 3.13 Configurable Options



Visual Studio Add-in Wizard [Page 4 of 6]

Choose Add-in Options.

Add-in Options

Would you like to create UI for the user to interact with your Add-in?

☐ Yes, create a 'Tools' menu item. By default this will cause the Add-in to load when the button is clicked unless the Add-in is set to load on startup of the host application.

You can fine tune when your Add-in loads:

☐ My Add-in will never put up modal UI, and can be used with command line builds.

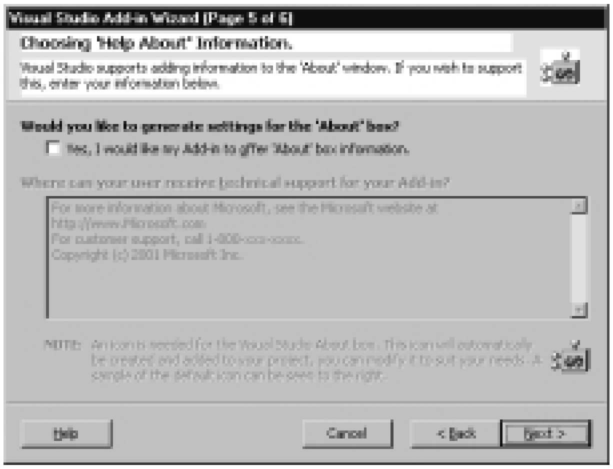
☐ I would like my Add-in to load when the host application starts.

Setting up access privileges:

☐ My Add-in should be available to all users of the computer it was installed on, not just the person who installs it.

6. Figure 3.14 shows the screen that allows you to configure the **About** option for your add-in. You can use this option to display the name of the application, the version number, and the author of the add-in.

Figure 3.14 Configuring the About Option



Visual Studio Add-in Wizard (Page 5 of 6)

Choosing 'Help About' Information.

Visual Studio supports adding information to the 'About' window. If you wish to support this, enter your information below.

Would you like to generate settings for the 'About' box?

☐ Yes, I would like my Add-in to offer 'About' box information.

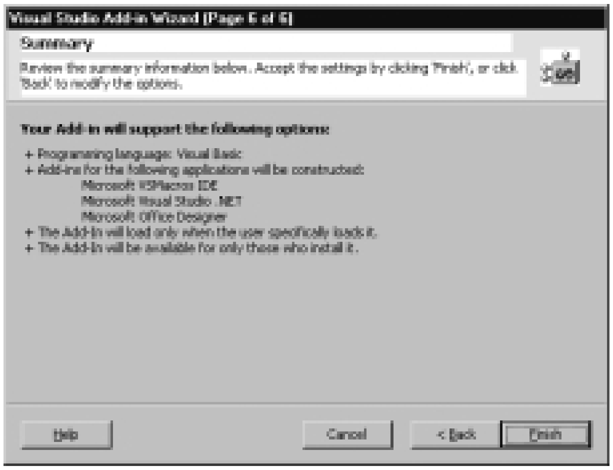
Where can your user receive technical support for your Add-in?

```
For more information about Microsoft, see the Microsoft website at
http://www.Microsoft.com.
For customer support, call 1-800-xxx-xxxx.
Copyright (c) 2001 Microsoft Inc.
```

NOTE: An icon is needed for the Visual Studio About box. This icon will automatically be created and added to your project, you can modify it to suit your needs. A sample of the default icon can be seen to the right.

7. Figure 3.15 shows you the last screen of the wizard. This screen merely summarizes all the options that you have configured. Click **Finish** to complete the wizard.

Figure 3.15 Summary



Visual Studio Add-in Wizard (Page 6 of 6)

Summary

Review the summary information below. Accept the settings by clicking 'Finish', or click 'Back' to modify the options.

Your Add-in will support the following options:

```
+ Programming language: Visual Basic
+ Add-ins for the following applications will be constructed:
        Microsoft VSMacros IDE
        Microsoft Visual Studio .NET
        Microsoft Office Designer
+ The Add-In will load only when the user specifically loads it.
+ The Add-In will be available for only those who install it.
```

Wizards

As shown in the preceding section, a wizard is a user assistance tool that helps to accomplish a task that is either complex or requires experience. The wizard typically consists of a series of dialog boxes that elicit information from the user in an organized manner. After the wizard collects all the necessary information, it goes about completing the task by implementing a method or methods using the information the user provides. Before you can implement a wizard, you need to add a reference to EnvDTE assembly. The EnvDTE assembly implements a lot of interfaces, one of which is the IDTWizard interface. The IDTWizard interface has only one method, called the *Execute* method. When you create a wizard by implementing the *Execute* method of the IDTWizard interface, the necessary code to complete the task is written as part of the *Execute* method. The Execute method takes in four parameters:

- A pointer to the DTE object.
- A handle to the wizard's parent window.
- An array of parameters that allow you to specify options such as WizardType, the directory where the solution files will be stored, the directory where the solution will be installed, and so on.
- An array of custom parameters.

You can also create a template wizard so that it is available for future use. A template wizard, after it is created, is added to the Add Project or the Add Item dialog boxes.

Macros

Macros are code snippets that you can invoked through a menu or a shortcut key, and you use them to automate repetitive tasks. Visual Studio .NET has a Macro IDE that lets you create, debug, and execute macros. The user interface for the Visual Studio .NET macro IDE is similar to the IDE for other development tools except that the Project Explore, Task List, Command Window, Properties Window, Class View, Dynamic Help, Toolbox, Object Browser and Web browser are designed specifically for the Macros environment. The macros that you code in the Visual Studio .NET macros IDE are written in Visual Basic .NET. Using the Macro IDE, developers can automate routine tasks and extend the functionality of the IDE, such as turning line numbers off and on, stripping tab spaces, saving and loading a view, and so on.

The Visual Studio .NET Macro IDE also has the macro recorder that allows a user to automatically record macros. The macro recorder records the keystrokes when it is in the recording mode. Once recording stops, the keystrokes are translated into code and stored. This provides an excellent learning tool for novice users. To begin the macro recording, select **Macros** from the **Tools** menu and then select **Macro Recorder**. The Recorder toolbar appears on the screen. The Recorder toolbar has buttons to pause, stop, and cancel recording. You can also control this operation from the **Macros** option in the **Tools** menu.

After you record and store a macro, you can run it from either the Macros IDE or the Visual Studio .NET command window. You can also place them on a menu and run them from there. Every time you record a new macro, the macro gets recorded as a temporary macro. This macro is not saved unless you save it explicitly by choosing the **Save Temporary Macro** option in the **Macros** submenu, which is under the **Tools** menu. The temporary macro is available until you record the next macro or close the current session with the IDE.

Home Page

The Visual Studio .NET start page is a central location for various features. From here you can do the following:

- Create a new project.
- See a list of recently opened projects by clicking on the **Get Started** option.
- Find information about the new features in Visual Studio .NET and check for Visual Studio .NET updates by clicking on the **What's New** option.
- Set preferences through the **My Profile** option.
- Get online help through the **Search Online** option.
- Get information on the latest happenings through the **Headlines** option.
- Get detailed information on hosting your solutions through the **Web Hosting** option.
- View the latest news on the MSDN online library including all announcements related to seminars and technical presentations by clicking on the **Headlines** option.

- Interact with various developers and other experts in the field through the **Online Community** option.

The start page also serves as a Web browser for the IDE. You can configure this Web browser to be docked, hidden, or floating.

The What's New, Online Community, and Headlines options can periodically receive updates from the Internet. The updates are received whenever you click on any of these options when connected to the Internet. If you are not connected, the last updated information is available. You can also customize what you see in these pages by configuring the filter that is available. For example, two filters are available to see Visual Basic–related information. The Visual Basic Related and Visual Basic options allow you to view only information and news related to Visual Basic. Note that the filter setting also affects what topics you view if you have MSDN installed.

The My Profile option allows you to customize various options of the IDE. These options set your working preferences in the IDE. The My Profile options consist of the following configurable parameters:

- **Profile** The profile option is used to set the keyboard scheme and layout of windows and to filter MSDN help. If you change either one or all of the options mentioned in the previous paragraphs to suit your needs, the profile option is reset to custom. If you choose Visual Basic Developer as your profile, the corresponding keyboard scheme, window layout, and the MSDN filter are set to those options that resemble Visual Basic 6.0.
- **Keyboard Scheme** The keyboard scheme lists the various shortcut key combinations that are available for various options such as running a solution, debugging, turning on or turning off breakpoints, and so on. If Visual Basic developer was chosen as the profile, the keyboard scheme is automatically set to the layout similar to Visual Basic 6.0. For example, the function key F5 is used to run a project in Visual Basic. The same key can now be used in Visual Studio .NET because the keyboard layout is now the same as in Visual Basic 6.0. This allows you to leverage existing knowledge and does not require you to learn new keyboard configuration.
- **Window Layout** The Window Layout configuration allows you to configure the toolbar, solution explorer, server explorer, and so on to the layout similar to previous versions of Visual Basic or Visual C++. If you

choose Visual Basic 6.0 as the option for Window Layout, then the IDE places the Server Explorer window on the left of the IDE and auto-hides it. The toolbox is docked on the left. The properties window and Dynamic Help window are tab-docked at the bottom. The Solution explorer and Class-view window are tab-docked on the right and on the top of Properties window.

- **Help Filter** The Help filter lets you configure the topics that are relevant to your scope. This feature was available in the earlier versions of MSDN as well. Note that this option does not apply to the content shown in the Dynamic Help window. By choosing Visual Basic or Visual Basic Related in the Help filter, you can view all topics related to Visual Basic documentation as well as topics relating to Visual Database tools, source code control, and the .NET Framework Software Development Kit (SDK).
- **At Startup Show** This option indicates what should appear when you start Visual Studio .NET. The choices that you can choose from are Visual Studio Home Page, Most Recent Solution, Open Project Dialog, New Project Dialog, or an Empty IDE.

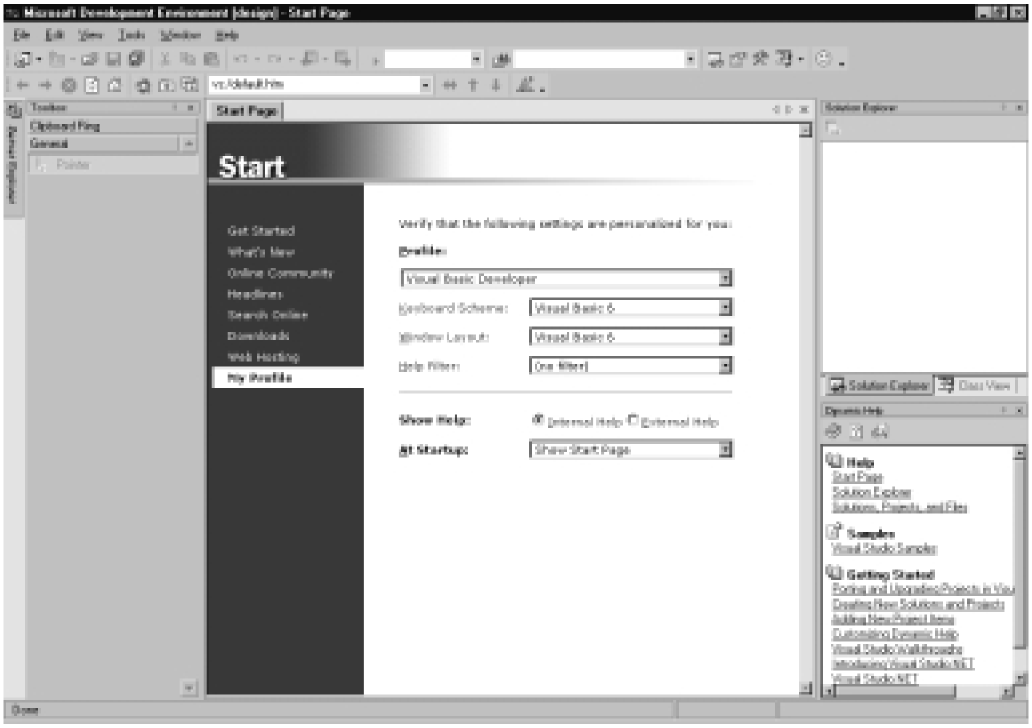
When you check the **Open links from within the start page in a new window**, the topics or links that you view from the Visual Studio Start Page are opened in a new window. When you click on the **Get Started** hyperlink, it displays the Get Started option in the Visual Studio .NET home page.

Figure 3.16 shows you the IDE after Visual Basic has been chosen as the layout. Note the position of Server Explorer, Toolbox window, Solution explorer, Class view, Properties, and the Dynamic Help window.

Project Options

The project options that are available in Visual Basic .NET are different from the previous versions of Visual Basic. WebClasses and DHTML applications have been removed and changes have been made to the Standard EXE, ActiveX EXE, and ActiveX DLL projects.

Figure 3.16 Visual Studio .NET Start Page Configured for Visual Basic



Verify that the following settings are personalized for you:

Profile: Visual Basic Developer

Keyboard Scheme: Visual Basic 6

Window Layout: Visual Basic 6

Help Filter: (no filter)

Show Help: [✓ Internal Help] External Help

At Startup: Show Start Page

The list of new projects is shown when you want to create a new project or when you want to add a new project to your existing project. A project template is associated with each project icon that you choose. This, in turn, determines the output type and the other project options that are available for this project. The project types that are available in Visual Basic .NET are distinctly different from those in Visual Basic 6.0. Table 3.2 shows the various project types available in Visual Basic .NET.

Table 3.2 Project Templates in Visual Basic .NET

Project Type	Description
Windows Application	A Windows Application project type is used to create a windows-based application that has the Windows forms as the primary tool for user interface. This template creates a project with a default form with a set of related references to libraries present in the System namespace.

Continued

Table 3.2 Continued

Project Type	Description
Windows Control Library	A Windows control library is similar to the ActiveX control project found in Visual Basic 6.0. The template creates an empty container and references the related libraries in the System namespace. You can build the user interface for the control, with the help of existing controls, in the empty container. Once the controls are built, you can use them along with the existing controls provided by the IDE.
Class Library	A project type for creating classes to use in Windows-based applications. This template creates a project with a default class with references to libraries in the System namespace. The default interface that Visual Studio .NET provides is that of a blank screen. The user-interface is created by dragging and dropping controls from the toolbox.
ASP.NET Web Service	A Web service is typically a middle-tier business functionality that is exposed through the HTTP protocol. This project type allows you to create a Web service.
ASP.NET Web Application	A Web application project is primarily used to create Web pages that serve as the user interface.
Web Control Library	A Web control library project is used to create controls for Web applications. The template creates a Web control template with default properties. You can then customize the control to your requirements.
Console Application	This project type is used to create applications that do not have a user interface. They are typically invoked from the command prompt. A console application project contains a module with only subroutine called **Sub Main**.
Windows Service	A windows service project is used for creating services for Windows. A window service project template consists of a blank screen similar to that of a class library. The template also creates a module called user services, which contains the basic framework that will help you get started on coding windows services.

Continued

Table 3.2 Continued

Project Type	Description
New Project in Existing Folder	A wizard for creating a project in an existing folder. The wizard allows you to create an empty project in an existing folder. The wizard queries the user for the name of the folder in which the project is to be created and creates an empty project in the specified folder. You can use the Import Folder Wizard when you already have an existing project configured for a specific functionality, and you merely want import it to your existing solution.
Empty Project	An empty project for creating a Windows application. The empty project template creates an empty project. You can then add necessary references, Windows forms, and other project items as necessary. The difference between an Import Folder Wizard template and an Empty Project template is that the Empty Project template creates the specified folder if it does not exist, whereas an Import Folder Wizard requires that a folder be present.
Empty Web Project	An Empty Web Project template is similar to an empty project. The only difference is that it allows you to create a Web application instead of a local application.

Visual Studio .NET supports a variety of file types and their related file extensions. Visual Studio .NET uses two file types to store settings specific to solutions. The file types are SLN and SUO. The SLN file is the Visual Studio solution and it organizes projects, project items, and solution items into the solution by providing the environment with references to their locations on disk. The SLN file is analogous to a Visual Basic group (VBG) file found in Visual Basic 6.0. The VBG file is created if the application contains one or more projects. It also acts as a logical container to various miscellaneous files that are opened outside the project group. The SUO file contains the solution user options and stores all of the options that are associated with the solution. This helps in restoring the customizations each time the project is opened.

Debugging…

Debugging Various Projects

Visual Studio .NET introduces new project types that allow you to build applications that can take advantage of the .NET framework. The introduction of these new project types also means that you can employ some new techniques while debugging these project types. This following list discusses some of the project types and the procedure to debug these project types:

- You can debug Windows Application projects by choosing **Start** from the **Debug** menu.
- Class Library Projects are very similar to DLLs. Because DLLs are hosted by an application, you need to debug the host application as well. If the host application is a managed-code application, you will be able to debug the DLL as part of the application. But if the host application is an unmanaged-code application, you need to attach a debugger to the process.
- Windows Controls projects are similar to class library projects. They cannot be debugged during design-time. A Windows control is usually added to a Windows form. Once the control is instantiated, you can set breakpoints in your code to debug the control.
- Console Application projects have special debugging mechanisms. Console application projects may require the use of command-line parameters to start the debugger. You can specify command-line parameters in the application's property pages. Once specified, these are stored with the solution.

Toolbox

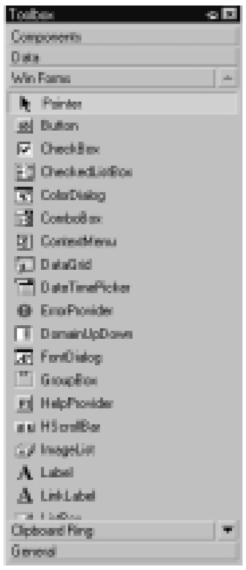
The Toolbox window is organized into various tabs and contains a host of user controls for use in Visual Studio .NET. You can open the Toolbox window by choosing **Toolbox** from the **View** menu. The controls in the Visual Studio .NET IDE have been categorized under different headings. Each heading is represented by a tab in the Toolbox window. Thus, the toolbox contains the following tabs:

- Windows Form Controls
- Data Controls
- System Controls
- HTML Controls

The toolbox has the unique feature of context-sensitivity in relation to the designer. So, if you are designing a Web form, only the HTML Controls tab is displayed. Or, if you are designing a Windows form, only the Windows Form controls are displayed. This reduces a lot of clutter and facilitates ease of use.

Two tabs are displayed by default when you open the IDE: the General tab and the Clipboard Ring tab. You can customize the toolbox window by adding more tabs. Each tab in the Toolbox window, even the ones you create, has an item called the Pointer, indicated by an arrow that points diagonally to the left. The purpose of this item is to return the cursor to its original state. For example, suppose you choose to add a Listbox to a Windows form. You then changed your mind to include a ComboBox instead of a Listbox. Because you have already selected a Listbox, you will have to place the Listbox on the form, delete it, and then choose the ComboBox. You can do this more efficiently by clicking on the **Pointer** button. So in this situation, before placing the Listbox control on the form, click on the **Pointer** button to return the cursor to its original state, and then choose the ComboBox control. Figure 3.17 shows you the picture of a toolbox.

Figure 3.17 Toolbox Window



You can customize the appearance of the toolbox and its items by using various methods:

- **Add and remove tabs** In order to add a new tab, right-click on a tab and choose **Add Tab** from the shortcut menu. A textbox is displayed at the bottom of the toolbox window. Give a suitable name for the tab. The new tab appears as the last tab in the Toolbox window. Once added, you can use the new tab to store items. Note that the pointer item is automatically added to the new Toolbox.
- **Add and remove items contained in the tabs** Right-click on the tab that you want to customize. Choose **Customize Toolbox** from the shortcut menu. This displays a tabbed window displaying various control classes with each class containing different controls. The various classes are COM controls, Modeling shapes, General shapes, and .NET framework components. Each control is displayed with a checkbox alongside it. You can check to add a new control to the tab. If a control already exists in the tab, the control is already checked. You can uncheck it to remove the control from the tab.
- **Rename tabs and items** Right-click on the tab that you want to rename and choose **Rename** from the shortcut menu. Type the new name in the textbox and press **Enter**.
- **Choose to display all tabs and hide unwanted tabs** You can choose to display all tabs or let the IDE decide which tabs to display depending on the context. If you choose to display all tabs, right-click on any of the tabs and choose **Show All Tabs**. This option toggles on or off.
- **Choose the type of view for items displayed in the tabs** You can choose to configure how the items on the tab are displayed. The options that are available are Compact View and List View. In Compact View, the items are displayed without their names. Use this option if you are familiar with controls and can identify the control just by looking at it. The List View option displays the controls with their associated text.. To change the view, right-click on the tab and choose **List View**. This is a toggle-on-or-off option. If a tick mark is displayed, the current view is that of a List View.
- **Sort items in the tab** The items in the toolbox can be sorted alphabetically. You can do this by right-clicking on the items area and choosing **Sort Items Alphabetically**.

- **Reposition the items in the tab** You can reposition items displayed in the toolbox by clicking on an item and choosing **Move Up** or **Move Down**. The action of moving up or down depends on the view. If a compacted view is selected, the Move Up option moves the item to the left and the Move Down option moves the item to the right.

The toolbox normally contains the following tabs if **Show All Tabs** is chosen:

- **XSD Schema** The XSD Schema tab contains items that are used when creating schemas.
- **Dialog Editor** The Dialog Editor contains items such as Button, text box, list box, and so on. These items pertain to those that are normally used in a dialog box.
- **Web Forms** Web form controls contain controls such as hyperlink, ImageButton, and so on.
- **Components** The Components tab contains controls that allow access to system operations. The following controls are part of the components tab: FileSystemWatcher, EventLog, Directory Entry, Directory Searcher, Message Queue, Performance Counter, Process, Schedule, Service Controller, and Timer.
- **Data** The Data tab contains controls that can be bound to data. You use these when you connect to a database and retrieve data from it. Some of the controls that are a part of the Data tab are DataSetView, DataView, SQLConnection, ADOConnection, and so on.
- **Win Forms** The Win Forms tab contains controls that are normally used in Windows forms.
- **HTML** The HTML tab contains controls that are used to format a HTML page.
- **Clipboard Ring** The Clipboard Ring tab is similar to the clipboard functionality offered by an operating system. In this case, the clipboard ring stores code that was either cut or copied within the IDE. Each item is stored in the Clipboard Ring tab. Once stored, you can place the cursor in the appropriate position in the code editor window and double-click on the appropriate item in the clipboard ring tab. So, for example, if you have a subroutine that you frequently refer to, you can code it for the first time and then copy it. Once copied, it is stored in

the Clipboard Ring tab. Subsequently, whenever you want to reference the procedure in code, you only have to double-click to paste it in the code-editor window.

- **General** The General tab is provided to the programmer as a matter of convenience. You can store frequently-used or user-created controls in the General tab.

Child Windows

Visual Studio .NET contains various tools and options that allow you to configure windows present in the IDE. Windows are displayed in the IDE in two ways: Multiple Document Interface (MDI) mode and Tabbed Document mode. In the MDI mode, the IDE provides a parent window that is a container for all other windows. All windows that are opened are opened within the context of this container. In the tabbed document mode, all windows are tabbed. You can choose the appropriate document by clicking on the corresponding tab. This is the default mode that Visual Studio .NET uses. You can configure the IDE to choose a specific mode by choosing **General** under the **Options** submenu in the **Tools** menu. Figure 3.18 shows you the IDE that uses the tabbed mode.

Figure 3.18 Tabbed Mode

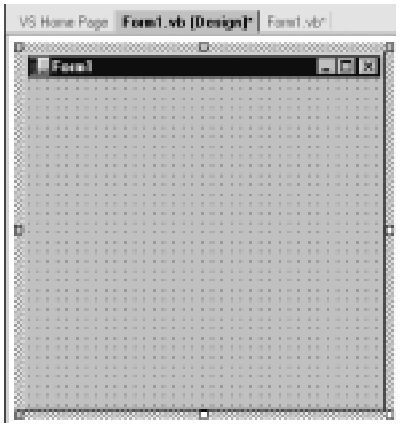


Figure 3.19 shows you the IDE when it is configured to use the MDI mode to arrange windows.

It is interesting to note the changes made to the menu items related to arranging windows of the Window menu. When the IDE is configured to use the

MDI mode, the menu items in the Window menu change from Tile Horizontally and Tile Vertically to New Horizontal group and New Vertical group. Choosing the new Horizontal or new Vertical group splits the existing screen vertically or horizontally and places the active tab in the new pane. Figure 3.20 is an extension of Figure 3.18 after a new horizontal group is selected.

Figure 3.19 MDI Mode

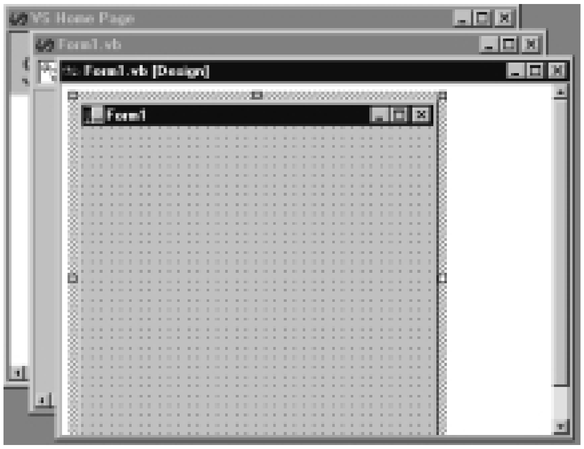
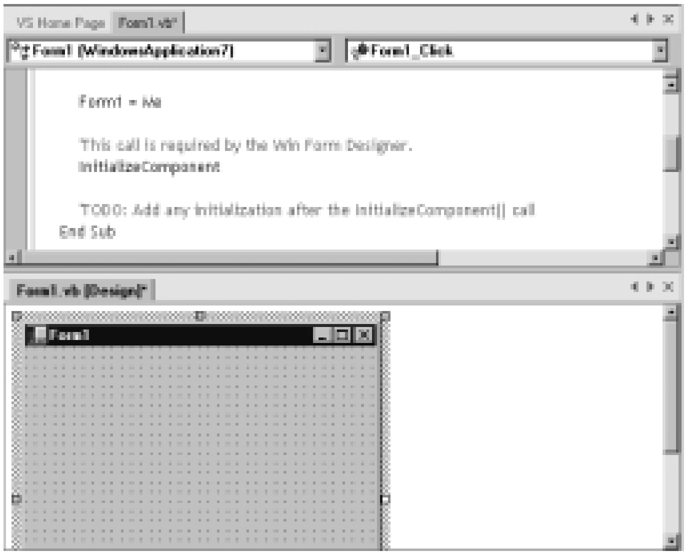


Figure 3.20 Tabbed Mode with a New Horizontal Group



```
        Form1 = Me

        'This call is required by the Win Form Designer.
        InitializeComponent()

        'TODO: Add any initialization after the InitializeComponent() call
    End Sub
```

Window Types

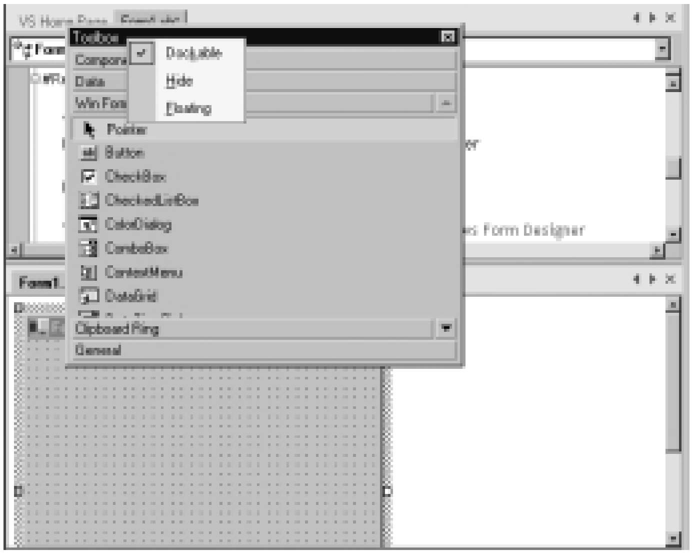
The IDE consists of two types of windows:

- Tool windows
- Document windows

Tool windows are those that are listed in the View menu. They are defined by the current application. You can configure the tool windows to show or hide automatically, link with other tool windows, dock against the edges of the IDE, and float over other windows.

Tool windows can be made dockable or undockable by selecting or deselecting the **Dockable** option. *Docking* is a term used when two or more windows are combined. This option is available on the shortcut menu when you right-click on the tool window. When you make a window dockable, it floats over other windows or it snaps to the side of the application window. When a tool window is in an undocked state, it appears as a document window. A document window appears a child window if the IDE is in a MDI mode, or it appears as a tabbed window if the IDE is configured to use the tabbed window option. Figure 3.21 shows you the illustration with the Toolbox window set in a docking state.

Figure 3.21 Docked Toolbox Window



Arranging Windows

The IDE allows you to arrange tool windows and document windows in such a way that it maximizes the viewing area. You can dock or hide tool windows, tab dock windows, or even tile document windows.

In order to dock or hide tool windows, select **Dockable** from the **Window** menu and drag the window toward the edge of the IDE window until you see a superimposed outline of the window in the location you want. You can also move the tool window without letting the window snap into its place. In order to achieve this, press the CTRL key as you move the window. In order to hide the window, you can right-click on the tool window and choose the **Auto Hide** option. Or, if the window is already docked, you can hide the window by clicking on the push-pin option on the window. If the push-pin is pointing down, then the Auto Hide is disabled; if it is pointing horizontally, the Auto Hide option is turned on.

In order to tile document windows, if the IDE is configured to use the tabbed document mode, select a tab and drag it below or above the current document title. A rectangular outline appears on the area in which it will be placed. Alternatively, you can do the same by selecting the **New Horizontal Group** or **New Vertical Group** from the **Window** menu. If the IDE is in the MDI mode, you can choose the **Tile Horizontally** or the **Tile Vertically** option from the **Window** menu.

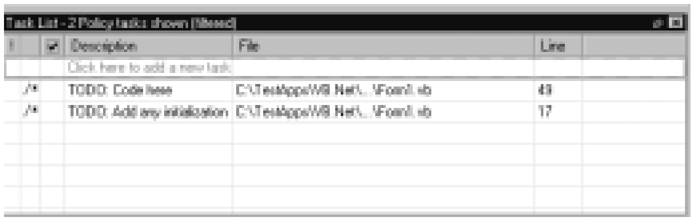
Task List

The TaskList window allows you to organize and manage your development process. You can associate this to a TODO list, which you might have to complete a set of tasks. You can display the Task List window by selecting **TaskList Window** under the **Other Windows** submenu on the **View** menu. The task list window can help you do the following:

- Locate build and compile errors
- Mark items as completed as you complete each task
- Add user notes in the solution
- Filter task list according to the predefined views
- Sort entries in the TaskList by Priority, Category, Checked, Description, File, or Line

Figure 3.22 shows a sample TaskList window. The advantage of using tasks is that you can double-click on a task that is listed in the task list window, and the IDE directly takes you to the document and the line for editing. You can add tasks by adding a comment in your code, followed by the token TODO:. Once added, the corresponding task is listed in the TaskList window. You can also configure the TaskList window to display a custom token. You can use a custom token to represent a user-defined situation in a solution. For example, you might want to add a custom token with the name FUTURE. This could represent features of your application, which will not be implemented in the current version but might be implemented in future versions. So, you can mark portions of code that will be implemented in the future with this custom token. This also serves as a reminder when this project is revised for later editions. Exercise 3.3 guides you through the process of setting up a custom token.

Figure 3.22 TaskList Window



Task List - 2 Policy tasks shown (filtered)

!		✓	Description	File	Line
			Click here to add a new task		
	/*		TODO: Code here	C:\TestApps\VB.Net\...\Form1.vb	43
	/*		TODO: Add any initialization	C:\TestApps\VB.Net\...\Form1.vb	17

Exercise 3.3 Setting Up a Custom Token

1. Click on the **Tools** menu and choose **Options**. In the Options window, click on the **Environment** tab and choose the **Task List** option.
2. Type a name of the custom token in the textbox that is present below the **Name: caption**.
3. Click on **Add** to add it to the list of tokens. You can also set the priority of the token to **High**, **Normal**, or **Low**.

TaskList Views

You can configure the TaskList to displays tasks according to predefined views. In order to do this, right-click on the window and choose the **Show Tasks** option. This option lists various views that you can configure to view the tasks relevant to the current situation. Table 3.3 lists the various views.

Table 3.3 TaskList Views

Category	Description
Previous View	The Previous View option restores the view that was in effect before the current view. For example, if your previous view was set to All and the current view is set to Comment, choosing Previous View restores the view to list all tasks for the current project.
All	Displays all the tasks for the current project. No filter is applied.
Comment	The Comment view displays comments in the code that includes the standard comment tokens and custom comment tokens. Any change made to the comment token in the form of editing or deleting has an immediate effect on the TaskList view. You can remove a comment item from the TaskList window by removing the comment from the code.
User	You can add a user task manually by entering it in the column that has the *Click here to add a new task* text. These can be checked off as completed when you complete them.
Shortcut	The shortcut is used to point to the code in the solution that you frequently refer to. For example, if you have declared a number of constants in your solution, and you frequently refer to it, you can mark the first line where declarations start and refer to it as you code. In order to add a TaskList shortcut, select the line of code that you want as a shortcut. Select **Bookmark** from the **View** menu and choose **Add Task List Shortcut** from the **Bookmark** submenu. Once added, the TaskList window displays the shortcut if the current view is set to All or if the current view is set to Shortcut.
Policy	The policy view lists errors thrown by the Template Description Language. The Template Description language is the notation used to write the policy files of Visual Studio Enterprise Templates. These policies define the structure of an enterprise application. You can choose to view policy messages by selecting **Policy** from the **Show Tasks** shortcut menu. In order to remove the policy message from the TaskList window, fix the problem and reopen the solution.
Current File	Lists all tasks for the file currently in view. There is a slight difference between the All and the Current File views. The All view shows you all views in all the files, the Current File shows you all tasks in the current file only.

Continued

Table 3.3 Continued

Category	Description
Checked	The Checked option shows you all tasks that have been checked off as completed.
Unchecked	The Unchecked view shows you all tasks that have not been checked.

Locating Code

The IDE provides you with several options that allow you to browse through documents to locate lines of code. These features make working with the IDE easy, particularly when you have a solution that contains numerous files containing many lines of code. You can bookmark various lines of code and navigate through the bookmarks using the **Next bookmark** and **Previous bookmark** commands. In addition, you can annotate code by adding a standard comment token or a custom comment token and adding shortcuts to a line of code. You can also scroll through the documents that have been edited in the current session by using the Forward and Backward toolbar items.

Annotating Code

Annotating code is the process of adding user information to the code. Annotating code usually takes the form of comments. Visual Basic .NET allows you to annotate code by adding standard comment tokens and custom tokens, which are listed in the TaskList window. When you double-click on a task listed in the TaskList window, the IDE automatically takes you to the code location. Note that comment tokens in HTML or CSS or XML markup are not displayed in the Task List. Annotating code has various advantages:

- It makes the code more readable. But you must exercise caution here. Too much annotation might make it look like more of a story, thus defeating the main purpose of making the code self-describing.
- It makes it easier to view changes made to the code over a period of time, if the programmer indicates what has been changed.
- It also helps to understand the programming logic used by a programmer.

In order to add a comment link to the TaskList window, enter the comment marker for Visual Basic .NET, which is an apostrophe ('). Then begin the comment with one of these tokens: TODO, HACK, or UNDONE. You can then write the comment text after this token. Once you add this to your code, the TaskList view is automatically updated. If you do not see this in the TaskList window, check out the filter settings.

You can also create custom tokens other than the default tokens of TODO, HACK, or UNDONE. These custom token also serve as personal markers. In order to do this, select **Options** from the **Tools** menu. Select **Environment** and then choose **task list**. In the Comments token text field, type the name of the token and click **Add**. You can also set the priority of the token to **Normal**, **Low**, or **High**.

Another form of annotating code is to include shortcut to code. In order to add a TaskList shortcut, select the line of code that you want as a shortcut. Select **Bookmark** from the **View** menu and choose **Add Task List Shortcut** from the **Bookmark** submenu. In order to remove the shortcut, choose the **Remove Task List Shortcut** from the **Bookmark** submenu.

Solution Explorer

The Solution Explorer in Visual Studio .NET is the equivalent of the Project Explorer found in the previous versions of Visual Studio. The Solution Explorer organizes the files contained in the current solution. Figure 3.23 shows you an illustration of the Solution Explorer.

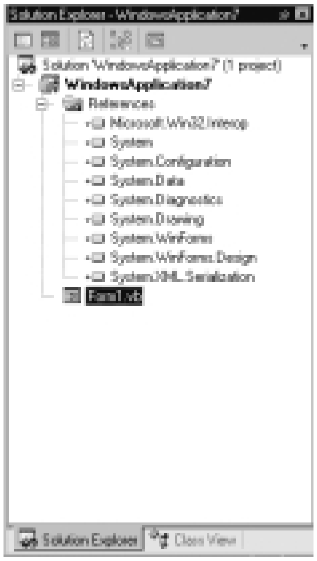
The main purpose of a Solution Explorer is to manage files contained in a solution. The Solution Explorer also helps you move and copy files within a solution, select multiple files to perform a single operation related to the selected files, and assign a project in a multiple-project environment as a startup project.

The Solution Explorer provides a limited set of toolbar buttons that allow you to perform specific operations on the object that is currently in view. For example, if you are working on a form, the Solution Explorer will display five different toolbar buttons. The purpose of each toolbar button, shown in Figure 3.23, is as follows:

- Clicking on the first toolbar button opens the code editor for the form. This is identified by the icon with some lines in a window.
- Clicking on the second toolbar button displays the form designer. This is identified by the icon that has two boxes in the window.
- Clicking on the third toolbar button refreshes the Solution Explorer's view. This is represented by two arrows following each other.

- Clicking on the fourth toolbar button displays all the files that are contained in the solution. Normally, only the forms, classes, and references are displayed. Miscellaneous files, such as object and debug files, are not displayed by default. This is identified by a series of small file icons.
- Clicking on the last toolbar button displays the properties for the selected object, if a property page is available. Thus, when either a solution or a project in the solution has the focus, you can click this button to bring up the properties for that object. This is represented by a tabbed dialog box.

Figure 3.23 Solution Explorer



Solution Explorer allows the user to perform many file and project related management tasks. Some of the most common tasks include moving and copying items, setting up a startup project, selecting multiple items, assigning a project, in a multiproject environment, to be a startup project, and so on.

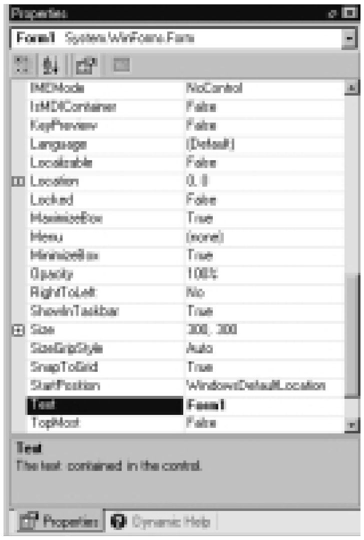
You can perform common file operations, such as move or copy, on the files present in the Solution Explorer. Moving and copying files, in this context, merely refers to referencing the name of the file. So, when you click on a form and drag it onto the code editor window and drop it, the physical path of the form is displayed in the position it was dropped. You can perform other file operations as follows:

- **Opening files** You can open files from Solution Explorer by merely double-clicking them. You can also change an item's default editor by right-clicking the item and choosing **Open With…** from the shortcut menu.
- **Multiple Selection** You can select multiple items from a single project or across multiple projects in a single solution. If you need perform the same operation to a set of files, you can multiselect all these items and perform the operation only once. For example, if you want change the properties of two or more items or exclude only these items from the project. Note that when you select multiple items, the commands available are the ones that are common to both the items.
- **Startup Project** You can set a project, in a multiproject solution, to be a startup project. This is the same as in the previous versions of Visual Studio. The Solution Explorer displays the name of the startup project in bold.

Properties Window

The Properties window, shown in Figure 3.24, lets you set properties for user controls and other objects present in the form or a designer. Note that the Properties Window displays only design-time properties. Runtime properties are not displayed in the Properties Window.

Figure 3.24 Properties Window



Form1 System.WinForms.Form

IMEMode	NoControl
IsMDIContainer	False
KeyPreview	False
Language	(Default)
Localizable	False
Location	0, 0
Locked	False
MaximizeBox	True
Menu	(none)
MinimizeBox	True
Opacity	100%
RightToLeft	No
ShowInTaskbar	True
Size	300, 300
SizeGripStyle	Auto
SnapToGrid	True
StartPosition	WindowsDefaultLocation
Text	**Form1**
TopMost	False

Text
The text contained in the control.

The dropdown listbox that you see on top of the Properties box lists the various controls that are on the form, including the form itself. When you select multiple objects in a form or in a designer, the dropdown listbox does not display anything. The properties that will be displayed are the ones that are common to all selected objects.

The first toolbar button that you see below the listbox is the Categorized button. This is represented by the plus and minus signs. When you click on this button, the properties window lists all properties and its values for the selected object after grouping it by category. Each category is a grouping of logically related properties. For example, a Windows form's properties can be categorized as follows:

- Accessibility
- Appearance
- Behavior
- Data (Bindings)
- Design
- Focus
- Layout
- Misc
- Window style

The second toolbar button lists all the properties alphabetically. This is represented by the letter *Z* below the letter *A* followed by a down arrow. When you click on this button, all properties are sorted in alphabetical order. The third toolbar button is used to display the properties of the document. The properties are displayed for the object that is currently selected.

Form Layout Toolbar

The form layout toolbar contains various options to align controls on the form. This toolbar is very helpful in building an attractive user interface. Table 3.4 lists the various toolbar buttons and their descriptions.

Table 3.4 Form Layout Buttons

Toolbar Button	Description
	Align the selected controls to the grid
	Align all the selected controls to have the same left coordinates
	Align all the selected controls to have the same center coordinates
	Align all the selected controls to have the same right coordinates
	Align all the selected controls to have the same top coordinates
	Align all the selected controls to have the same middle coordinates
	Align all the selected controls to have the same bottom coordinates
	Make all selected controls to be of the same size
	Make all selected controls to be of the same height
	Make all selected controls to be of the same width
	Size selected controls to grid
	Configure selected controls to have the same horizontal spacing
	Increase the horizontal spacing between the controls
	Decrease the horizontal spacing between the controls
	Remove the horizontal spacing between the controls
	Configure selected controls to have the same vertical spacing
	Increase the vertical spacing between the controls

Continued

Table 3.4 Continued

Toolbar Button	Description
	Decrease the vertical spacing between the controls
	Remove the vertical spacing between the controls
	Center controls horizontally
	Center controls vertically
	Bring the selected control to front
	Move the selected control to back

Hide/Show Code Elements

The code editor in Visual Studio .NET gives you the option of outlining code. This feature reduces clutter in your code editor and allows you to see only the current code you are working with. Outlined code is not deleted—it is merely collapsed. You can identify outlined code by a rectangular box containing three dots. Outlining code is an effective way to work only with relevant subroutines or functions.

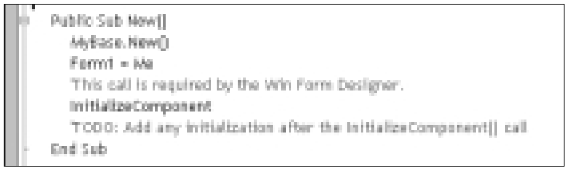
The **Collapse…** or **Expand…** option in the shortcut menu allows you to hide or show code elements by selecting the contents of the procedure or function. If the code is collapsed, you see a rectangular box containing three dots after the name of the function. In order to expand the code, you can either double-click the rectangular box, click on the plus sign found in the left corner of the code editor, or choose **Expand…** from the shortcut menu. In order to collapse the code, choose the contents of the procedure or function and choose **Collapse…** from the shortcut menu.

Figure 3.25 shows the part of the code editor window with some collapsed. Note the plus sign on the margin and the ellipsis (…) at the end of the subprocedure. Figure 3.26 shows you the subprocedure after it has been expanded.

Figure 3.25 Code Editor with Collapsed Code

```
Public Sub New()...
```

Figure 3.26 Code Editor with the Same Code, Now Expanded



```
Public Sub New()
    MyBase.New()
    Form1 = Me
    'This call is required by the Win Form Designer.
    InitializeComponent()
    'TODO: Add any initialization after the InitializeComponent() call
End Sub
```

Web Forms

The Web forms technology is used to create Web pages that contain programming logic embedded besides code that creates the user interface. Web applications that are created using this technology can exploit the new features of browser independence, event manipulation, and enhanced scalability, to name a few. Another advantage of using Web forms is that various development languages support it. Highlighted text is similar to the MSDN.

Applications built using Web forms are spread over two layers: the user-interface layer and the business logic layer. The user interface consists of a Web form containing user controls to accept input. The business logic for the Web form consists of code that interacts with the form in the backend. The programming logic is written in Visual Basic .NET or C#. When the form is executed, the Web forms application dynamically produces the HTML output for your page. Web applications built using Web forms have the following characteristics:

- The Web forms technology involves isolating all application logic to the server. This leaves the client free to be designed so that it can run on any browser without worrying about coding for specific browsers.
- The Web forms technology provides the facility of handling events. The object model supports events on the client-side as well as on the server.
- The Web forms framework introduces enhanced state management. The Web forms framework saves the state of the forms and the controls using a state bag, session object, and an application object. A *state bag* is an extensible data structure that stores various values. This is an important

aspect because every time a page is refreshed, any form-specific values could be lost.

- Client forms created using the Web forms framework require only the services of a browser to run. No other component is necessary.

Intellisense

The Intellisense technology has been around for a long time. The advantage of Intellisense is that you do not have to remember the properties and methods that are associated with the object. In Visual Studio .NET, the Intellisense technology has been beefed up to automatically list classes across various namespaces.

The editor provides the completion on various keywords. The editor also filters tokens with respect to the current context. For example, if you are inside a subroutine and you type **End** followed by a space, the code editor quickly recognizes the context and displays Sub as a member in the drop down listbox. Another example is the usage of the Option keyword. You can use the Option keyword with Compare, Explicit, and Strict. When you type the Option keyword followed by a space, a listbox containing the three choices appears.

The code editor also supplies completion on Enum and Boolean keywords. When a statement refers to a member of an enumeration, Intellisense automatically displays a list of all the members in the enumeration. The same holds good for a Boolean statement as well. When a statement refers to a Boolean, Intellisense automatically displays true or false. Some of the options available under Intellisense are the following:

- **Member Listing** Intellisense displays the list of members related to the class or the specific namespace.
- **Parameter Info** The parameter info option displays a list of parameters that are required for the subroutine or the function and the return type if the method happens to be a function. The Intellisense feature boldfaces the current parameter to indicate the current parameter that you are working with. Intellisense has been upgraded to support overloaded functions as well. For overloaded functions, you can select which parameter list you want to view.
- **Word Completion** Intellisense does a word completion when you have entered the minimum number of characters to resolve any ambiguity.

- **Quickinfo** The quickinfo feature of Intellisense displays the signature of the function or a subprocedure. For example, if you type in **msgbox** and then select the **QuickInfo** option from the **Intellisense** submenu, the IDE displays the list of parameters that are required by the **Msgbox** function. The **Intellisense** submenu is a part of the **Edit** menu.

Customizing the IDE

The Visual Studio .NET environment allows you to customize various settings to suit your needs. You can configure the code editor, customize the start page, customize shortcut keys, customize toolbars, and so on. All these allow you to work more easily with the Visual Studio .NET environment.

Customizing the Code Editor

You can customize the code editor to change the settings that apply to the general actions and view of the code editor. You can do so by setting various options in the **Text Editor** folder, found under the **Options** submenu in the **Tools** menu. The folders under Text Editor allow you to tailor the settings on a per-language basis. You can also customize the settings in such a way that it applies to all languages. This is done by choosing the **All Languages** folder.

For example, you can configure the editor to set some Visual Basic–specific commands. You can configure the editor to automatically insert the **end** constructs. This way, if you type in an **If** construct and press the enter key, the **End If** statement is automatically inserted.

Customizing Shortcut Keys

Shortcut keys are assigned to menu items so that they can be invoked by a combination of keystrokes. This saves you time by not accessing the menu each time you want to use a particular command. Visual Studio .NET contains various keyboard mapping schemes. These schemes represent the various shortcut key combinations that are specific to Visual Basic 6.0, Visual C++ 2.0, Visual C++ 6.0, and Visual Studio 6.0. If you choose any of the predefined schemes, the appropriate shortcut key combination is assigned to the commands. For example, if the Visual Studio .NET IDE is configured to use the Visual Basic 6.0 keyboard-mapping scheme, the **Step Into** option in the **Build** menu is assigned the F8 function key. Whereas, if you choose the Visual C++ 6.0 keyboard mapping scheme, the same option is assigned the F11 function key. Besides the predefined keyboard mapping schemes, you can configure custom keyboard schemes to assign various shortcut key combinations. A list of all available commands is available in a listbox displayed below the Show Commands Containing text box. You can invoke this option by choosing **Keyboard** option from the **Environment** tab. This window is displayed when you choose **Options** from the **Tools** menu. Choose a command for which you want to assign a shortcut key. You can scope the shortcut key to be applicable throughout the IDE or only to specific editor. If you choose Global, the shortcut key is applicable to the entire IDE. Shortcut keys are a combination of text key and a nontext key. The nontext keys are **Ctrl**, **Alt**, and **Shift**. When assigning a shortcut key, place the cursor in the Press Shortcut Key(s) textbox and press a nontext key and a text key. You can then click on **Assign** and click **OK**.

Customizing the Toolbars

You can configure the toolbars to suit to your requirements. You can move the toolbar to new location by clicking and dragging it. You can also create a new toolbar, add new commands, or remove existing commands from a toolbar. Exercise 3.4 allows you to add a new toolbar to the existing set of predefined toolbars. Once added, the new toolbar is available for use just as any other pre-existing toolbar is.

Exercise 3.4 Adding a New Toolbar to the Existing Set

1. Choose the **Customize** submenu from the **Tools** menu, or right-click on the menu bar and choose **Customize** from the shortcut menu.
2. The Customize window has three tabs, which represent the Toolbars, Commands, and Options. The Toolbars tab displays the list of default toolbars provided by Visual Studio .NET along with a checkbox. You can select a toolbar by checking the appropriate checkbox. You can create a new toolbar by clicking on the **New…** button available in the **Toolbars** tab.
3. After you click the **New…** button, a dialog box appears prompting the user to type a name for the toolbar. After typing the name of the new toolbar, click **OK** to dismiss the dialog box. The newly added toolbar is selected by default and is added to the list of existing toolbars. The new toolbar is displayed as a floating toolbar in the IDE.

You can also add commands to the existing toolbars. For example, you might want to add debugging commands to the standard toolbar. Exercise 3.5 outlines the procedures for adding commands to the existing toolbar.

Exercise 3.5 Adding Commands to Toolbars

1. Choose the **Customize** submenu from the **Tools** menu, or right-click on the menu bar and choose **Customize** from the shortcut menu.
2. Select the **Commands** tab from the Customize dialog box. The Commands tab contains two listboxes that displays various categories of commands and the commands available in each category.
3. Choose the appropriate category from the Categories listbox relevant to the task that you want to accomplish. The Commands listbox is automatically updated to reflect the relevant commands available in the selected category.
4. Click on the specific command that you want to assign to the new toolbar and drag and drop it onto the new toolbar.

Customizing Built-In Commands

You can program Visual Studio commands in such a way that you can invoke them from the command window. These are the actual commands that are executed when you choose an option from the menu. For example, if you want to open a new project, select the **File** menu, choose **New…** from the submenu, and choose **Project**. The Visual Studio .NET IDE has commands built in for each of the menu items. So, in this case, the IDE executes the following command to actually accomplish the operation:

```
File.NewFile
```

You can accomplish the same operation by opening the command window and typing this command at the command prompt. In other words, the IDE has encapsulated a host of commands and provided the menu as the user-interface object. This also means that the IDE hosts a lot of other commands that have not been coded as items in the menu. Table 3.5 lists some of the unadvertised commands.

Table 3.5 Unadvertised Commands

Visual Studio .NET Built-In Command	Description
File.AdvanceSaveOptions	The advanced save options allows you to set the encoding format and also allows you to configure line endings. Line endings differ for various operating systems. In Windows, line endings are denoted by a carriage return and a line feed, whereas in Unix it is denoted only by a linefeed.
Edit.DeleteToEOL	Deletes the current line fully from the current cursor position to the end of line.
Edit.DeleteToBOL	Deletes the current line fully from the current cursor position to the beginning of line.
Edit.DocumentStart	Moves the cursor to the beginning of the document.
Edit.DocumentEnd	Moves the cursor to the end of the document.

Creating an alias helps you avoid typing a lengthy command. So, every time you invoke the specific command, type in the name of the alias and press **Enter** to invoke a command. The alias command helps you to create an alias for a command. The syntax for the alias command is as follows:

```
Alias <custom name> <command>
```

Exercise 3.6 shows this process.

Exercise 3.6 Creating an Alias

1. In the command window, specify the alias command according to the syntax by providing a custom name and the actual command that you want to alias. The Intellisense features kicks in as soon as you provide the custom alias, indicating the available commands that you can alias. For example, the following statement creates a custom alias to run the project (the appropriate Visual Studio command is **Debug.Start**):

   ```
   >alias RunProj Debug.Start
   ```

2. After you enter the command, press **Enter** to create the command. The status bar displays a message that the command is created.
3. The following command deletes an alias.

```
>alias RunProj /delete
```

You can also list the currently stored aliases by typing the alias command. The alias command without any additional parameters lists the currently configured aliases. You can also view the definition for a single alias by typing the alias followed by the custom alias name. You can clear the command window by typing **cls**. The alias cls is a custom alias for the command **Edit.ClearAll**.

Customizing the Start Page

Visual Studio .NET allows the programmer to customize the start page to include any information that is of interest to the programmer. However, you must complete a few prerequisites before the customizations can take effect. The prerequisites are the following:

- Make sure that a folder called Custom is present under <*Microsoft Visual Studio.NET root folder*>\Common7\IDE\HTML\StartPageTabs\1033. The Microsoft Visual Studio .NET root folder is the folder where you have installed Microsoft Visual Studio .NET. This is normally under the Program Files folder, which is located on the C drive. So, if you had installed Microsoft Visual Studio .NET under C:\ProgramFiles, the Custom folder must be created under C:\Program Files\Microsoft Visual Studio.NET\Common7\IDE\HTML\StartPageTabs\1033 folder.
- The content presented in the start page is actually a collection of XML files that adhere to specific XDR (XML Data-Reduced) schema. As long as the file that you create complies with the schema, the content is sure to be displayed on the start page. The .XDR file is located in <*Microsoft Visual Studio .NET root folder*>\Common7\IDE\HTML\1033. You can then create the XML file containing the required content and save it in under <*Microsoft Visual Studio .NET root folder*>\Common7\IDE\HTML\StartPageTabs\1033. Once this is done, you can refresh the start page if the IDE is already open or open the IDE to view the newly created link.

  The following code segment shows you how to customize the start page that contains links to external Web sites:

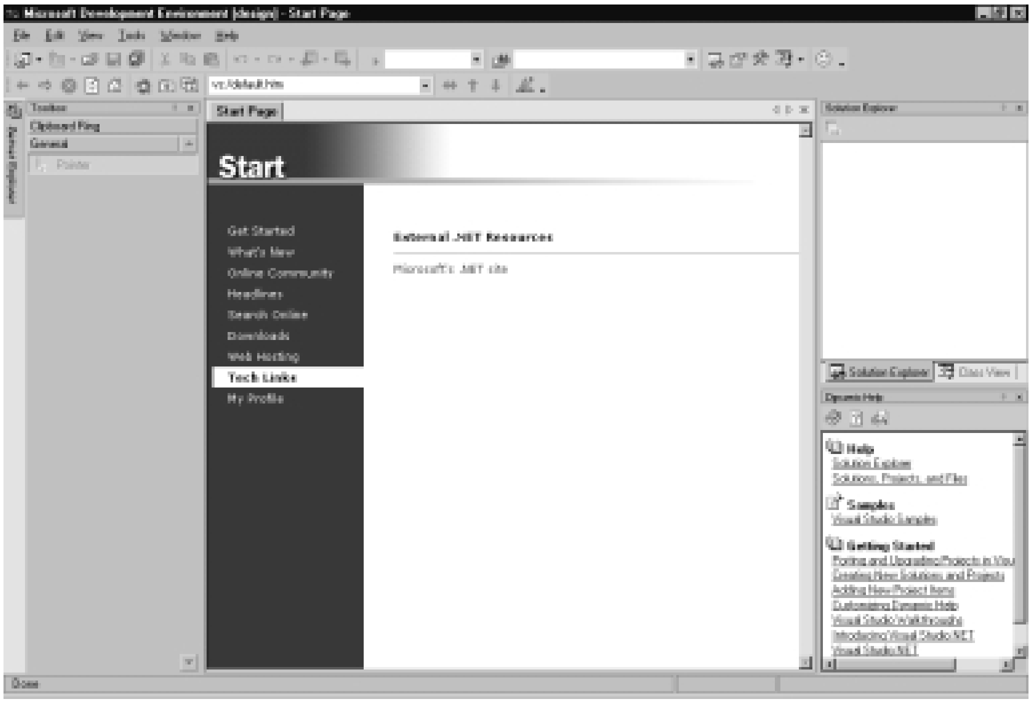
```
<?xml version="1.0" encoding="UTF-8"?>
<Tab Name="Tech Links" ID="vs_techlinks">
```

```
<Application Links" ID="vs_techlinks.app">
<Pane ID="MainPane">
<Title>External .NET Resources</Title>
<HRule/>
<LItemEx>
<LItem URL="http://www.microsoft.com/net">Microsoft's .NET
    site</LItem>
</LItemEx>
</Pane>
</Application>
</Tab>
```

The *<Tab>* element identifies a tab in the start page. In this case, the tab is titled *Tech Links*. The *ID* attribute is used to differentiate one tab from another. One of the child elements of the *<Tab>* element is the *<Application>* element. The *<Application >* element identifies the content that is contained in the tab. The ID attribute serves the same purpose as before. The right-hand side of the start page is called a pane and is associated with a *<Pane>* element. The *<Title>* child element is used to specify a title for the pane. The *<HRule/>* element is used for the purposes of formatting. The *<LItemEx>* is group element that contains a list of *<LItem>* elements. The *<LItems>* elements are used when you want to specify links. Because we are trying to link to external sites, we use the *<LItems>* elements.

This is a very basic example that shows you how to customize a start page. Further enhancements are limited only by the creativity of the individual and the support provided by the schema. Figure 3.27 shows you the IDE with the start page configured with the help of the XML file.

Figure 3.27 Customized Start Page



Accessibility Options

The Accessibility options available in Visual Studio .NET allows users to work with ease. The user interface can be configured in the following ways:

- Increasing the text size of code and menu options
- Changing the color of toolbar items, window text, and so on to make it more bright or dark depending on the requirement
- Changing the size of icons present in the toolbars to large icons
- Making the toolbars buttons more accessible by assigning text to the corresponding toolbars buttons
- Assigning shortcut key combinations to facilitate entry of frequently used text or graphics

Summary

Visual Studio .NET is available in three different versions: Academic, Professional, and Enterprise editions. You can install Visual Studio .NET on all Microsoft Windows operating systems except on Windows 95. Visual Studio .NET IDE provides a new and a comprehensive extensibility model to automate its environment. The IDE now hosts Visual Basic, Visual C++, and the new programming language C#. Visual Basic .NET and Visual C++ share the same extensibility model.

Visual Studio .NET allows a programmer to work with different objects, such as add-ins, wizards, and macros. You can create a Visual Studio .NET add-in using Visual Basic, Visual C++ or C#. A wizard is a user-interface object that helps the user complete a complex or difficult task. A macro is a collection of code snippets that you can invoke with a combination of keys. The Visual Studio .NET start page is a central location for various features provided by the IDE. You can customize the Visual Studio .NET home page to your requirements by creating an XML page containing user-specific content. Visual Basic .NET introduces new project templates that outrun the options provided by the earlier versions of Visual Studio. A toolbox window contains various tabs that contain Visual Basic controls. Visual Studio .NET introduces the task list, which allows you to track the compiler errors, syntax errors, and upgrade errors. Custom tokens allow you to configure the task list to your requirements. The code editor offers the expand/collapse feature. Visual Studio .NET also introduces a new technology called Web forms that allows creation of Web pages that can respond to events.

Solutions Fast Track

Editions

☑ Visual Studio .NET Beta2 currently is available only in what will be released as the Professional format. Microsoft plans on releasing Enterprise Developer and Enterprise Architect versions soon.

Installing Visual Studio .NET

☑ You can install Visual Studio .NET on Windows 2000 and NT 4.0. You can't install it on Windows 95, 98, or ME, although for code execution Windows 98 and higher will be supported.

☑ Visual Studio .NET and Visual Studio 6.0 can co-exist on the same machine.

The New IDE

☑ Visual Studio .NET IDE introduces a new extensibility model.

☑ All development tools are included in the IDE.

Customizing the IDE

☑ You can configure the IDE to suit the development tool that you are using.

☑ Dynamic help displays help that is context sensitive.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: Can Visual Studio .NET and Visual Studio 6.0 co-exist on the same machine?

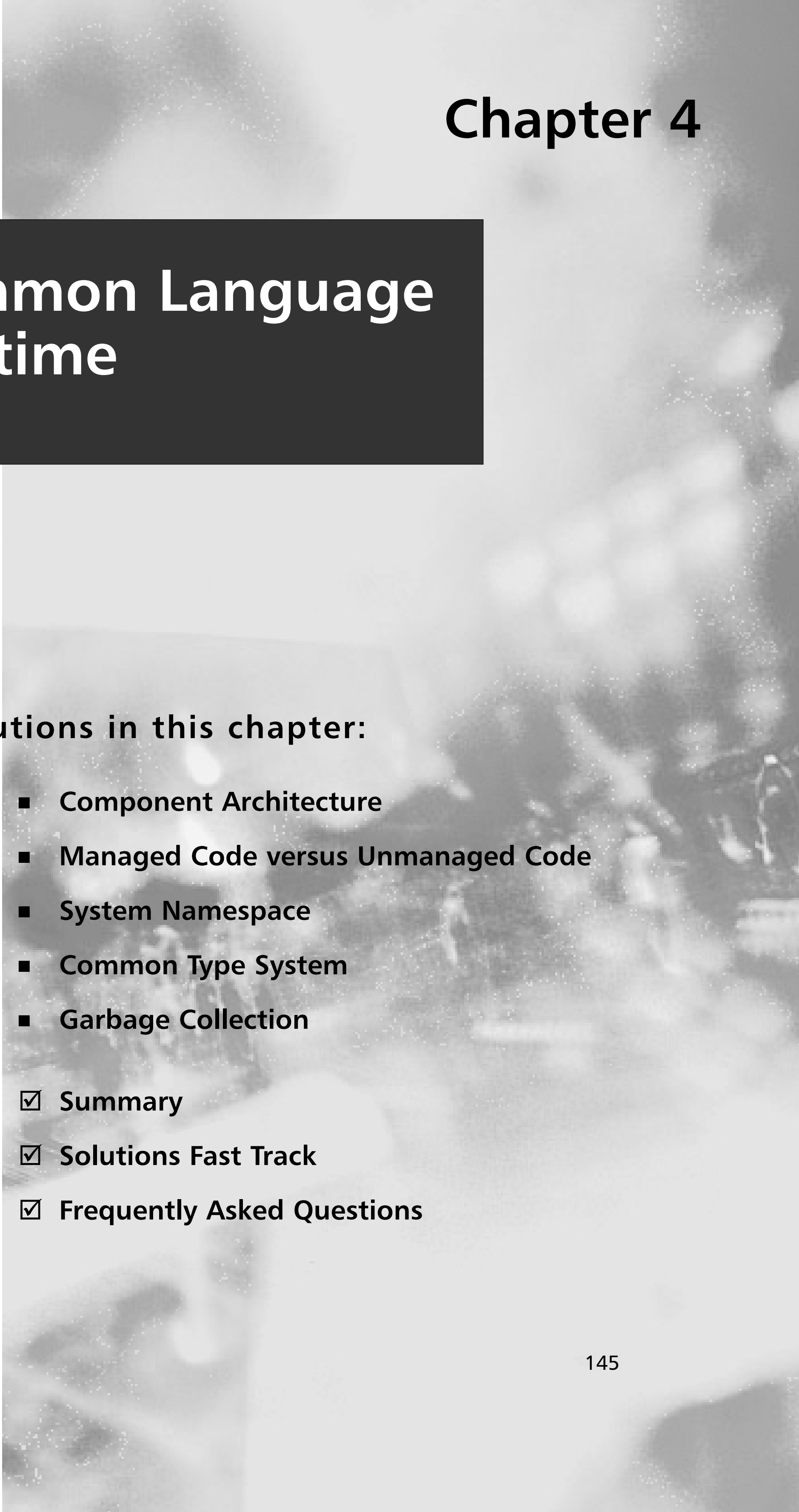
A: Yes, they can. But please be advised that Visual Studio .NET is in beta stages and you should not install it on development boxes.

Q: Does this version of Visual Studio .NET work on Windows 95?

A: No, it does not work on Windows 95.

Q: Can I develop Web applications using Visual Studio .NET on Windows 98?

A: No. You can develop Web applications only if Visual Studio .NET is installed on Windows NT 4.0 or Windows 2000. Besides, you also need the Internet Information Server installed.



Chapter 4

Common Language Runtime

Solutions in this chapter:

Introduction

As part of the .NET Framework, all .NET applications will execute using the same runtime environment. This is referred to as the Common Language Runtime (CLR). It is the driving force behind putting Visual Basic on the same footing as Visual C++, for example, as a powerful object-oriented language. The CLR will improve performance and ease the usage of components created in different languages via cross-language integration.

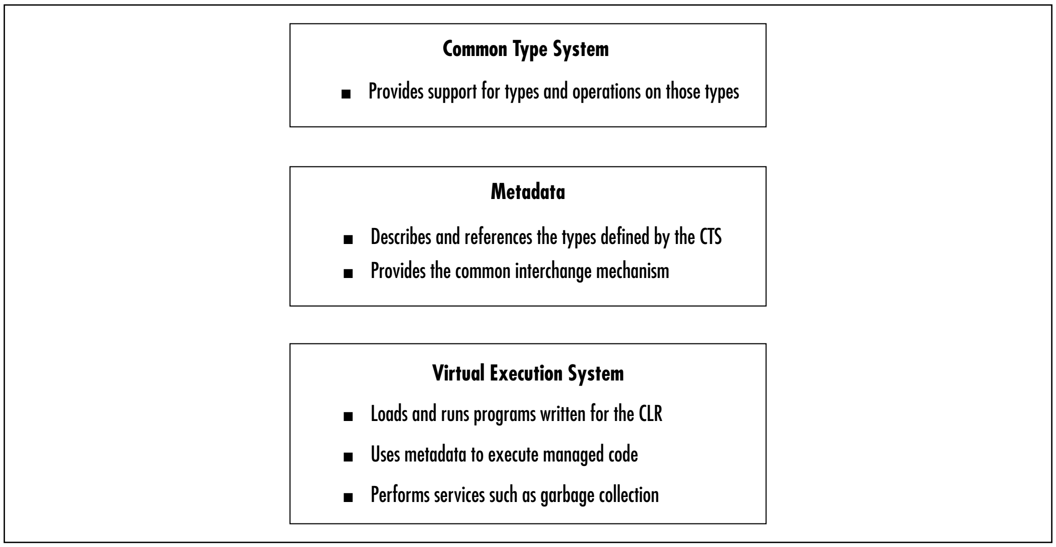
The CLR controls or manages the execution of a program. When you develop code using VB.NET, the code will be compiled for use under the control of the CLR. This is called *managed code*. This allows your code to take full advantage of the .NET Framework. If you develop code with previous versions of Visual Basic, you will create unmanaged code. It will not be able to utilize the power and benefits that the CLR brings.

The CLR (see Figure 4.1) is the heart of the .NET platform. The CLR offers such a radical change from the old runtimes, it is no wonder that .NET is often referred to as a revolution, and not an evolution, of the current Visual Studio development platform. It introduces a whole slew of new and exciting features for developers. In this chapter, we discuss some of the fantastic features of the .NET platform and the vital role the CLR plays in implementing them, including the following:

- Component architecture
- Managed code versus unmanaged code
- System namespace
- Common Type System
- Garbage collection

Every object is now inherited from within a common entity known as the System namespace and specifically, the *System.Object* class. This is the base foundation for all of your objects. Almost all of your system functionality is now included in the System namespace. This will ease development because you won't have to go digging through documentation looking for the correct Windows API call to perform a task. All tasks will be available from within the System namespace. The System namespace also contains data types. This feature allows a common type system for all languages, which gives a standard for passing data between components developed in different languages. We're sure most of you have had the wonderful experience of passing data with the Windows API or a COM component written in Visual C++.

Figure 4.1 The Common Language Runtime


Common Type System
Provides support for types and operations on those types
Metadata
Describes and references the types defined by the CTS
Provides the common interchange mechanism
Virtual Execution System
Loads and runs programs written for the CLR
Uses metadata to execute managed code
Performs services such as garbage collection

Objects will be handled differently in .NET than what you are used to. You will notice differences in how they are allocated and deallocated. A glaring change is the absence of reference counters, which removes the need for the AddRef and Release mechanisms of COM objects and allows components to use less memory and load faster. This is accomplished with the Garbage Collector, which monitors the objects and determines when they are no longer needed and releases them automatically, thereby eliminating the circular reference problem.

Components written in .NET will have many advantages over COM components. One of a programmer's biggest headaches will be alleviated. You will no longer have to register your components in the Registry. You will also be able to utilize different versions of the same COM component on the same machine. However, you will still need to use COM components—they will not disappear overnight. COM Services will allow you to access COM components from within your .NET applications.

Component Architecture

One of the most powerful capabilities that we have come to love about Visual Basic is the capability to easily create components. Reusable code in the form of distributable binaries has been a key player in the success of Visual Basic to date. It has allowed programmers to reduce code redundancy, increase productivity, and provide more scalable applications. Luckily, Microsoft is well aware of the success component development has brought to its league of programmers and has only enhanced what component development offers in their new component architecture.

Clearly, the new object-oriented features of VB.NET will be quickly embraced in the design of components. The capability to provide constructors, inheritance, overriding, and overloading gives VB developers a real step up in code reuse and extensibility (not to mention bragging rights that we are a *true* OO language now, as well). This tends to go without saying. In addition, however, one of the key benefits of designing components under the new .NET platform is that we are finally escaping from *DLL Hell*. Yes, that's right! No more versioning problems, confusion over which compatibility to set when compiling, registering and unregistering of components, and just general headaches causing keep-you-up-'til-dawn deployment issues. Those days are over, and we recommend you close the door on them. But do not lock it and toss the key just yet. COM is not dead by any means, and we discuss later in the chapter how .NET can communicate with your existing COM objects. With that said, you may be wondering how this is all possible.

Components created under the .NET platform are compiled into executables called assemblies, which serve as the building blocks of all .NET applications. Assemblies are reusable, versionable, self-describing entities that alleviate the deployment and maintenance problems often encountered with COM. Each assembly contains an assembly manifest, which is the name given to the self-describing information located therein. This makes perfect logical sense. Why store information about a component outside of the component—as was the case with COM utilizing the registry, type libraries, and so on—when the information can be self-contained. Keeping everything in one central package makes it clear where information will be located and where you need to go to find it. The assembly manifest contains the following information describing the assembly itself:

- **Assembly's identity** Its name and version number.
- **File table** Describes any other files that make up the assembly, including other assemblies, graphics files, text files, and so on (these

other file types included can be correlated to items you might have included in resource files used in previous versions of Visual Basic).

- **Assembly reference** Lists all external dependencies, such as DLLs and other files that your assembly depends on to execute, but that you did not create yourself.

The CLR can use the information contained within the assemblies at runtime to determine what dependencies the application may have and where to look for those dependencies.

In addition to alleviating the problems encountered with versioning and maintaining compatibility, the CLR provides another huge revolution to the idea of component scalability and increased productivity via true language interoperability. Originally, COM was created to allow for a standard by which its compiled binaries could be language neutral and used in any other language that understood—and could implement—COM. COM had defined a standard for defining interfaces to describe components that allowed for language interoperability…so they said. If you have ever wanted to call a Visual Basic ActiveX DLL from a C++ client application, you know that doing so really isn't all that simple. You would need to go into the IDL code for the component and do a little reworking to generate a type library that could be used and understood by the calling application.

So it works, sort of. However many of you out may have often thought, "Can't this be done more efficiently? There has to be an easier way." Well, either Microsoft has a team of mind readers, or they themselves thought the same thing. Regardless of how true language interoperability came about, we can now rejoice that it has. The CLR offers a cross-language integration scheme that cannot be matched, and a big player behind this integration is the introduction of metadata.

Metadata is essentially binary information that describes just about every aspect of your code. It contains information about every type defined and referenced within your code and assemblies. Therefore, at execution time, the CLR can extract this information, store it into memory, and, thus, reference this memory at any time to determine information about your classes, defined types, which classes are inherited from whom, and so on. The big plus in all of this is that previously, information about the syntax of components, or their interfaces, was the only thing stored. Metadata allows the semantics, or meanings, of these interfaces to be stored. By ensuring that this kind of information is made available to the runtime, you can maintain type compatibility between languages, object management, and all aspects for implementing cross-language integration.

Now, all the code that runs within the CLR is called managed code, and it is this concept that allows for true language interoperability. Okay, so what does this all mean already? Take our earlier scenario where a C++ coder wanted to use the capabilities of a pre-existing Visual Basic component. We have noted that it wasn't a simple matter of setting a reference and then being able to use the component. Under .NET, all the languages producing managed code can interoperate with one another. For example, you write a class in Visual Basic that a C++ coder can then inherit in his own class. Say a C# programmer on your development team has created a fully functional assembly, which you need to be able to use on your portion of the project. No problem, simply import the namespaces you want from it and start working with the classes it defines. It's just that simple. In a world where there is growing diversity in technology and growing diversity in technical skill sets, the .NET platform and its CLR create an embodiment of uniformity that allows developers from varying backgrounds to come together and work side by side in a seamless environment.

Managed Code versus Unmanaged Code

By now you have probably seen the term *managed code* thrown around as part of the new .NET lingo. Simply put, managed code is code that runs within the CLR, or rather, is *managed* by the CLR. All languages targeting to run in .NET will produce managed code that will run under control of the CLR. Visual Basic, C++, and C# code will all be managed by one runtime, which does away with the multiple runtimes required previously (depending on which language you developed with). Now you can develop your applications using multiple languages and have to worry only about distributing a single runtime to manage all of that code. This managed code gives the CLR the information it needs to provide many of its core services. When we talk about the benefits of managed code, we are, essentially, also speaking about the benefits of the CLR and the .NET platform as a whole.

So what does this really mean to you as a developer? We still have a runtime, right? Was there anything really wrong with the previous VB runtime (other than it being a bit bloated)? To answer this question, let's look at a fictitious analogy to demonstrate the benefits of having this global commonality. Suppose that a trucking agency delivers large shaped objects in the form of triangles, squares, and rectangles. They have built trailers shaped the same size as each of these large objects for transportation. A triangular trailer ships the triangles, a square trailer ships the squares, and a rectangular trailer ships the rectangles. Imagine that they experience a sudden increase in demand for triangles, but they don't have enough trucks to deliver them. Unfortunately, the triangles do not fit in either the square or rectangular trailers, so the trucking agency is unable to keep up with the demand. Now imagine a new shape, the hexagon, is created. The trucking agency must develop new trailers to accommodate this new object because none of the previous trailers can carry it.

Ok, so what does this have to do with managed code under a common runtime, namely the CLR? The CLR provides a common ground for all languages targeting the .NET platform. It can be thought of as a large spherical trailer that can carry any of the other shapes in our analogy. VB, C++, and C# can all be managed by the same runtime. In fact, any language targeting to run under the .NET platform will be managed by the CLR. A new language can be constructed for inclusion into .NET, and the runtime is unchanged. This gives the platform extensibility that did not exist in previous versions of Visual Studio.

A key to all of this arises from the fact that the CLR provides an Execution Engine that creates what is known as the Virtual Execution System (VES). The VES is essentially what handles and maintains all of this managed code. The VES loads managed code at runtime, and by inspecting the metadata of that code, performs all of the wonderful tasks that make .NET so fun and easy to use. Some of the more important tasks that the VES performs are the following:

- Converting MSIL code into native code
- Garbage collection
- Security services
- Profiling and debugging services
- Thread management and remoting

With the structured, self-describing information stored in the metadata, and an execution system that can use that information, we now have a simple model for language interoperability. All languages that produce managed code can communicate with other languages that also produce managed code. Now, you may be saying, "Yes, but other languages—via binaries such as COM—could already use code in different languages." This is true, but as we have noted previously, the process is neither as simple, nor as intuitive, as it is under .NET. Inheriting from a class written in C# from Visual Basic will appear no different than if the inherited base class had been written in Visual Basic itself.

After all of this talk about managed code, it seems rather intuitive that corresponding unmanaged code must exist. Indeed, it does. Whether you know it or not, you are already very familiar with unmanaged code. All the code you have written prior to .NET is referred to as unmanaged. Essentially, *unmanaged code* is simply a term referring to code that is not managed, and thus, is not targeted to be under the control of the CLR.

As we have mentioned, all code you have written to this point has been unmanaged (in the sense that it is not managed by the CLR, but it is obviously being managed by something). Under .NET, you can't write unmanaged code except with C++. The only type of code you can create with VB.NET and C# is managed code.

Unmanaged code does not benefit from all of the great features that are accessible with managed code. You lose seamless language interoperability, as well as the other features the CLR provides for you under .NET, such as automatic memory management via Garbage Collection (more on this in the Garbage Collection section). But, you have been living without these features for some time now and getting by just fine. Hopefully by now, however, you are beginning to see the true power and potential that the new .NET platform will be offering you as a developer.

You're asking yourself, "What about all of the unmanaged code we developed under COM? Surely we aren't expected to port all of this perfectly functioning code over to .NET to be able to utilize it." Have no fear, your assumption is correct. Microsoft has provided the runtime with a pair of wrapper classes to allow managed and unmanaged entities to work with one another seamlessly. The client will think they are calling the object within their own respective environment regardless of where the object actually originated. Let's take a brief look at how this works.

Interoperability with Managed Code

Managed code and unmanaged code may want to communicate with each other in two different ways. For example, either a .NET client will want to call upon a method of a COM object, or a COM client will want to call upon a method of a .NET object. Each case uses a different wrapper class to handle the communication between the managed and unmanaged objects. These are referred to in .NET as COM Interop wrappers.

When a .NET client wants to talk to a COM object, the runtime creates a Runtime Callable Wrapper (RCW). The RCW is responsible for resolving the differences between the managed .NET client and the unmanaged COM server. The RCW acts as a proxy that marshals calls between the client and server. In essence, the RCW is a translator, which can convert method calls and return types unfamiliar to the client or server into something they can understand. For example, the .NET client may send across a value of type String (which is now an Object in .NET, as is everything else), which the RCW would interpret and translate into a BSTR type for the COM server to understand, and vice versa.

When the roles are reversed, and a COM client wants to call a .NET server object, then the runtime creates a COM Callable Wrapper (CCW). The CCW functions similarly to the RCW except that it marshals method calls and values going the other direction. You can think of a RCW as a Spanish to English translator whereas a CCW is a English to Spanish translator. They perform complementary roles to ensure proper communication between two entities in both directions: sending and receiving.

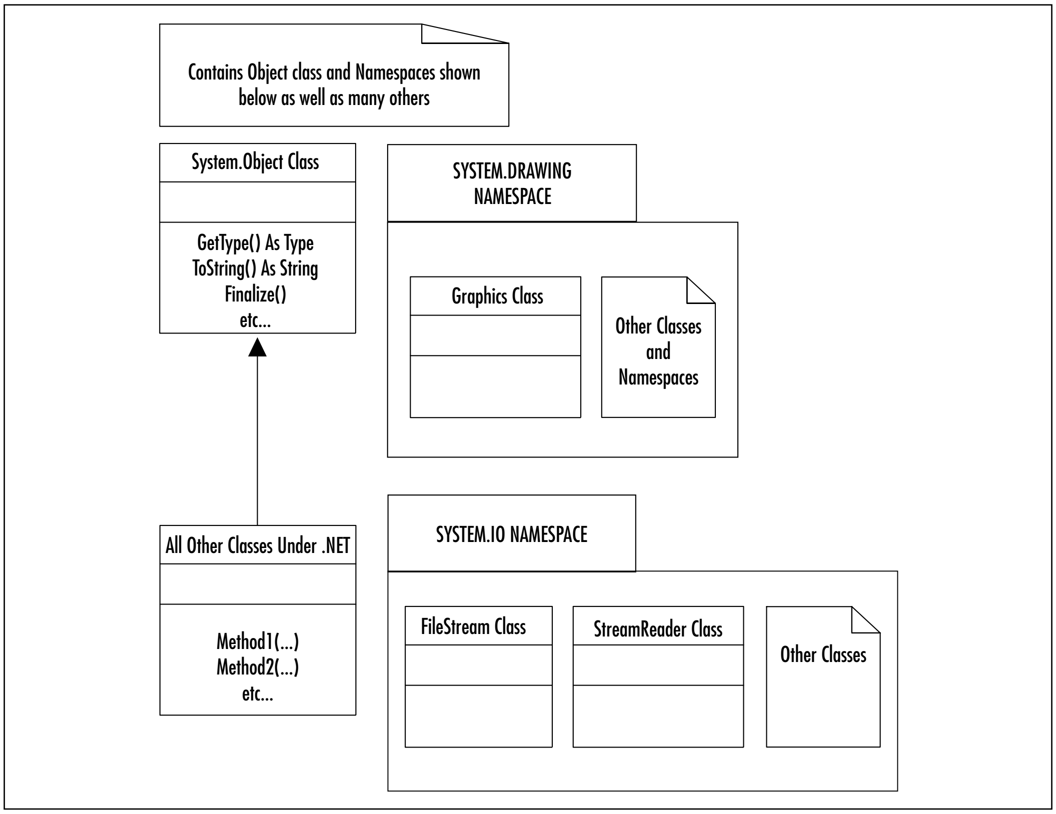
In addition, Microsoft's push to COM+ since the release of Windows 2000 was a foreshadowing of their .NET initiative. COM+ was created with .NET in mind (or maybe it was the other way around). At any rate, COM+ will still provide the services we have come to depend on for providing things such as transaction support, object pooling, and queuing. Managed components and unmanaged components will both be able to happily coexist under the services provided by COM+, and they will have the capability to communicate seamlessly via the automatic creation and implementation of the callable wrappers.

System Namespace

Although all of that general technical stuff discussed previously is important, most developers are interested in the language itself. We keep referring to all of these new features and benefits that VB.NET brings to us, so let's take a look at some of these features and begin to familiarize ourselves with the changes the new platform introduces into our programs code-wise.

It has been stated before and we state it again here: *Everything* in VB.NET is an Object. For anyone who has ever had some exposure to Java, the concepts presented herein will strike a nerve of familiarity. In VB.NET, Microsoft has introduced the idea of namespaces. Namespaces organize all of the objects that are defined within an assembly. The assembly can contain multiple namespaces, which in turn can contain other namespaces. Under VB.NET, all objects derive from the System namespace (see Figure 4.2). The System namespace contains all the fundamental classes that define most of the common data types, events, interfaces, and exceptions. All other objects derive from the *System.Object* class, which implements all of these base features. Any classes you write will also depend on the *System.Object* class, which you will extend to provide the functionality you want your class to offer. This results in a hierarchical structure of object inheritance that clearly defines the true object-oriented nature of the .NET platform. Another huge bonus is the fact that a lot of those hard-to-use, messy API calls have been replaced with more intuitive objects complete with properties and method calls. Let's take a look at some common subcomponents of the System namespace that you may be using when writing code under the .NET platform and see how this new framework replaces the independent functions you are accustomed to working with.

Figure 4.2 System Namespace



File I/O

Probably one of the most familiar tasks for Visual Basic developers is simple file I/O. Reading and writing to a file is something we have all done a lot of for a multitude of purposes. Under VB.NET, file I/O is encapsulated in the System.IO namespace. Let's take a look at a simple example of reading from a file with VB.NET. To make the example clearer, we first look at the code in current versions of Visual Basic, and then we compare it to how it might look under .NET. This example simply opens a file for reading, reads in a single line, and displays it in a message box to the user:

```
Sub Main()
    Dim sBuff As String

    Open "C:\Temp\Sample.txt" For Input As #1
        Line Input #1, sBuff
        MsgBox sBuff
    Close #1
End Sub
```

CD File 4-1

Simple enough. Now let's take a look at its VB.NET equivalent:

```
Imports System.IO

Module TestMod
    Sub Main()
        Dim oFile As FileStream = New FileStream _
            ("C:\Temp\Sample.txt", FileMode.Open, FileAccess.Read)

            Dim oStream As StreamReader = New StreamReader(oFile)
            MsgBox(oStream.ReadLine)

            oFile.Close()
            oStream.Close()
    End Sub
End Module
```

The first order of business here is the importing of the System.IO namespace. VB.NET uses the Imports statement to include namespaces that contain classes you will want to utilize in your code. Within the System.IO namespace are the two classes we are using in the example, namely FileStream and StreamReader. The *FileStream* class allows us to open a stream to a file, and the *StreamReader* class allows us to read from that stream. Notice the capability to declare and initialize an object using its constructor all on the same line. This is another wonderful convenience introduced into Visual Basic under .NET. This example should give you confidence knowing that the logic necessary to work with files has not changed, only its syntax has, which you will become familiar with once you get started coding with it on a regular basis.

Drawing

Anyone who has ever done heavy, intensive graphics work with the Graphics Design Interface (GDI) Windows API in current versions of Visual Basic knows that it is not the most fun, nor the easiest, code to work with and manage. This is a perfect example of where the .NET platform has done a terrific job of encapsulating difficult to learn/read/use API functions into more comprehensive, reusable objects. Classes for working with graphics have been included in the System.Drawing namespace. The new functions provided in this namespace have often been referred to as the GDI+ functions. The *Graphics* class in the System.Drawing namespace is where most of the magic happens. For example, in a simple WinForms application, you could draw a blue line on your form using code like this:

CD File 4-2

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

    Dim G As Graphics = Me.CreateGraphics()
    G.DrawLine(New Pen(Color.Blue), New Point(10, 10),
    New Point(50, 50))
End Sub
```

Or you could draw a curved red line connecting several points:

CD File 4-3

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

    Dim G As Graphics = Me.CreateGraphics()
    Dim pts(3) As Point

    pts(0).X = 10
    pts(0).Y = 10
    pts(1).X = 40
    pts(1).Y = 40
    pts(2).X = 70
    pts(2).Y = 50
    G.DrawCurve(New Pen(Color.Red), pts)
End Sub
```

Working with graphics is just that easy. Although you have a lot of new syntax changes to digest, they all share the common property of working with objects. This sort of commonality will only speed the process of familiarizing yourself with the new look and feel of Visual Basic.

Printing

The familiar Printer object has been replaced by the System.Drawing.Printing namespace. Recall earlier we mentioned that a namespace can contain other namespaces. Well, here is an example. The System.Drawing.Printing namespace is contained within the System.Drawing namespace, which we discussed in the preceding section. Here is a small example showing how you might print a simple text string to the printer:

CD File 4-4

```
Imports System.Drawing
Imports System.Drawing.Printing

Module TestMod
    Private WithEvents oPrint As PrintDocument

    Sub Main()
        Try
            'instantiate PrintDocument object and call Print method
            oPrint = New PrintDocument()
            oPrint.Print()
        Catch e As Exception
            MsgBox("Error printing file: " & e.ToString)
        End Try
    End Sub



    Public Sub oPrint_PrintPage(ByVal sender As Object, _
        ByVal e As System.Drawing.Printing.PrintPageEventArgs) _
        Handles oPrint.PrintPage

        'this event fires for each page printed
        'you will handle all printing logic here
        Try
            e.Graphics.DrawString("Print test", New Font("Arial", 10), _
                New SolidBrush(Color.Black), 100, 100)
        Catch ex As Exception
            MsgBox("Error: " & e.ToString)
        End Try
    End Sub
End Module
```

In this example, you are also getting a glimpse of structured error handling provided in VB.NET. You might think that this is a lot of code to simply output a single line to the printer, but the extra work required here is well worth the benefits gained when you move to implementing more complex printing schemes. Anyone who has ever struggled with providing rich and powerful printing capabilities for their applications is going to love the functionality included within this namespace. Carried over from the Printer object are the basic properties for setting paper type, paper size, and so on. Two exciting new features are the capability to receive events from your print job, such as when the printing begins and ends, and to provide full print preview capabilities.

Common Type System

We have already talked a bit about CLR's cross-language integration capabilities. Several parts help make up this seamless environment. One of the most important parts, if not the most important, is the Common Type System (CTS). The entire .NET framework is built on this type system that the CLR defines. We take a moment here to discuss why this component of the CLR is so important for ensuring proper language interoperability.

We briefly mentioned the CTS—also referred to as the Universal Type System—earlier when we discussed the capability of interoperability between managed and unmanaged code. The COM Callable Wrapper (CCW) was responsible for ensuring that types transferred by the COM object be translated into a type supported by the CLR. The CTS describes these types supported by the runtime, and how those types can interact with one another. Simply put, a type is a semantic definition describing an entity that can accept certain values and certain operations on those values.

Previously, each language had its own definition of types it supported, which was often not consistent across separate languages. For example, if you have ever taken a C++ DLL function declaration and tried to figure out how to port the types in its formal parameter definitions to those types supported by Visual Basic, you will have a good appreciation of what a convenience a common type system will provide. The CTS shines through in its definition of rules it places on language compilers targeting the .NET platform. It provides rules that the compilers must follow with respect to defining, referencing, using, and storing types.

Language compilers have always been responsible for maintaining information about types of variables used throughout a program and storing information about those types to perform a certain amount of type checking at compilation time. In addition, the language defines sets of rules pertaining to what operations are allowed on particular types of data. For example, assume that you had the following code fragment in your current Visual Basic application:

```
Dim i As Integer
Dim s As String

i = 1
s = "Hello"
Debug.Print i + s
```

This will result in a run-time error that pops up informing you of a "Type Mismatch." This makes sense when you look at the operation the code is attempting to perform. Logically, one would imagine that attempting to add an integer to a string would cause an error. Remember, though, that the semantic rules applied to these types are what determine whether this would raise an error or not. However, let's look at the following C code that attempts a similar operation:

```
int i = 1;
char s[5] = "Hello";
printf("%d", i + s);
```

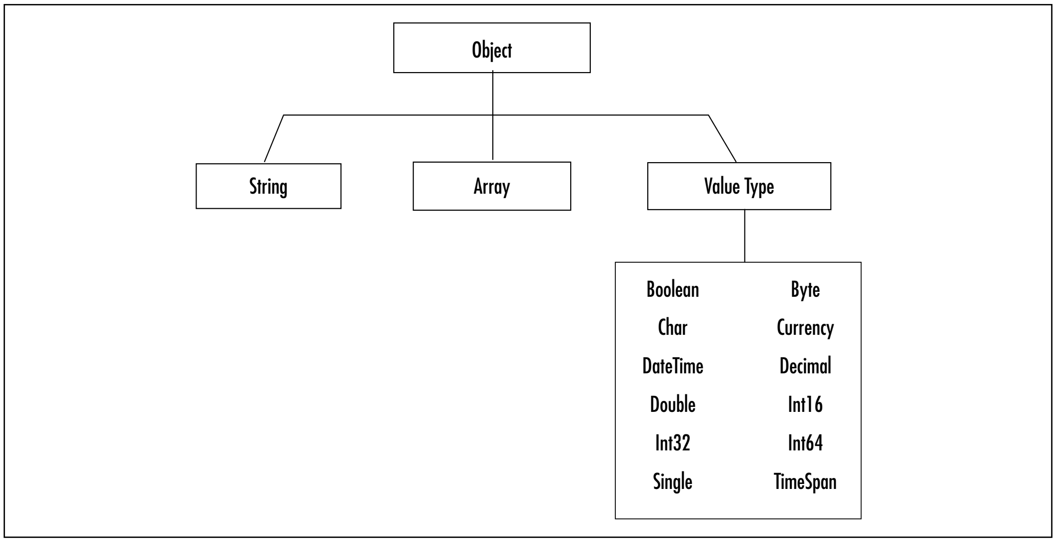
Running this code will not raise an error at all. It outputs a large number and continues on without a glitch. This is because C handles strings differently than Visual Basic does. C really has no concept of the string data type, and it implements them as character arrays. Because arrays are treated as pointers in C, it simply adds the integer value 1 to the number value of the array's location in memory (you can show this by outputting the value of &s as well). These two languages clearly exhibit a difference in how they define encapsulating a string into a type. To imagine these two languages coexisting with one another would not be very feasible. Not only does one language exhibit a type that the other does not, but their implementations of that particular type of data is also completely different. We can not have this kind of inconsistency if we want to achieve efficient cross-language integration, and the CTS ensures that these sort of infractions would not occur.

The CTS defines the rules that the language compilers must abide by to ensure strong type standardization. The fact that each language compiler must treat types in a consistent manner is the basis on which the CTS exists and allows for objects created in the different languages to correctly interact with one another. The .NET platform provides a programming model that is based on the CTS (see Figure 4.3). As mentioned previously, everything derives from the *System.Object* class. The *String* and *Array* classes are direct descendants thereof, and the other familiar primitive datatypes reside within, or inherit from, the *ValueType* class.

Type Casting

Type casting, or the capability to change a variable from one type to another, is a common practice in any programming language. The capability to convert values from one type to another is essential to the usefulness and power of a programming language. Type casting is similar to what it has always been in Visual Basic. At times, you may be getting data from somewhere in one data type, but you need to use it in a different context from within your code. A perfect example of this is grabbing data from a text file. This data may be a set of delimited numeric values that you need to perform some computations on. When you read the data in, it will more than likely be read into a string that must then be converted to a usable number.

Figure 4.3 .NET Type System



VB.NET still supports all of the explicit type conversion functions you are accustomed to, such as CStr, CDbl, CSng, and so on. However, note that all of the primitive data types from previous versions of Visual Basic are now encapsulated into objects with constructors and all (remember, *everything* is an object). These objects provide their own internal mechanisms for providing type casting and offer more capabilities than the standard type conversion functions you are accustomed to. Again, those with any exposure to Java will feel right at home seeing how this is implemented. For example, in current versions of Visual Basic, we may take a string and wish to store it into an integer:

```
Dim i As Integer
Dim s As String

s = "12345"
i = CInt(s)
MsgBox i
```

We could perform the exact same operation in VB.NET, but let's begin to familiarize ourselves with our new OO language and its syntax:

```
Dim i As Integer
Dim s As String

s = "12345"
i = s.ToInt32
MsgBox(i)
```

Notice that everything here is left the same except for how we perform the type conversion and the parentheses around the parameter in the call to our MsgBox function.

Here, we are using the String object's intrinsic type conversion method, ToInt32, to perform the actual conversion to an Integer (note that Integer's are now 32 bits in VB.NET). However, let's see how we can wrap up that code into a more appealing VB.NET OO compliant syntax:

```
Dim i As Integer = New String("12345").ToInt32
MsgBox(i)
```

This snippet is not a very practical one, but it offers you an idea of what working with the new syntax of VB.NET will be like. The main focus of this section was to comfort you in knowing that the type conversion functions you have become accustomed to are still there if you want to use them. Hopefully, however, you will try to quickly adopt the intrinsic type conversion functions of the data types themselves as your preferred method.

Developing & Deploying…

Embrace Your Parameters

VB.NET is insistent upon enclosing parameters of function calls within parentheses regardless of whether we are returning a value or whether we are using the Call statement. It makes the code much more readable and is a new standard for VB programmers that is consistent with the standard that nearly all other languages adopted long ago.

Garbage Collection

Another huge transition in how you will code is the inclusion of Garbage Collection under the CLR. Essentially, the CLR Garbage Collector monitors a program's resources looking for objects no longer in use when the available resources are reduced to a certain limit. It then frees the memory of these unused objects to allocate memory for other objects and tasks that will need it. This is quite a change from the reference counting scheme implemented in COM. Given this basic idea of what Garbage Collector is doing for us, let's talk in a bit more detail on how it all works and about the pros and woes you might face with this new automatic memory management implementation.

The decision to move to automatic memory management did not come about without a lot of heated debates. Many hardcore COM developers were insistent upon maintaining the reference counting scheme. The most influential reasons behind the move to implementing Garbage Collection was that it increased performance, eliminated common reference counting errors resulting from misuse, and did away with the circular reference problem. Some of these issues you may feel have little relevance to you. Some of these issues you may have never faced nor feel that you will ever encounter. So why this drastic change in the way things work? Remember that one of the main features of the .NET platform is language interoperability. By moving the task of memory management to the runtime to handle, we remove inconsistencies and errors that can be introduced by the programmer. This allows programmers of different languages under the .NET platform to concentrate on implementing objects with rich functionality, without having to worry about implementing a scheme to manage those objects. Oh, and of course, we do away with the circular references problem. To better understand why this is so, we must look at how Garbage Collection works under the CLR.

The CLR requires that all resources be allocated from the managed heap. Unlike previous runtimes when you had to free objects from the heap explicitly, Garbage Collector in the CLR does all of this for you automatically. It does this via a rather complex algorithm.

Essentially, Garbage Collector provides a mechanism by which it can determine which objects are still being used and which are not. Those that are no longer in use get collected. To help improve performance, Garbage Collector implements generational garbage collecting. Generally speaking, a generation simply categorizes objects on the heap into sections called *generations*. The idea behind this is that the longer an object *stays alive*, the older the generation it will exist in. Research and experience shows that this is the usual trend. If an object is relatively new, we assume it will have a short life span (no guarantee we will use it for very long). However, if we see an object that is relatively old, for example, it has already survived several collections, we may assume it will continue to last even longer. Currently, Garbage Collector under the CLR supports only three levels of generations: 0, 1, and 2. Generation 0 is where new objects are placed, generation 1 is for those that have survived a single collection, and generation 2 is for those that have survived two or more collections. How does this improve performance? Well, Garbage Collector performs a collection when generation 0 becomes full. It can then decide if it should perform a collection on all the objects on the heap or just the newer ones located within generation 0. For applications that may contain many objects in generations 1 and 2, this can greatly reduce the overhead encountered during a collection.

Developing & Deploying…

Collection of What?

In .NET, the term "collection" is often used to refer to the garbage collecting mechanism. Of course, the Collection object we are familiar with still exists, as well as the introduction of a Collections namespace that introduces some useful data structures for you to use. Just remember to take into account the context under which the term "collection" is used, and you will be fine.

Object Allocation/Deallocation

When a process is first initialized, the CLR reserves some contiguous space in memory for the process, which has no storage allocated to it. This is the managed heap. As objects are created via the New keyword, they are placed onto the heap. This process continues until there is not enough memory left on the heap to allocate memory for the next object requesting resources. At this point, a collection must be performed. Garbage Collector applies its algorithm for determining which objects are no longer in use on the managed heap and disposes of them accordingly.

In previous versions of Visual Basic, because we were in control of destroying objects that we created, object deallocation was clearly defined. When writing classes, you could write code in the Terminate event of the class and feel comfortable knowing that after the object was destroyed, the code in that event would fire immediately. This is known as *deterministic finalization*. With the introduction of Garbage Collector, this is no longer the case. Now we are dealing with *non-deterministic finalization*—we can not predetermine the exact time when the finalization for an object will occur. This probably raises a few brows. Many developers have come to rely upon the Terminate event to perform other maintenance or cleanup routines within their applications. With the introduction of Garbage Collector, the Terminate event has been replaced by the Finalize event. The Finalize event does not offer the same functionality as the Terminate event for reasons we have just talked about. Though both events fire when the object is released from memory, we lose the deterministic characteristics that the Terminate event and COM reference counting offered.

Now the main recommendation is to develop objects that do not require any sort of cleanup. This is a nice consideration but it just isn't possible in a lot of cases. For example, you may create a class that holds an open, locking reference to some sort of data file, and in the case of your object being terminated, you want to ensure you release this hold on the file that you had. Previously, you would just write the code necessary to release this resource in your class Terminate event. Now, you may be tempted to do the same in the Finalize event, but in a time critical application, where perhaps another object will be instantiated to use this file before any collection is performed, you need a way to release this resource much more quickly. Well, we all need to start practicing a new standard of coding when it comes to situations like these. You will want to implement a Close or Dispose method in your class that the client will call explicitly to instruct your object to perform the cleanup operations required. You are not limited to naming your cleanup routines either Close or Dispose, but this is the convention Microsoft would like for you to use.

Close/Dispose

So how should these methods be implemented, exactly? Sticking to the recommended conventions, you should use Close if you want to allow the object to be able to be reopened or reused again in future operations. Continuing with the earlier example, the Close method may close the file that your object had a lock on, but remain "alive" for future use. On the other hand, the Dispose method would be called to completely destroy your object. This is synonymous to setting an object to Nothing in current versions of Visual Basic.

Summary

In this chapter, we took an overall look at the new features offered by the .NET platform, mainly through its inclusion of a common runtime. The idea of managed code seems so straightforward and logical that it's a wonder we haven't stumbled across an architecture like this sooner. Regardless of that perception, the day has arrived, and developers across the globe can begin to prepare for, and take full advantage of, this new realm of coding and design that lies ahead.

We discussed the benefits of managed code by the CLR throughout the chapter, and saw how it will change the way you program. Automatic memory management, self-describing components, and true language integration pave the way for a more scalable, maintainable future in development. We also covered, in brief, what will become of the COM era with this new birth of .NET. We made it quite clear, or at least we hope, that COM is far from dead, and the team at Microsoft, we believe, is aware of that fact. Thus, you know that you have some means of "backwards compatibility" with regards to some of your pre-existing components, if and when you make the full transition over to .NET.

We noted that everything derives from the *System.Object* class. We saw how even our most primitive data types have been included in this true hierarchical framework. A truly object-oriented language has been born with the release of VB.NET, and we highly recommend that you embrace it as soon as you have the opportunity. Becoming familiar with OO concepts and principles now has real purpose and relevance to you as a Visual Basic developer. The syntax may take a bit of time to become accustomed to, but hopefully by now you realize that once you do become familiar with it, it will only speed up and ease your work and improve your productivity.

To sum up, we have seen how the different components comprising this runtime work together to provide a common type system and self-describing components that allow for the runtime to offer us great new benefits, such as cross-language integration and garbage collecting. Understanding how each of these services work is important. The concepts of namespaces and providing a truly hierarchical inheritance framework gives us a better understanding of why Visual Basic had to become more object-oriented, and gives us a better understanding of the benefits and scalability this sort of framework offers to the platform as a whole.

Solutions Fast Track

Component Architecture

☑ Self-describing components allow for easy maintainability and deployment.

☑ Components developed under .NET will still be able to utilize the benefits of the transactional services provided by COM+.

Managed Code versus Unmanaged Code

☑ Managed code provides the information necessary for the CLR to provide numerous services.

☑ Managed code allows all languages that conform to the standards required to run under the .NET platform to use a single shared runtime.

☑ Unmanaged code does not benefit from all of the things that managed code benefits from.

☑ Microsoft has provided a means by which managed code and unmanaged code can communicate, thus allowing integration of new, powerful .NET applications with legacy components.

System Namespace

☑ Almost all system functionality has been wrapped up within a single entity called the System namespace.

☑ The System namespace provides the *Object* class from which all other classes derive.

☑ The System namespace contains all the other classes and namespaces that provide a means for developers to utilize objects to carry out almost any task.

Common Type System

☑ All languages producing managed code abide by a strong type standardization.

☑ Use of types across different languages will no longer cause any headaches.

Garbage Collection

☑ The CLR performs automatic memory management in your applications.

☑ COM reference counting has been replaced by the Garbage Collection algorithm.

☑ Deterministic finalization is lost, and you must employ new methods to code around this issue.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: Sometimes I may want to force a collection to occur. Can I do this or must I rely on the system invoking a collection only when the managed heap becomes full?

A: Yes you do have some control over Garbage Collection via the *System.GC* class. To tell the runtime to perform a collection, you simply invoke its Collect method:

```
GC.Collect()
```

This is an overloaded method in which you can also specify, as a parameter, which generation you wish to be collected (either 0, 1, or 2).

Q: Okay, I can live with implementing a Close/Dispose method for my objects, but I also want to implement a Finalize method to perform cleanup in case the Close/Dispose method is not properly called. If the Close/Dispose method is properly called, how do I ensure that the Finalize method does not also try to execute the code already run in the Close/Dispose method?

A: Well, you might think of implementing a flag variable that you set in your Close/Dispose method and then check in the Finalize method to determine whether the code needs to be run in the Finalize method when your object is finally collected. However, a much more intuitive and efficient solution exists. The *System.GC* class provides a method, SuppressFinalize, which takes as a parameter an object and instructs the runtime not to call the object's Finalize method. If you implement something like the following in your object, you can ensure proper behavior:

```
Public Sub Dispose()
    'perform clean up on this object here

    'call method to suppress the finalize method
    GC.SuppressFinalize(Me)
End Sub

Protected Overrides Sub Finalize()
    'if Dispose was not called, call it now
    Dispose()
End Sub
```

This is an improvement over the idea of simply using a variable flag in your code. One reason is that it reads better. But more importantly, this method implements something like setting a flag, but at the system level. It ensures that the object is not placed in the queue of objects that need to be finalized, and thus, the Garbage Collector will not have another object to worry about, and performance will be improved.

Q: Can I monitor the CLR's activities/performance in real time?

A: Yes, you can achieve this by running Perfmon.exe in Windows2000, clicking the "+" icon on the toolbar, and selecting which object of the CLR you wish to monitor.

Q: We have a lot of code in place under COM that we will want to utilize even after we have moved to .NET. Should we port this code over or should we simply use the COM Interop services to communicate with our unmanaged objects?

A: This is a delicate issue and will take some serious consideration. What to do will most likely vary from situation to situation. Things you may want to consider are the following:

- How long can you afford to use these legacy objects? Will they need to be ported eventually? If so, you may want to go ahead and consider the transition process sooner than you had anticipated.
- Is performance a factor? If yes, then you will want to port over for a couple of reasons. One, your existing objects are communicating through wrapper classes. This is introducing another level of indirection and work that must be done by the Interop service through the wrapper class. This, in and of itself, will cause a hit on performance. Secondly, by porting your code over to run under the CLR, you get to take advantage of all the benefits of managed code we have discussed herein.
- Do the objects function correctly via the COM Interop service? The wrapper classes have yet to be heavily utilized in industry at this point, and some inconsistencies may exist between running your COM objects directly as opposed to through the COM Interop service. The wrapper classes are customizable, however, and you may find solutions that way. Again, though, you will need to make considerations based on cost of learning and implementing fixes (workarounds) for legacy code versus the cost of porting the code to target the CLR.

Chapter 5

.NET Programming Fundamentals

Solutions in this chapter:

- Variables
- Constants
- Structures
- Program Flow Control
- Arrays
- Functions
- Object Oriented Programming
- String Handling
- Error Handling

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

Even though this book is focused on the intermediate programmer, some fundamental programming is included. This can be used as a refresher, or for those of you from different programming languages, it will provide you the syntax for some fundamental programming. This chapter is not intended to teach beginners how to program. We cover how variables are declared and used. Variable types have changed since Visual Basic 6.0. You must be more specific with data types now. You cannot count on Visual Basic to automatically convert everything for you. Also new to Visual Basic .NET are structures. If you have programmed in C, this will be familiar. This replaces the Type in previous versions of Visual Basic. Structures allow you to logically group together variables of different (or same) data types. Each member of a structure is given a name. It allows you to utilize a group of data as a single unit with access to its members by name.

When developing applications, you have to be able to dynamically set the flow of a program's execution. There are several programming fundamentals to allow you to control the flow of execution. This chapter shows you the syntax and usage for decision making and looping. Arrays allow you to store data that can be accessed by indexes rather than names. Arrays have changed somewhat from previous versions, and it is important to understand these differences. Functions allow you to separate code into units of functionality. There are many benefits to functions when developing applications. Working with strings can sometimes be confusing. There are numerous functions available for manipulating strings. We look at some of these functions and how to use them.

Visual Basic .NET is now arguably a true object oriented programming language. Everything is an object. It is important to understand what object oriented programming is and to shift your thought process. A brief overview of what constitutes object oriented programming—and how Visual Basic satisfies it—is included in this chapter. You can now create and inherit classes. You can even inherit classes written in other programming languages, which is a powerful new feature. Visual Basic classes are now more like C++ classes. You don't have to create class modules anymore to define a class.

A major paradigm shift in Visual Basic .NET is error handling. After years of begging by programmers, Visual Basic now uses structured error handling. Error handling now shifts to the use of exceptions. This is similar to other programming languages. This will allow you to have more robust error handling with better control and more comprehensive handling of errors than previous versions.

You will have to learn some new concepts, but this will empower you to develop applications that are more responsive to errors.

Variables

A *variable* is simply a named memory location. When writing programming code, you'll find numerous situations where you need to store data It could be for temporary values during calculations, information on a customer, and so on. This data is stored in memory. Instead of referring to the memory location by actual memory address, you can give it a name by which you can refer to it. You give it a name by declaring a variable and giving it a data type. When naming a variable, you must follow a few rules:

- It must start with an alphabetic character.
- It can only contain alphabetic characters, numbers, and underscores.
- It cannot contain a period.
- It can not be more than 255 characters.
- It must be unique within the current scope (we discuss scope shortly).

The data type determines how much memory is allocated to store data in. Visual Basic .NET has a number of built-in data types that are specified in the Common Language Runtime. Later, we will see how to create our own custom data types. For those of you who have used the previous versions of Visual Basic, you know that if you don't give it a variable type, it is implicitly assigns it as a Variant data type. In .NET, the Variant data type no longer exists. In Visual Basic .NET, if you don't specify a data type, it defaults to an Object data type. You should always specify the data type for a variable. When you declare the data type for your variables, this is called *strong typing.* Using the type of variable with the smallest size that will meet your needs is good programming practice. For instance, if you are going to add integers that will always be less than 100, using an Integer would be a waste of memory when a Byte would suffice. However, ensure that the variable is large enough for all circumstances. Also, by using strong typing, you will be able to use Intellisense for your variables, the compiler can perform type checking to help reduce the possibility for runtime errors, and your code will execute faster because it doesn't have to implicitly determine the data type. In Table 5.1, we take a look at the data types in Visual Basic .NET, including their size and range.

Table 5.1 Comparing Built-In Data Types

VB.NET Type	Size	Range
Boolean	4 Bytes	True or False
Byte	1 Byte	0–255 unsigned
Char	2 Bytes	0–65,535 unsigned
Date	8 Bytes	1/1/1 CE to 12/31/9999
Decimal	12 Bytes	+/–79,228,162,514,264,337,593,543,950,335 with no decimal point; +/–7.9228162514264337593543950335 with 28 places to the right of the decimal; smallest nonzero number is +/–0.0000000000000000000000000001
Double	8 Bytes	–1.79769313486231E308 to –4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values
Integer	4 Bytes	–2,147,483,648 to 2,147,483,647
Long	8 Bytes	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
Object	4 Bytes	Any object type
Short	2 Bytes	–32,768 to 32,767
Single	4 Bytes	–3.402823E38 to –1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values
String	10 Bytes + (characters in string * 2)	0 to approximately 2 billion Unicode characters
User-Defined Type	Sum of the size of its members	Range dependent data type for each member

We have discussed how to name variables and the data types available. Now let's see how to declare variables. As in previous versions of Visual Basic, you use the **Dim** keyword. The following are some common examples of declaring variables:

```
Dim x as Integer
Dim y, z as Single
Dim str as string
Dim obj  'defaults to Object
```

As in previous versions of Visual Basic, you can also specify the data type of a variable by using *identifier type characters*. This is done by appending a type character to the end of a variable name to specify its type. These identifiers can also be used with constants and expressions to explicitly declare their data type. Here are some examples of using the identifier type characters:

```
Dim intX%     ' % character identifies it as an Integer data type
Dim lngX&     ' & character identifies it as a Long data type
Dim decX@     ' @ character identifies it as a Decimal data type
Dim sngX!     ' ! character identifies it as a Single data type
Dim dblX#     ' # character identifies it as a Double data type
Dim strX$     ' $ character identifies it as a String data type
```

New to Visual Basic is the capability to initialize variables when you declare them, which is a feature that C++ programmers are accustomed to. By default, numeric variables are initialized to zero, a string is initialized to an empty string (""), and an object variable is initialized to Nothing. If you want to initialize a variable to a value other than the default, you can now do it on the same line that you declare it. You can also initialize Object data types, but we take a look at that later in the chapter. Here are some common examples of initializing variables when declaring them:

```
Dim x as Integer = 5
Dim dblValue as Double = 22.5
```

Constants

Constants are similar to variables. The main difference is that the value contained in memory cannot be changed once the constant is declared. When you declare a constant, the value for the constant is specified. So, why bother with a constant? Why not just use the value in your code? Because using constants rather than hard-coded values is good programming practice. A common illustration for the use of constants is the use of rates. Say you are developing an application that uses a special internal company factor when determining prices for products. You might use this factor in numerous places throughout your code. If you used hard-coded values, and the factor changed, you would have to search the code and change the value everywhere it was used. If you were using a constant instead, all you would have to change is the value of the constant—this would automatically propagate the change to your entire application. Here are some examples of declaring constants:

```
Const X As Integer = 5
Const str As String = "Company Name"
Const X As Double = 0.12
```

Structures

A *structure* allows you to create your own custom data types. A structure contains one or more members that can be of the same or different data types. Each member in a structure has a name, which allows you to reference the members individually by name even though the structure as a whole is considered a single entity. In previous versions of Visual Basic, structures were implemented using the **Type** keyword. In Visual Basic .NET, the **Structure** keyword is used to define a structure. This is the syntax for structures:

```
[Public|Private|Friend] Structure varname
NonMethod Declarations
Method Declarations
End Structure
```

The following code examples show how structures were declared in Visual Basic 6.0 and now in Visual Basic .NET. Notice how a scope now has to be given to the members. In the examples, we are just using the **Dim** statement. Basically, scope determines where in the program's code a variable can be accessed. The Visual Basic 6.0 example is as follows:

```
Type Employee
  No As Long
  Name As String
  Address As String
  Title As String
End Type  'Employee
```

In Visual Basic .NET, structures are declared like this:

```
Structure Employee
  Dim No As Long
  Dim Name As String
  Dim Address As String
  Dim Title As String
End Structure 'Employee
```

Now let's take a look at how structures are used programmatically. The example is for employee data. The data for an employee is a single entity, but you would like to reference the members individually. If you did this with individual variables, controlling the variable for each employee could become difficult. In the following example, we can easily reference the data for different employees:

```
Dim emp1 As Employee
Dim emp2 As Employee

emp1.No = 12345
emp1.Name = "Cameron Wakefield"
emp1.Address = "123 Somewhere Ave."
emp1.Title = "President"

emp2.No = 12346
emp2.Name = "Lorraine Wakefield"
emp2.Address = "123 Somewhere Ave."
emp2.Title = "Vice President"
```

NOTE

Structures are now very similar to classes. Structures can even contain methods as shown in the structure syntax. There are some restrictions on structures that are not limited in classes. For instance, you cannot inherit a structure. Also, you cannot initialize structure members, and you cannot use the **As New** statement when declaring the members. Structures are referenced by value, not by reference. So if a structure variable is assigned to another variable, the structure would be copied to the new variable. This also applies to passing it to a function by value.

Program Flow Control

In this section, we cover how to control the flow of execution of your program code. When code is executed, certain blocks of code might only need to execute for certain conditions. For this type of execution control, we discuss the use of the **If…Then…Else** and **Select** statements. In other circumstances, you may want code to execute multiple times until a certain condition occurs. You can accomplish this with **While** loops. A similar case is when you want a loop to execute a certain number of times—this case uses the **For** loop.

If…Then…Else

If…Then…Else statements allow you to determine which block of code is executed based on specified criteria. For the block of code to execute, the condition must evaluate to True. Let's take a look at the syntax for these statements, which take on two basic forms: the single-line form and the block form. The single-line form allows you to put the statement all on one line, as follows:

```
If condition Then [statement] [Else elsestatement]
```

The block form breaks the statement up over multiple lines. This format is more structured and easier to read and follow. The syntax for the block form is shown in the following code. The brackets around the **ElseIf** and **Else** statements indicate that they are optional:

```
If condition Then
  [statements]
[ElseIf condition-n Then
  [elseifstatements] ...
[Else
  [elsestatements]]
End If
```

When implementing **If…Then…Else** statements, you must always include the **If…Then** statement and provide a condition. If this condition is true, the block of code will execute. This block is terminated by one of the following statements: **ElseIf**, **Else**, or **End If**. If the condition is not true, it will see if any **ElseIf** conditions are true. The **ElseIf** statement allows you to enter multiple additional condition execution blocks. The blocks will execute if its condition is true. However, only one block will execute. Once a block of code is executed, execution move to the end of the **If…Then…Else** statement. It doesn't execute every True condition, which means that you should be careful about the order of the blocks. The **Else** statement does not have a condition; this block will execute if none of the blocks above it are true. The **Else** block is always the last block in an **If…Then…Else** statement. You can have only one **Else** statement.

Let's take a look at an example. This example determines the shipping cost based on amount of purchase. Notice that the **Else** clause is executed when the purchase amount is greater than all of the specified amounts. Notice the order of the conditions. If we had put the condition for less than the third shipping class, the code would never get to the blocks for the first and second shipping classes:

```
Const MIN_AMT As Single = 9.99
Const SECOND_AMT As Single = 29.99
Const THIRD_AMT As Single = 49.99
Const MIN_SHIP_COST As Single = 10.49
Const SECOND_SHIP_COST As Single = 21.5
Const THIRD_SHIP_COST As Single = 26.33
Const MAX_SHIP_COST As Single = 30.48

Dim sngShipCost As Single

If sngAmt <= MIN_AMT Then
    sngShipCost = MIN_SHIP_COST
ElseIf sngAmt <= SECOND_AMT Then
    sngShipCost = SECOND_SHIP_COST
ElseIf sngAmt <= THIRD_AMT Then
    sngShipCost = THIRD_SHIP_COST
Else
    sngShipCost = MAX_SHIP_COST
End If
```

Notice that the **If…Then…Else** statements use less-than or equal comparison operators. Visual Basic provides a number of comparison operators (shown in Table 5.2) to allow extensive comparisons of expressions.

Table 5.2 Comparison Operators

Comparison Operator	Comparison Type
=	Equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
<>	Not equal to

In addition to the comparison operators listed in Table 5.2, you can use two additional operators for special purposes: the **Is** and the **Like** operators. The **Is** operator is used to compare object references. If two object references point to the same object, the comparison is True. The **Like** operator is used to compare a string to a string pattern rather than the exact copy of a string. This is similar to the SQL **Like** clause. Wildcards give you flexibility in the pattern matching. Table 5.3 lists the wildcards that are available for the **Like** operator.

Table 5.3 Like Operator Wildcards

Wildcard Character	Pattern Matches
?	Matches a single character
*	Matches all or none characters
#	Matches a single digit
[character list]	Matches any single character in the character list
[! Character list]	Matches any single character NOT in the character list

The **Like** operator gives you many options when looking for patterns in a string. If the pattern is found in the string, the expression returns True. If the pattern is not found, the expression returns False. Let's look at some examples of using the Like operator and the value of the expression:

```
"abcdefg" Like "a*a"       'Expression is False
"abcdefga" Like "a*a"      'Expression is True
"abc" Like "a?a"           'Expression is False
"aba" Like "a?a"           'Expression is True
"aba" Like "a#a"           'Expression is False
```

```
"a1a" Like "a#a"           'Expression is True
"abcdefga" Like "a[a-z]a"  'Expression is False
"aba" Like "a[a-z]a"       'Expression is True
"aba" Like "a[!a-z]a"      'Expression is False
"aBa" Like "a[!a-z]a"      'Expression is True
```

Sometimes a single expression in an **If…Then…Else** statement is not enough. You can use multiple expressions to create a single True or False value for an **If…Then…Else** statement. You can use the logical operators to create compound expressions that, as a whole, returns a single Boolean value. Table 5.4 lists the logical operators.

Table 5.4 Logical Operators

Logical Operator	Function
And	Both expressions must be True for a True result
Not	Expression must evaluate to False for a True result
Or	Either one of the expressions must be True for a True result
Xor	Only one expression can be True for a True result

You can combine the logical operators to create multiple expressions. You can use parentheses to logically group expressions together. Let's take a look at some examples of compound expressions:

```
1=1 And 1=2              'Expression is False
1=1 And 1<2              'Expression is True
1<1 Or  1=2              'Expression is False
1=1 Or  1=2              'Expression is True
(1=1 And 1=2) Or 1=3     'Expression is False
(1=1 And 1=2) Or 1<3     'Expression is True
Not 1=1                  'Expression is False
1=1 And Not 1=2          'Expression is True
1=1 Xor 1<2              'Expression is False
1=1 Xor 1<2              'Expression is True
```

As you can see, you have virtually unlimited possibilities for determining a result in your applications. Now let's take a look at using some of these expressions in an **If…Then…Else** statement. In the following example, the numbers from the preceding example are replaced with variables:

```
IntA = 1
IntB = 2
IntC = 3
IntD = 1

If intA=intD And intA=intB Then               'Expression is False
  'Do something
ElseIf intA=intD And intA<intB Then           'Expression is True
  'Do something
ElseIf intA=intD Or intA=intB Then            'Expression is True
  'Do something
ElseIf (intA=intD And intA=intB) Or intA=intC Then 'Expression is False
  'Do something
ElseIf intA=intD And Not intA=intB Then       'Expression is True
  'Do something
ElseIf intA=intD Xor intA<intB Then           'Expression is False
  'Do something
Else
  'Do something
End If
```

Select Case

The **Select Case** statement is similar to the **If…Then…Else** statement. The functionality is basically the same, except you get a cleaner way to write the code. Whenever **If…Then…Else** statements have more than a few **Else…If** blocks, the code becomes hard to read and follow. A general rule is to use the **Select Case** statement when you have more than two **Else…If** statements. The syntax for the **Select Case** statement is as follows:

```
Select Case testexpression
[Case expressionlist-n
   [statements-n]] ...
```

```
[Case Else
   [elsestatements]]
End Select
```

The **Select Case** will compare the **Case** statement expressions to the *testexpression*. The *expressionlist* is one or more values separated by commas to compare against the *testexpression*. The statements for the first **Case** that matches the *testexpression* will be executed. Even if subsequent **Case** expressions match, they will not be executed. If none of the **Case** expressions match, the statements under the **Case Else** will be executed. Let's look at an example. Let's take the **If…Then…Else** statement from the preceding code and convert it to a **Select Case** statement:

```
Select Case sngAmt
    Case Is <= MIN_AMT
        sngShipCost = MIN_SHIP_COST
    Case Is <= SECOND_AMT
        sngShipCost = SECOND_SHIP_COST
    Case Is <= THIRD_AMT
        sngShipCost = THIRD_SHIP_COST
    Case Else
        sngShipCost = MAX_SHIP_COST
End Select
```

You can also use multiple expressions or even ranges in a **Case** expression, and you can also match strings, as shown in the following code:

```
Select Case intTest
    Case 1 To 5, 10 To 15, 21
         'Do something
End Select

Select Case strTest
    Case "match1", "match2"
        msgbox("Found match 1 or 2")
    Case "match3"
        msgbox("Found match 3")
End Select
```

While Loops

At times, you may wish to execute a block of code multiple times without knowing beforehand how many times it needs to execute. You can use a **While** loop to execute a block of code until a condition becomes False. That is, the loop will continue to execute as long as the condition remains True. A common example is looping through a recordset until you reach the end of it. You can use **While** loops in multiple ways—let's look at the syntax for the first way to use it:

```
Do [{While | Until} condition]
   [statements]
   [Exit Do]
   [statements]
Loop
```

Or:

```
Do
   [statements]
   [Exit Do]
   [statements]
Loop [{While | Until} condition]
```

The loop is started with the **Do While** keywords. The *condition* is checked to see if it is True prior to each execution of the *statements* including the first iteration. The **Do Until** statement will execute until the condition is True, whereas the **Do While** will execute until the condition is False. For those of you accustomed to using the **While…Wend** syntax, this is no longer available in Visual Basic .NET.

In the following example, we keep adding the value 5 to a variable as long as its value stays below 100:

```
Dim val As Integer = 0
Do While val < 100
    Val = val + 5
Loop
```

The code inside the **While** loop will execute over and over until the variable *val* becomes greater than or equal to the value 100.Sometimes, you may want to exit a **While** loop before the condition is False. Let's expand our previous example to count how many times we add the value 5 to the variable, and if we add it 10 times, exit the loop.You can exit a **While** loop at any time by using the **Exit Do** statement as shown here:

```
Dim val As Integer = 0
Dim ctr As Integer = 0
Do While val < 100
    val = val + 5
    ctr = ctr + 1
    If ctr >= 10 Then
        Exit Do
    End If
Loop
```

The **While** loop will stop executing under two conditions: when the variable *val* is greater than or equal to the value 100 or when the variable *ctr* is greater than or equal to the value 10. Of course, we could create a compound conditional statement to achieve the same result and make our code cleaner.You can use the same logical and comparison operators described in the "If…Then…Else" section earlier in the chapter. Sometimes, exiting a loop cleanly is difficult, and you will need to use the **Exit Do** statement. Here is an example of how this can be rewritten:

```
Dim val As Integer = 0
Dim ctr As Integer = 0
Do While val < 100 And ctr < 10
    val = val + 5
    ctr = ctr + 1
Loop
```

Sometimes, you may always want a loop to execute at least one time. In the preceding examples, if our variable *val* had been initialized to the value 100, the code inside the loop would have never executed.To force our code inside the loop to execute at least once, you can use another variation of the **While** loop as shown here:

```
Dim val As Integer = 0
Dim ctr As Integer = 0
Do
```

```
    val = val + 5
    ctr = ctr + 1
Loop Until val > 100 And ctr >= 10
```

In this example, the code always executes at least once because the conditional is not evaluated until after the code inside the loop has executed. Also note that you can use **Loop While** to execute until the condition is False. **While** loops can be very powerful for performing complex operations. You can also nest loops inside of each other.

For Loops

The **For** loop is similar to the **While** loop except in this case, you are executing the code in the loop a fixed number of times. This is useful when reading or writing to arrays (covered in the next section). Let's take a look at the syntax:

```
For counter = start To end [Step step]
    [statements]
    [Exit For]
    [statements]
Next
```

The *counter* is a numeric variable used as the loop counter. This variable is used to keep track of the number of iterations through the loop. The *start* value is what the loop counter is initialized to. The *end* value is the max value of the loop counter before the loop stops executing. The **Step** clause is optional. This is how much the loop counter will be incremented by each time through the loop after the first execution. By default, this will increment the counter by a value of 1. The *step* value can be negative to decrement the counter. In this case, the loop will execute until the loop counter is less than the *end* value. As an example, let's create a **For** loop that will increment a variable 10 times by 5:

```
Dim I As Integer
Dim val As Integer = 0
For i = 1 To 10
    Val = val + 5
Next
```

After the loop is finished executing, the value of the variable *val* is equal to 50 and the value of *i* will be 11. This is because after the tenth iteration through the loop, *i* is incremented by 1 to 11, which is greater than the *end* value of 10, and the loop stops executing. We will see some more examples of **For** loops later in this chapter in the "Arrays" section.

Another form of the **For** loop is **For…Each…Next**. This loop is used with arrays and collections. It *loops* through each item in the array or collection. We haven't discussed arrays and collections yet, but let's look at how they are used. The syntax is very similar to a **For** loop, as shown here:

```
For Each element In group
   [ statements ]
[ Exit For ]
   [ statements ]
Next [ element ]
```

The *element* must be the same data type as each item in the array or collection. The *group* is an array or collection. The loop will automatically step through each element in the array or collection and exit the loop to the end of the array or collection. Here is an example of looping through a collection:

```
Dim objItem, MyCollection As Object
For Each objItem In MyCollection   ' Iterate through items.
    If objItem.val = 5 Then
        Exit For   ' Exit loop.
    End If
Next
```

Arrays

At times, you may need to store multiple instances of like data. *Arrays* allow you to do this without having to create a separate variable for each item of data. An array stores all of the items in a single variable, and you can reference each item by using an array index or numerical subscript. All the elements of an array have the same data type (structures are allowed). You can "cheat" this rule by using the Object data type, which allows you to use different data types in an array. This is similar to using the Variant data type in previous versions of Visual Basic.

Arrays have a lower bound and an upper bound. Arrays have changed slightly in Visual Basic .NET. The lower bounds for an array is always 0. You cannot change the lower bound of an array as you could in previous versions of Visual

Basic. So, if an array had 10 elements, the lower bound would be 0 and the upper bound would be 9. As we will see later in this chapter, everything in .NET is an object. Arrays are inherited from the **System.Array** object and as such can use the properties and methods available.

NOTE

When porting Visual Basic applications to Visual Basic .NET, be careful of the lower bounds of arrays. If you are using a for loop to iterate through the array, and it is hard-coded to initialize the counter at 1, the first element will be skipped. Remember that all arrays start with the index of 0.

You can think of an array as a row of items that contain values. For example, if you had an array of integers with five elements, it would be represented as shown here with one row of five columns where each element is a column:

1	2	3	4	5

Declaring an Array

To declare an array variable, the syntax is similar to other variables. You still use the **Dim** or **Scope** (**Public**, **Private**, **Friend**, and so on) keyword except that you add parentheses after the variable name to indicate that it is an array. For example, to declare an array of integers with 10 elements, use the following syntax (you are limited to an upper bound of the max value of a Long data type [$2^{64} - 1$]):

```
Dim arr(10) As Integer
Dim arr() As Integer = New Integer(10) {}
```

You can also initialize the values of an array when you declare it. The following line of code is the syntax for declaring an array of integers and initializing the values. Notice that the parentheses are left blank. It is automatically dimensioned to the correct number of elements:

```
Dim arr() As Integer = {0,1,2,3,4}
```

Now let's see how we can use an array. To read or write an element in the array, you use the array's variable name followed by parentheses. Inside the parentheses, the number indicates which element in the array is being referenced. Let's take a look at some examples:

```
arr(0) = 5
i = arr(3)
```

Now, let's take a look at how you can use a **For** loop with an array as mentioned earlier in the chapter. We will set each element of an integer array to its index value using a **For** loop as shown here:

```
Dim arr(5) As Integer
Dim i As Integer
For i = 0 To 4
    arr(i) = i
Next
```

We could also loop through an array looking for a specific value, as in this example:

```
For i = 0 To 4
    If arr(i) > 10 Then
        Exit For
    End If
Next
```

Visual Basic has two functions that are used to determine the upper and lower bounds of an array. The **LBound** function is used to retrieve the lower bound of an array (always zero), and the **UBound** function returns the upper bound of an array. We could change our for loop to read as follows:

```
For i = LBound(arr) To UBound(arr)
    arr(i) = i
Next
```

Multidimensional Arrays

Arrays can have more than one dimension. In fact, in Visual Basic .NET, they can have up to 60 dimensions, although it is uncommon to go above 3 dimensions. An easy way to picture this is to expand our previous example beyond one row of columns to multiple rows of columns. For example, an array with three rows of five columns would be represented by the following illustration where the first digit represents the row and the second digit represents the column:

11	12	13	14	15
21	22	23	24	25
31	32	33	34	35

To declare this array, use the following syntax:

```
Dim arr(3,5) As String
```

To declare a multidimension array, you separate the length for each row by a comma. If you wanted to declare a 2-dimensional array with 5 elements in the first dimension and 10 elements in the second dimension, declare it as follows:

```
Dim arr(5,10) As String
```

To initialize a multidimensional array when declaring it, you leave the parentheses empty except for a comma for each additional array dimension. For example, to initialize an array, use the following syntax:

```
Dim arr(,) As String = {{"11", "12", "13"}, {"21", "22", "23"}}
```

You can still use the **LBound** and **UBound** functions with multidimensional arrays, but you need to also tell it which row to return the value for. By default, it returns the value for the first row, which is why we didn't need to pass in a value for single dimension arrays. Let's look at using the functions for iterating through each element in the array and setting it to zero:

```
Dim arr(3, 5, 7) As Integer
Dim i As Integer
Dim j As Integer
Dim k As Integer
For i = LBound(arr, 1) To UBound(arr, 1)
    For j = LBound(arr, 2) To UBound(arr, 2)
        For k = LBound(arr, 3) To UBound(arr, 3)
            arr(i, j, k) = 0
        Next
    Next
Next
```

NOTE

The memory size of an array is larger than just the memory needed to hold the data. An array requires 20 bytes for the array, plus 4 bytes for each dimension in the array plus the size of the data type for each element.

Dynamic Arrays

Once you have declared an array, you may need to change the size of it, which Visual Basic allows you to do. This is accomplished using the **Redim** keyword. When you declare an array, you don't have to specify its size. You can just declare it and the set its size later with the **Redim** keyword or you can redimension an existing array. Let's look at some examples:

```
Dim arr() As Integer
Dim i As Integer

ReDim arr(i)
For i = 0 To 3
    arr(i) = 0
Next
```

This example does not specify the size of the array when declaring it, but dimensions it using the variable *i* to specify its size. This allows you to create arrays to a size that is not known until runtime. Unlike previous versions of Visual Basic, you can now change the size of all the dimensions as shown here:

```
Dim arr(5, 5, 5) As Integer
Dim i As Integer

ReDim arr(3, 3, 3)
For i = 0 To 3
    arr(1, 1, i) = 0
Next
```

When using the **Redim** keyword as we have seen so far, a completely new array is created and any new existing data is lost. For example, if we initialized an array with values for each element and then redimensioned it, the values would be set to their defaults (for example, integers would be set to zero). Let's look at an example:

```
Dim arr() As Integer = {1, 2, 3}
Dim i As Integer
ReDim arr(5)
i = arr(0)
```

In this example, when *i* is set to *arr(0)*, its value is 0 rather than 1. To keep any existing values in an array when redimensioning it, use the **Preserve** keyword. This will copy the existing values into the new array, with one limitation: You can only resize the last dimension. The other dimensions must stay the same size. So let's see what happens if we change the preceding example:

```
Dim arr() As Integer = {1, 2, 3}
Dim i As Integer
ReDim Preserve arr(5)
i = arr(0)
```

This time, when *i* is set to *arr(0)*, its value is still 1. Thus, the original values are preserved after resizing the array.

NOTE

Some things to remember when using arrays:

- Because every array is an object, it also contains members that contain the array's rank and length information.
- If you assign one array variable to another, only the pointer is copied, not the entire array.

Functions

In most applications, some blocks of code are executed multiple times in different parts of your application. You can't just use a loop because the code is executed in different parts of your program. Sure, you can cut and paste the code wherever you use it, but if you discover a bug or have to change the code, you have to find and change the code everywhere it is used. You also can't have any typos or miss any of the blocks spread throughout the application. You can get around this problem by using functions. A *function* is a block of code that can be called (and even passed parameters) to perform some type of functionality. When the block of code is completed, execution returns to the line of code after the function was called. A procedure performs this same functionality, except that a function can return a variable. Let's see the syntax for functions:

```
[Public | Private | Friend] [Static] Function name [(arglist)] [As type]
[statements]
[name = expression]
[Exit Function]
[statements]
[name = expression]
End Function
```

The keywords prior to **Function** (**Public**, **Private**, and so on) deal with scope of a function (where it can be called from).

In previous versions of Visual Basic, to return a value of a function you set the name of the function (as if it was a variable) to the value. In Visual Basic .NET, you can use this method or use the **Return** keyword to return a value from a function as shown in the following code. The difference is that the **Return** keyword returns control from the function immediately, whereas using the function name sets the value to be returned but does not return from the function until it hits either an **End function** or **Exit function**. In Visual Basic 6.0 and earlier, you can do this:

```
Function GetPi() As Double
    GetPi = 3.14
End Function
```

In Visual Basic .NET, you can do this:

```
Function GetPi() As Double
    Return = 3.14 'return immediately
End Function
```

Or you can do this:

```
Function GetPi() As Double
    Dim pi as double = 3.14
    GetPi = pi 'doesn't return yet
    Pi = 4
End Function 'returns 3.14
```

Another significant difference is how parameters are passed to functions. In previous version of Visual Basic, parameters were passed by reference by default. In Visual Basic .NET, parameters are passed by value by default. The difference between the two is that when a parameter is passed by value, if the code inside the function changes the value of the parameter, the change is not seen by the code calling the function. It just passes in a copy of the value. When parameters are passed by reference, a pointer to the variable is passed into the function so that, if the parameter is changed inside the function, that change is also reflected outside the function. Let's look at an example where we pass a parameter by value:

```
Function F1(ByVal x As Integer) As Integer
    X = 1
    Return 0
End Function

Sub F2()
    Dim z As Integer = 0

    F1(z)
    MessageBox(z)
End Sub
```

In this example, when the variable *z* is displayed in the message box, the value will still be 0. Now let's change this example to pass the parameter by reference:

```
Function F1(ByRef x As Integer) As Integer
    X = 1
    Return 0
End Function

Sub F2()
    Dim z As Integer = 0

    F1(z)
    MessageBox(z)
End Sub
```

In this example, when the variable *z* is displayed in the message box, the value will be changed to one.

You can use two other keywords with function parameters: **Optional** and **ParamArray**. The **Optional** keyword is used for parameters that are not required to be supplied when calling a function. However, when a parameter is declared as **Optional**, all parameters after it must also be **Optional**. You must also supply a default value for the parameter. When this parameter is not specified when the function is called, the default value is used. Let's look at an example:

```
Function Multiply(ByVal x1 As Integer,
                   ByVal x2 As Integer,
                     Optional ByVal x3 As Integer = 1) As Integer
    Return x1 * x2 * x3
End Function
```

In this example, you have to pass in only parameters for *x1* and *x2*. If you don't provide a value for *x3*, it defaults to 1. Let's look at two calls to this function:

```
i = Multiply(2, 3)
i = Multiply(2, 3, 4)
```

In the first call, the **Optional** parameter is not provided, and the function returns the value 6. In the second call, the value 4 is provided, and the function returns the value 24. Now let's take a look at how you use **ParamArray**. You can only use this keyword as the last parameter and with only one parameter. This keyword allows the function to be called with any number of arguments. The limitation is that all values must be passed by value. Let's modify our **Multiply** function to use **ParamArray**:

```
Function Multiply(ByVal ParamArray Args() As Integer) As Integer
    Dim i As Integer
    Dim val As Integer = 1

    For i = 0 To Args.Length() - 1
        val = val * Args(i)
    Next
    Return val
End Function
```

Notice that the *Args* array is an object and that we have used the length property to determine how many arguments were passed into the function. We then used a **For** loop to multiply all of the arguments together and return the result.

Object Oriented Programming

An object is an entity that contains state (or data) and behavior (methods). Think of a car. A car is a single object, yet it has numerous parts and behaviors. A car has a body, frame, wheels, doors, and so on. It's a single object made up of smaller objects. A car also has behavior (or actions). You can drive it, open the doors, turn, and so on. All of this is encapsulated in a single object: a car. Let's see what the pseudocode would look like for a basic vehicle object:

```
Vehicle Object
    NumWheels
    NumDoors
    Color
    Drive()
End Vehicle Object
```

A class is a template for objects. An object is an instance of an object in memory. You can have multiple instances of the same class. In previous versions of Visual Basic, you had some limited benefits of objects. You could create a class by adding a Class module to your application, but you were limited to one class per class module. In Visual Basic .NET, you can still use Class modules, but they aren't required, and you can have multiple classes in the same module. Object oriented programming has three basic concepts: Inheritance, Polymorphism, and Encapsulation. Visual Basic .NET has extended its object oriented functionality beyond encapsulation and a pseudo-inheritance to create a true object oriented programming language. Microsoft has added true inheritance and polymorphism to Visual Basic. This section is by no means a complete reference to implementing objects in Visual Basic .NET—it is meant to provide a basic understanding. A complete book could be devoted to this topic.

Inheritance

Finally, Visual Basic allows true inheritance of objects. So, what does this really mean? *Inheritance* is a relationship where one object is derived from another object. When an object is inherited, all of its properties and methods are automatically included in the new object. Let's take our car example a little further and create a new object for a specific type of vehicle, a truck:

```
Truck Object
    Inherits Vehicle Object
    BedLength
End Vehicle Object
```

This new truck object will now have the same properties as the vehicle object (number of wheels and doors as well as color), but we have now added how long the truck bed is without have to recreate the other properties. As you can see, this can be a powerful tool for code reuse. You don't have to rewrite code just to tweak it to your specific needs.

Polymorphism

Polymorphism allows an inherited method to be overridden. This means that if we inherit an object and want to change the functionality of an inherited method, we can add a new method to the new object with the same method name. Thus, when the method is called for the new object, it will execute the new method. If an object is created from the original object, the original method will be used. If an object is created from the new object, the new method is used. This prevents you from having to implement separate methods for drive for each type of vehicle (such as TruckDrive, CarDrive, and so on). If we wanted to change the functionality of our Drive method, our new object would look like the following example:

```
Truck Object
    Inherits Vehicle Object
    Drive()
    BedLength
End Vehicle Object
```

Encapsulation

Encapsulation was available in previous version of Visual Basic. *Encapsulation* is, simply put, a technique to hide information in an object by combining both data and operations on that data within an object. For example, our vehicle object is a single entity that contains data (number of wheels and doors, color, and so on) and operations (drive). This is all wrapped up inside a single object. You can control what methods and properties are available outside the object and which are hidden. A common practice is that before a property is changed, it can be validated before changing it.

Classes

A *class* is a template for an object. This is how you define a class. This is where you specify the properties and methods of your class. It is a data structure that can contain data members such as constants, variables, and events and function members such as methods and properties. Let's start by looking at the basic syntax for declaring a class:

```
Class name
   [ statements ]
End Class
```

The **Class** keyword start the class definition. The *name* is the name to be used to create instances of this class. The *statements* comprise the methods, properties, variables, and events of the class. Within a class, access to each member can be specified. A member declared as **Private** is available only from within the class. A member declared as **Public** is available inside the class as well as outside the class. **Public** is the default declaration if not specified.

In Visual Basic .NET, the **Set** keyword is no longer needed and in fact will give you a syntax error if you try to use it. In Visual Basic 6.0, the **Set** keyword was needed because of default properties (as in a Label control had the default property of Caption). So when setting one label variable to another, if you didn't use the **Set** keyword, it just copied the value in the Caption property from one control to the other. If you used the **Set** keyword, it actually set the object reference to the other object. In Visual Basic .NET, default properties are no longer valid unless they take parameters, which eliminates the need for the **Set** keyword.

Adding Properties

Properties store information in an object. Properties can be specified by either public variables or as property methods. When properties are declared using a public variable, they are also referred to as *fields.* This method does not provide the ability to control (validate) reading and writing a property. Property methods allow you to control read and write operations of properties in a class. This is similar to creating properties in a COM object in previous versions of Visual Basic. Let's look at how to declare each type. Following our vehicle example, we will use a public variable for the number-of-doors property and property methods for our number-of-wheels property:

CD File 5-1

```
Class Vehicle
    Public NumDoors As Integer = 2

    Private NumWheelsValue As Integer = 4 ' Used for property ops.
    Public Property NumWheels() As Integer  ' This is a Property.
        Get
            Return NumWheelsValue
        End Get
        Set(ByVal Value As Integer)
            ' Only allow set operation for values less than 13.
            If Value < 13 Then
                NumWheelsValue = Value
            End If
        End Set
    End Property
End Class

Function testVehicle()
    Dim clsVehicle As New Vehicle()
    Dim clsOtherVehicle As Vehicle

    clsVehicle.NumDoors = 4
    clsVehicle.NumWheels = 4

    clsOtherVehicle = New Vehicle()
    clsOtherVehicle.NumWheels = 13

End Function
```

In this example, the **testVehicle** function instantiates (or creates) the object in the variable *clsVehicle*. Unlike other data types such as integers and strings, a class has to be explicitly created using the **New** keyword. You can do this when declaring the variable or later in code. Notice that both of the properties are used the same way. However, when we try to set the number of wheels to 13, the value remains 4. No error message occurs, it just silently ignores the invalid value. So, when should you use property methods versus public variables? We have already seen that to validate values for properties, you must use the property method. There may also be cases where a property setting affects other portions of a class. Say, for example, our class had a number-of-windows property. If we changed the number of doors, this would automatically affect the number of windows, and you could use a property method to reflect this change transparent to the user of the class. To make a property read-only, you must use property methods. This is achieved by using the **ReadOnly** keyword, supplying only a *Get* method and not implementing a *Set* method. Let's change our class to reflect this:

CD File 5-2

```
Class Vehicle
   Public NumDoors As Integer = 2

   Private NumWheelsValue As Integer = 4 ' Used for property ops.
    Public ReadOnly Property NumWheels() As Integer
        Get
            Return NumWheelsValue
        End Get
    End Property
End Class
```

Adding Methods

Methods of a class perform an action or operation. These are simply public functions or procedures in a class. Let's implement the *Drive* method for our Vehicle class in the following code:

CD File 5-3

```
Class Vehicle
   Public NumDoors As Integer = 2

   Private NumWheelsValue As Integer = 4 ' Used for property ops.
    Public ReadOnly Property NumWheels() As Integer
        Get
            Return NumWheelsValue
        End Get
    End Property

    Public Sub Drive()
        'make vehicle drive
    End Sub
End Class
```

Even though our Drive procedure doesn't do anything, you can see how easy it is to create class methods. If the procedure had been declared as **Private**, it could be called only within the class and would not be available outside the class.

System.Object

Everything in .NET is derived from the *System.Object* class. You can think of it as the super class or root of all classes in .NET. When a class is created, it automatically inherits the properties and methods of the *System.Object*. Everything in .NET is an object, therefore everything in .NET is derived from *System.Object*.

Constructors

Constructors are methods of a class that are executed when a class is instantiated. This is commonly used to initialize the class. To create a constructor, you simply add a public procedure called **New()**. Then add any initialization code inside this method as shown here:

```
Public Sub New()
    NumDoors = 4
    NumWheelsValue = 4
End Sub
```

This constructor actually gets executed when the class is created with the **New** keyword. In this example, we just set the number of doors to 4 and the number of wheels to 4. We could expand the constructor to allow the user of the class to pass in the initial values for these by using a parameterized constructor. We will add two parameters for the number of doors and wheels to initialize to and pass in these values when we create the class as shown here:

```
Public Sub New(ByVal Doors As Integer, ByVal Wheels As Integer)
    NumDoors = Doors
    NumWheelsValue = Wheels
```

```
End Sub


Function testVehicle()
    Dim clsVehicle As New Vehicle(4, 4)
    clsVehicle.NumDoors = 4
End Function
```

What if we don't want to require the user of the class to have to specify these values? We can then use optional parameters with default values as shown here:

```
Public Sub New(Optional ByVal Doors As Integer = 4,

               Optional ByVal Wheels As Integer = 4)
            NumDoors = Doors
            NumWheelsValue = Wheels
        End Sub
```

Overloading

Overloading provides the ability to create multiple methods or properties with the same name, but with different parameter lists. This is a feature of polymorphism. It is accomplished by using the **Overloads** keyword. A simple example would be an Addition function that can add real number or integers. You could create two methods with the same name, but one would take integer parameters and one would take real numbers. This prevents you from having to create a method for each data type with different names. Let's take a look at how these are declared:

CD File 5-4

```
Public Overloads Function Add(ByVal x As Integer,
    ByVal y As Integer)
    Return x + y
End Function
Public Overloads Function Add(ByVal x As Double,
    ByVal y As Double)
    Return x + y
End Function


'method usage
clsVehicle.Add(1, 1)      'calls first Add function
clsVehicle.Add(1.5, 1.5)  'calls second Add function
clsVehicle.Add(1, 1.5)    'calls second Add function
```

This example creates two functions called Add using the **Overloads** keyword. If two integers are passed in, the first Add function is executed. If either of the parameters is a real number, the second Add function is executed. It implicitly converts the *integer* parameter to a double data type. When overloading methods the compiler must be able to differentiate between them. For example, you can't just change parameter names and leave them with the same data type. You cannot just change it from a public method to a private method or change the return data type. You cannot just change a parameter in one method from *ByVal* to *ByRef*.

Overriding

Inheriting a class allows you utilize the methods and properties of a class without having to implement them by simply reusing the existing ones. However, there are times when you want to change the functionality of an inherited method or property. You don't want to have to create a new method with a new name, you just want to override the existing member. This is another feature of polymorphism. You accomplish this by using the **Overridable** keyword in the base class and the **Overrides** keyword in the derived class. Let's look at an example where we start with a base class called *Square* with a method to calculate the circumference of a square:

CD File 5-5

```
Class Square
  Public Overridable Function getCircumference(ByVal r As Double)
                                                         As Double
        Return (2 * r) * 4 'length of side time 4 sides
    End Function
End Class
Function testSquare()
    Dim clsSquare As New Square()
    Dim circ As Double

    circ = clsSquare.getCircumference(1)
End Function
```

In this example, the value of the variable *circ* will be 8 after calling *GetCircumference* in the *Square* class. Now let's take a look at inheriting this class in a circle class. The circumference is calculated differently, so we will want to override it:

CD File 5-6

```
Class Circle
    Inherits Square

  Public Overrides Function getCircumference(ByVal r As Double)
                                                        As Double
        Return 2 * 3.14 * r
    End Function
End Class

Public Function testCircle()
        Dim clsCircle As New Circle()
        Dim clsSquare As New Square()
        Dim circSquare As Double
        Dim circCircle As Double

        circCircle = clsCircle.getCircumference(1) 'returns 6.28
        circSquare = clsSquare.getCircumference(1) 'returns 8
    End Function
```

In this example, we overrode the *getCircumference* function to calculate the circumference of a circle. In the *testVehicle* function, the call to the circle class method returns the value 6.28, and the call to the square class method returns the value 8. Now you can begin to see the power of polymorphism. It allows users of your classes to use some standardized methods without have to change the name of it for every little difference. You must follow some rules for overriding members. You can only override members that are declared with the **Overridable** keyword in the base class. When overriding a member, it must have the exact same arguments. You can call the base class method from within the derived class using the **MyBase** keyword. For example, we could add a method in the derived class called *getSquareCircumference*, and it could just call the base class method as shown in the following example. Granted, this isn't a plausible example, but it does illustrate the available functionality. This example simply passes the argument to the base class method and returns its value:

CD File 5-7

```
Public Function getSquareCircumference(ByVal r As Double) As Double
```

```
    Return MyBase.getCircumference(r)
End Function
```

You can use some additional keywords for overriding members of a class. The **NotOverridable** keyword is used to declare a method that cannot be overridden in a derived class. Actually, this is the default, and if you do not specify **Overridable**, it cannot be overridden. The **MustOverride** keyword is used to force a derived class to override this method. You commonly use this when you do not want to implement a member, but require it to be implemented in any derived class. For example, if we started with a shape class with the *getCircumference* method, we couldn't implement it because each shape would require a different calculation. But, we could force each derived class to implement for its particular shape (such as circle or square). When a class contains a **MustOverride** member, the class must be marked with **MustInherit** as shown here:

CD File 5-8

```
MustInherit Class Shape
MustOverride Function getCircumference(ByVal r As Double) As Double
End Class

Class Square
    Inherits Shape
    Public Overrides Function getCircumference(ByVal r As Double)
                                                                  As Double
        Return 2 * r * 4
    End Function
End Class
```

Shared Members

In all the classes we have seen so far, a member is available only within that particular instance of the class. If two instances of the same class were created and you changed a property in one of the instances, it would not change the value of the property in the other instances. Shared members are members of a class that are shared between all instances of a class; for example, if you did change a property in one instance, that change would be reflected across all instances of the class. This lets you share information across instances of classes. Let's look at an example where we track how many instances of a class are instantiated:

CD File 5-9

```
Class SomeClass
    Private Shared NumInstances As Integer = 0
    Public Sub New()
        NumInstances = NumInstances + 1
    End Sub
    Public ReadOnly Property Instances() As Integer
        Get
            Return NumInstances
        End Get
    End Property
End Class

Public Sub testShared()
    Dim clsSomeClass1 As New SomeClass()
    Dim clsSomeClass2 As SomeClass
    Dim num As Integer

    num = clsSomeClass1.Instances ' returns 1
    clsSomeClass2 = New SomeClass()
    num = clsSomeClass2.Instances ' returns 2
End Sub
```

In this example, we created a constructor that increments the *NumInstances* variable. When the first class is instantiated, this variable is equal to 1. When the second class is instantiated, the value becomes equal to 2.

String Handling

For those of you have gotten accustomed to the powerful string manipulation functions in Visual Basic, don't worry, that power is still available. As we have stated numerous times already, everything in .NET is an object. Therefore, when you create a string variable, the string methods are already built in. A string variable is actually an instance of a string class. Table 5.5 lists the most common built-in methods of the string class.

Table 5.5 String Class Methods

Method	Description
Compare	Compares two string objects
Concat	Concatenates one or more strings
Copy	Creates a new instance of the string class that contains the same string value
Equals	Determines whether or not two strings are equal
Format	Formats a string
Equality Operator	Allows strings to be compared using the = operator
Equality Operator	Allows strings to be compared using the <> operator
Chars	Returns the character at a specified position in the string
Length	Returns the number of characters in the string
EndsWith	Determines whether or not a string ends with a specified string
IndexOf	Returns the index of the first character of the first occurrence of a substring within this string
IndexOfAny	Returns the index of the first occurrence of any character in a specified array of characters
Insert	Inserts a string in this string
LastIndexOf	Returns the index of the first character of the last occurrence of a substring within this string
LastIndexOfAny	Returns the index of the last occurrence of any character in a specified array of characters
PadLeft	Pads the string with spaces on the left to right-align a string
PadRight	Pads the string with spaces on the right to left-align a string
Remove	Deletes a specified number of characters at a specified position in the string
Replace	Replaces all occurrences of a substring with a specified substring
Split	Splits a string up into a string array using a specified delimiter
StartsWith	Determines whether or not a string starts with a specified string
SubString	Returns a substring within the string

Continued

Table 5.5 Continued

Method	Description
ToLower	Returns a copy of the string converted to all lowercase letters
ToUpper	Returns a copy of the string converted to all uppercase letters
Trim	Removes all occurrences of specified characters (normally whitespace) from the beginning and end of a string
TrimEnd	Removes all occurrences of specified characters (normally whitespace) from the end of a string
TrimStart	Removes all occurrences of specified characters (normally whitespace) from the beginning of a string

As you can see, numerous string manipulation methods are available. Let's take a look at how some of these are used. When working with strings in Visual Basic .NET, the strings are zero-based, which means that the index of the first character is 0. In previous versions of Visual Basic, strings were one-based. For those of you who have been using Visual Basic for a while, this will take some getting used to. Remember that most string methods return a string, they do not manipulate the existing string, which means that you need to set the string equal to the string method as shown here:

```
Dim str As String = "12345"
str.Remove(2, 2)     'str still = "12345"
str = str.Remove(2, 2) 'str = "12345"
```

In the first call to the *Remove* method, the *str* variable value does not change. In the second call, we are setting the *str* variable to the returned string from the *Remove* method, and now the value has changed:

CD File 5-10

```
Dim str As String = "Cameron"
Dim str2 As String
Dim len As Integer
Dim pos As Integer

len = str.Length() 'len = 7
str2 = str         'str2 now = "Cameron"

If str.Compare(str, str2) = 0 Then
```

```
8     'strings are equal
9 ElseIf str.Compare(str, str2) > 0 Then
10     'string1 is greater than string2
11 ElseIf str.Compare(str, str2) < 0 Then
12     'string2 is greater than string1
13  End If

14 If str = str2 Then
15     'same instance
16 Else
17     'difference instances
18 End If

19 str = str + "W" 'str = "CameronW"
20 str = str.Insert(7, " ") 'str now = "Cameron W"

21 If str.EndsWith(" W") Then
22     str.Remove(7, 2)     'str still = "Cameron W"
23     str = str.Remove(7, 2) 'str = "Cameron"
24 End If

25 pos = str.IndexOf("am")  'pos = 1
26 pos = str.IndexOfAny("ew") 'pos = 3
```

Now let's take a look at what we have done. In line 7, we are using the **Compare** function to see if the string values are equal. They are equal, and line 8 would be executed next. In line 14, we are comparing the string references, not the string values. Because these are two separate instances of the *String* class, they are not equal, and line 17 would execute next. Remember that this does not compare the string values. In line 25, the index returned is equal to 1 because arrays are zero-based. This function is looking for the entire substring in the string. In line 26, this method is looking for any of the characters in the substring in the string. Even though *w* is not in the string, it finds *e* and returns the index 3.

Error Handling

To prevent errors from happening after you distribute your application, you need to implement error trapping and handling. This will require you to write good error handlers that anticipates problems or conditions that are not under the control of your application and that will cause your program to execute incorrectly at runtime. You can accomplish this largely during the planning and design phase of your application. This requires a thorough understanding of how your application should work, and the anomalies that may pop up at runtime.

For runtime errors that occur because of conditions that are beyond a program's control, you handle them by using exception handling and checking for common problems before executing an action. An example of checking for errors would be to make sure that a floppy disk is in the drive before trying to write to it or to make sure that a file exists before trying to read from it. Another example of when to use exception handling is retrieving a recordset from a database. You might have a valid connection, but something might have happened after your connection to cause the retrieval of a recordset to fail. You could use exception handling to trap this error rather than a cryptic message popping up and then aborting your application.

You should use exception handling to prevent your application from aborting. They provide support for handling runtime errors, called *exceptions*, which can occur when your program is running. Using exception handling, your program can take steps to recover from abnormal events outside your program's control rather than crashing your application. These exceptions are handled by code that is not run during normal execution.

In previous versions of Visual Basic, error handling used the **On Error Goto** statement. One of the major complaints made by programmers has been the lack of exception handling. Visual Basic .NET meets this need with the inclusion of the **Try...Catch...Finally** exception handling. Those of you who have programmed in C++ should already be familiar with this concept. Let's take a look at the syntax for exception handling:

```
Try
   tryStatements
[Catch [exception [As type]] [When expression]
   catchStatements1
…
Catch [exception [As type]] [When expression]
   catchStatementsn]
 [Finally
   finallyStatements]
End Try
```

The **Try** keyword basically turns the exception handler on. This is code that you believe is susceptible to errors and could cause an exception. The compound statement following the **Try** keyword is the "guarded" section of code. If an exception occurs inside the guarded code, it will throw an exception that can then be caught, allowing your code to handle it appropriately. The **Catch** keyword allows you to handle the exception. You can use multiple catch blocks to handle specific exceptions. The type is a class filter that is the class exception or a class derived from it. This class contains information about the exception. The **Catch** handlers are examined in order of their appearance following the **Try** block. Let's take a look at an example:

```
Dim num As Integer = 5
Dim den As Integer = 0
Try                       ' Setup structured error handling.
    num = num \ den              ' Cause a "Divide by Zero" error.
Catch err As Exception ' Catch the error.
    MessageBox.Show(err.toString) ' Show friendly error message.
    num = 0               ' set to zero
 End Try
```

In this example, the code that divides one variable by another is wrapped inside a **Try** block. If the denominator equals 0, an exception will occur and execution will move to the **Catch** block. The **Catch** block displays the error message and then sets the variable to 0. You can also include multiple **Catch** blocks to handle specific types of errors, which allows you to create different exception handlers for different types of errors. Let's expand our previous example by adding an additional **Catch** block that catches only divide-by-zero exceptions. Any other exceptions are handled by the second **Catch** block.

CD File
5-11

```
Dim num As Integer = 5
Dim den As Integer = 0
Try                         ' Setup structured error handling.
    num = num \ den              ' Cause a "Divide by Zero" error.
Catch err As DivideByZeroException ' Catch the divide by zero error.
    MessageBox.Show("Error trying to divide by zero.")
    num = 0                ' set to zero
Catch err As Exception ' Catch any other errors.
    MessageBox.Show(err.toString) ' Show friendly error message.
End Try
```

Summary

In this chapter, we have covered a broad portion of programming concepts. We discussed what variables are and how they differ from variables in previous versions of Visual Basic. Of significance is the new ability to initialize a variable when it is declared. Another significant change is that the Variant data type is no longer available. In previous versions of Visual Basic, when a variable was not given a data type, it was implicitly declared as a Variant data type. In Visual Basic .NET, the Object data type is the default. The **Type** keyword is no longer available; it has been replaced by structures. Structures are similar to classes with some limitations. Structures are useful for lumping like data together and allow access to each data member by name, such as an employee record.

When developing applications, you cannot just write lines of code that will always be executed. You will have to be able to change the flow of execution based on specified conditions. You can accomplish this through several program flow techniques. The **If...Then...Else** technique allows only certain blocks of code to be executed depending on a condition. The **Select** statement to be used to provide the same functionality as the **If...Then...Else**, but it is cleaner and easier to read when you have multiple **ElseIf** blocks. For executing through the same block of code multiple times, we discussed the **While** and **For** loops. Use the **While** loop when the number of iterations through the loop is not known ahead of time. Use the **For** loop when you want to iterate through the loop a fixed number of times.

Arrays are used to store multiple items of the same data type in a single variable. Imagine the headache of trying to create a hundred variables with the number 1–100 appended to the end of the variable name. This provides a simpler way to store like information. We saw that you can create multidimensional arrays and resize arrays.

Functions have changed somewhat in Visual Basic .NET. We now have two ways to return values. We can use the **Return** statement or the function name. Parameters have also changed from passing by reference as the default to passing by value.

Everything in .NET is an object. This is a major paradigm shift from previous versions of Visual Basic. We saw that we now have true inheritance and polymorphism at our fingertips. We can create multiple classes in the same module and have a great deal of the functionality previously available only in C++. We saw how members can be overloaded and overridden and even how to share a member across instances of a class.

Because strings are objects, all of the string manipulation functions are part of the String class. If anything, it helps you to see all the methods available in place using Intellisense. No more searching through Help files for that desired string function. We also learned that string indexes are now zero-based instead of one-based as in previous versions of Visual Basic.

Error handling in Visual Basic .NET has changed dramatically compared to previous versions of Visual Basic. Previous versions used the **On Error Goto** syntax for error handling. Visual Basic .NET has changed to a structured approach using exception handling with the **Try…Catch…Finally** blocks, which make exception handling cleaner to read and provide more functionality. You can now use multiple **Catch** blocks to provide separate exception handlers for different kinds of errors.

Solution Fast Track

Variables

- ☑ Variables are named memory locations for storing data.
- ☑ The Variant data type is no longer available.
- ☑ You should use the correct data types. Do not use the Object data type unless necessary.
- ☑ You can initialize variables when you declare them as in other programming languages.

Constants

- ☑ *Constants* are similar to variables. The main difference is that the value contained in memory cannot be changed once the constant is declared.
- ☑ When you declare a constant, the value for the constant is specified.

Structures

- ☑ Structures allow you to create custom data types with named members in a single entity.

☑ The **Type** keyword is no longer available. You must now use the **Structure** keyword.

Program Flow Control

☑ **If…Then…Else** statements allow you to specify which blocks of code run under different circumstances.

☑ If you have several more than a few **ElseIf** statements, then you should use the Select statement for easier to read code.

☑ If you need to loop through a block of code an unspecified number of times, you should use a while loop.

☑ If you know how many times you need to loop through a block of code, you should use a for loop.

Arrays

☑ Arrays allow you to store multiple instances of a group of data with the same data type.

☑ All arrays now have a lower bound of zero.

☑ Arrays can be initialized when declared.

☑ Arrays can be created dynamically and resized as needed using the **ReDim** keyword.

Functions

☑ Functions now have two ways of returning values. You can still use the function name or the **Return** keyword.

☑ The default method of passing parameters is now by value. In previous version of Visual Basic, the default method was by reference.

☑ You can create optional parameters as well as parameter arrays that allow any number of parameters to be passed in.

Object Oriented Programming

- ☑ In Visual Basic .NET, everything is an object.
- ☑ Visual Basic .NET now supports true inheritance and polymorphism to become a more true object oriented programming language.
- ☑ You are no longer limited to one class per class module. In fact, your classes don't even have to be created in a class module.
- ☑ The **Set** keyword is no longer used when working with objects. This is because default properties without parameters are no longer used negating the need for the **Set** keyword.
- ☑ Constructors allow you to initialize your object when the object is created. You can even pass in parameters to the constructor.
- ☑ You can overload class methods. This allows you to create multiple methods with the same name but different parameters types.
- ☑ Overriding a method allows you to change the functionality of an inherited method.
- ☑ Shared members of a class are shared between all instances of the class. This means that if one instance of a class changes the member's value, then this changed would be seen in all instances of the class.

String Handling

- ☑ All those powerful string manipulation functions are now built into the String class.
- ☑ Strings indexes are now zero based instead of one based.
- ☑ When using a String class method, it does not directly manipulate the existing String value. It returns a string value. This means you must assign the return value to a String variable.
- ☑ Comparing two String variables using the equal operator compares the object reference, not the string values themselves.

Error Handling

- ☑ Error handling is now accomplished using a structured exception handling technique using the **Try…Catch…Finally**.
- ☑ You can use multiple **Catch** blocks to handle different types of exceptions appropriately.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: I am porting an application that used the Variant data type. Because this is no longer available, what data type can I use to replace it?

A: First of all, whenever possible, this should be replaced with the appropriate data type. If this is not possible, use the Object data type.

Q: I am porting an application from a previous version of Visual Basic. Do I have to change all of my functions to use the **Return** statement?

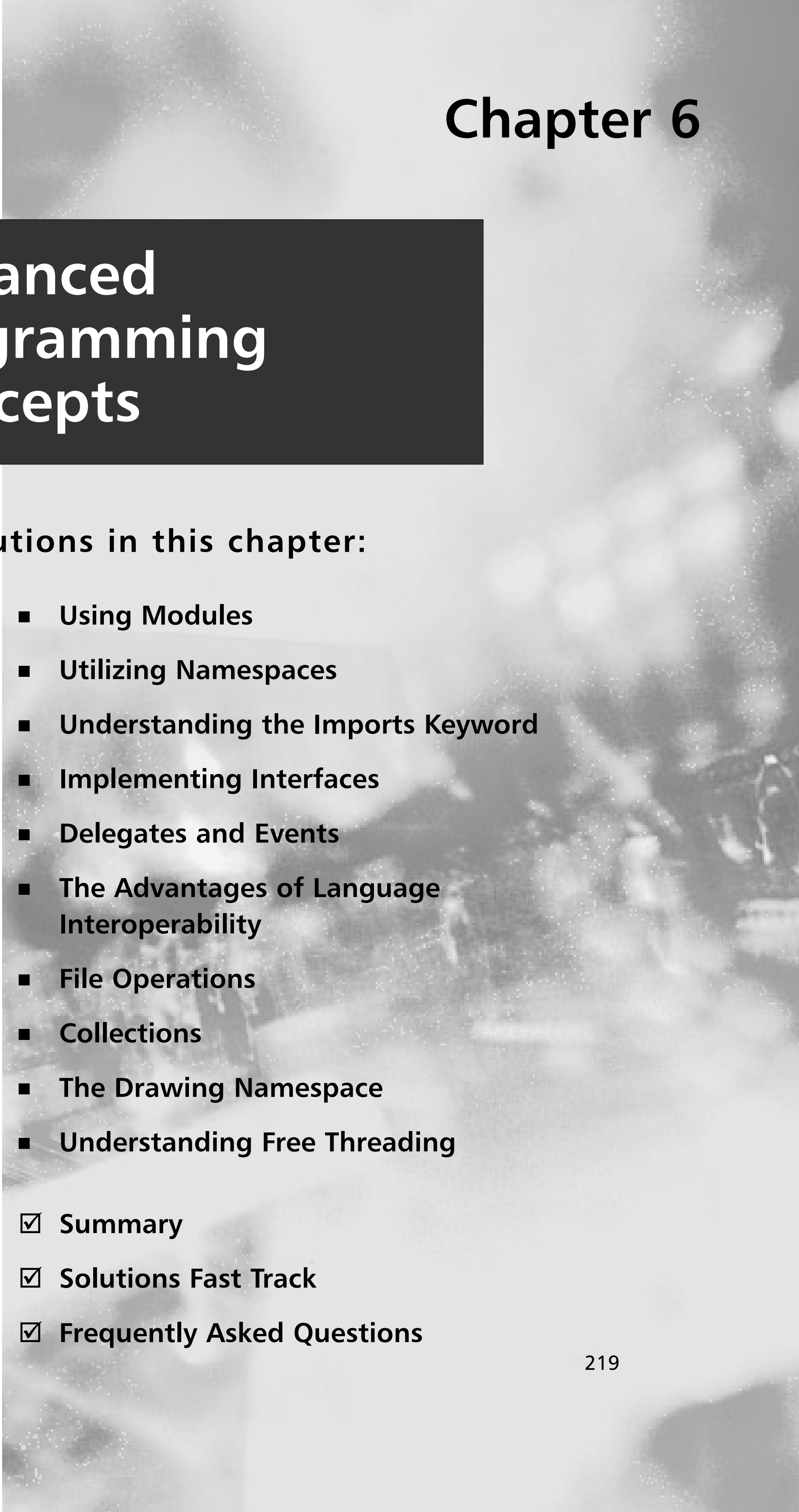
A: No, you can still use the function name to return a value from a function. Use the **Return** statement for new functions as they are created.

Q: I am reading in a file that has data stored in rows separated by commas (CSV format). Is there a function I can use to separate the data in each row into separate items without having to manually parse it?

A: Yes, you can use the *Split* method of the *String* class.

Q: I have a single function that I use frequently in my application. Because everything is now object oriented, should I put this function in a class?

A: No, if it is a standalone function that is not related to any other functions, you would just be creating unnecessary overhead.



Chapter 6

Advanced Programming Concepts

Solutions in this chapter:

Introduction

Now that we have covered the fundamentals of programming in Visual Basic .NET, let's take a look at some more advanced topics. The topics in this chapter are eclectic, but they are all important programming concepts. In object-oriented programming, we saw how there can be multiple instances of an object with different sets of data values. Shared members are class members that are shared across all instances of an object. This means that if the value is changed in one instance of an object, that change would be propagated to all of the instances. In previous versions of Visual Basic, you could create standard modules. You can still do this in .NET, but it creates a class where all the members are shared members.

We have covered the concept of namespaces and how they are used. In this chapter, we explore the programmatic side of namespaces and how to utilize them in your projects. The Imports keyword is used to allow the use of namespaces from other referenced projects or assemblies. This simplifies the use of other components and classes. An interface is a template for class members. It creates the contract for how the members are to be used. It does not provide the actual implementation of the class. In previous versions of Visual Basic, you could not explicitly create interfaces. You could create an interface of sorts by creating a class with all the methods left empty and then implement it. The process is cleaner in Visual Basic .NET. You can explicitly create interfaces using the Interface keyword. You can also implement interfaces using the Implements keyword. Interfaces allow you to design the interaction between components before you actually develop them. This allows a team of developers to create the contract on components and then split up and develop them independently.

Event handling has changed in Visual Basic .NET, including a new concept called *delegates.* Delegates can be thought of as function pointers that are type-safe. All events will be handled using delegates. For the most part, Visual Basic .NET will create all the delegates needed for you, but you will now have the power to override some event handlers. You can still create your own custom events as in previous versions of Visual Basic. Another new concept is the interoperability between languages. A project can contain code from different languages and you can even inherit classes in Visual Basic that were created in another programming language. This allows you to use the power of C++ or C# when needed, but you don't have to develop the entire project in one of them.

In Visual Basic 6.0, you could use the File System Object from the Scripting component. Now, with .NET, you can manipulate folders and files using the *System.IO* class. This is similar to the File System Object and gives you a nice clean object interface to file operations. This will allow for synchronous and asynchronous operations on files and data streams. Collections aren't new to Visual Basic, but it is now a .NET object. Collections are generally used with objects. It gives an easier way to program dynamic collections of data than arrays because it is object based with methods for adding and deleting members. The System object includes a namespace for drawing manipulation. This gives you access to the GDI+ engine. The functionality was available in previous versions of Visual Basic, but it is now available in one class to making it easier to use. Included in this namespace is the *Imaging* namespace for working with any type of images. The *Printing* namespace is also included, which allows you to control how documents are printed.

Finally, in Visual Basic, you can create true free threaded applications. This allows you to perform multitasking within your applications. As is normally the case, with this power comes the increased potential for problems. Use this feature with care and only when required. We also discuss some methods of synchronization that are available for multithreaded applications.

Using Modules

In previous versions of Visual Basic, generic functions that did not necessarily require the creation of an object were placed into Standard modules. Visual Basic .NET implements this functionality by allowing you to create shared members in your classes. Shared members are methods and functions of a class that can be called without actually instantiating an instance of the class.

When you add a shared method to a class, the method is accessed directly rather than through an object instance. A common use for shared methods is a utility class. In the following example, we create a utility class (*Math*) with a shared method (*Square*):

```
Public Class Math
     Shared Function Square(ByVal Number As Integer)
    As Integer
          Return Number * Number
     End Function
End Class
```

To use the *Square* function that we just created in the *Math* class, we do not need to create a Math object. We can call the function directly from the *Math* class as follows:

```
Dim iRet As Integer
IRet = Math.Square(25)
```

In the preceding example, note that no object of type *Math* was created or instantiated—not even behind the scenes. The *Square* method was called directly, just as it would have been called from a standard module in previous versions of Visual Basic.

All Visual Basic modules are now classes and need to be treated as such. You must be familiar with basic object-oriented programming techniques to be successful programming in Visual Basic .NET. You need to understand the basics of classes and how they are used.

If you are already familiar with using classes when programming in previous versions of Visual Basic, the adjustment to Visual Basic .NET will be easy because the principles are similar. If you are not familiar with using classes, fear not. The basic principles of programming with classes are easy to learn and will make you a better programmer.

Utilizing Namespaces

Namespaces are groupings of objects within assemblies. An *assembly* is everything that makes up a Visual Basic .NET application (which contains one or more namespaces). Namespaces allow programmers to create logical groups of classes and functions. Many assemblies are DLLs that can be used by an application.

Creating Namespaces

Namespaces allow programmers to group functions together logically. You can create a namespace as part of a DLL (which can be an assembly). This gives you the ability to easily reuse functions and classes. As you create namespaces, include your company name as the root of the namespace to help avoid naming conflicts as your code is reused.

The following code fragment will show programmatically how to create and implement namespaces in your code. This can help with code reuse and code segmentation. We can group like functions together within namespaces. This code is an example of a form that has a button on it that when clicked displays a message box. Some of the Windows Form Designer–generated code has been removed in order to conserve space. Additionally, we have removed the root namespace specified in the application's properties:

```
Imports System.ComponentModel
Imports System.Drawing
Imports System.WinForms

Namespace haverford.test.example
     Public Class frmDemoNameSpace
        Inherits System.WinForms.Form

   Public Sub New()
        MyBase.New()

        frmDemoNameSpace = Me

       'This call is required by the Win Form Designer.
        InitializeComponent()

   End Sub

  'Form overrides dispose to clean up the component list.
   Public Overrides Sub Dispose()
        MyBase.Dispose()
        components.Dispose()
   End Sub


Protected Sub Button1_Click(ByVal sender As Object, ByVal
    e As System.EventArgs)
        msgbox("Button Pressed!!!")
   End Sub

End Class
   End Namespace
```

In order to inherit or import this form, you need to prefix the form with its namespace. The following example shows a form inheriting this form. This form inherits all of the classes of the preceding form (*frmDemoNameSpace*). The most relevant part of the preceding code occurs in lines 4 and 20. These lines encapsulate the form in a namespace (determined by the programmer):

```
1       Imports System.ComponentModel
2       Imports System.Drawing
3       Imports System.WinForms

4       Public Class inhForm
5          Inherits haverford.test.example.frmDemoNameSpace

6         Public Sub New()
7           MyBase.New

8           inhForm = Me

            'This call is required by the WinForm Designer.
9           InitializeComponent

10        End Sub

    'Form overrides dispose to clean up the component list.
11       Overrides Public Sub Dispose()
12      MyBase.Dispose
13            components.Dispose
14        End Sub

15        End Class
```

In this example, we can see in line 5 that this form inherits the properties of the *haverford.test.example.frmDemoNameSpace* form.

Creating namespaces is especially helpful with code reuse. You can implement functionality between applications with very little effort, which makes programming much more efficient. You can think of namespaces as classes in a DLL, with the DLL itself being the assembly. If you are familiar with the concept of programming using classes in earlier versions of Visual Basic, the concept of namespaces should be easy to understand.

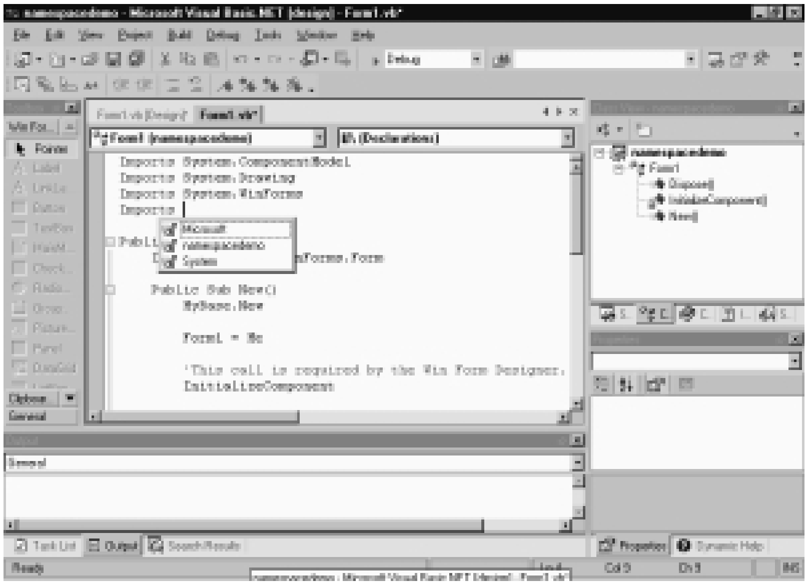
If a group of functions are contained in a namespace, you can import the namespace and then have access to all of the functionality of the namespace. A number of standard namespaces are used with Visual Basic. These namespaces will be used to accomplish different programming tasks in Visual Basic .NET. After a namespace is imported, you can access its methods directly. You do not have to prefix it with the namespace.

NOTE

If you have conflicting method names, you need to use the namespace to prefix the object. This allows the use of conflicting method names.

When you use the **Imports** command (as shown in Figure 6.1), you are given a list of assemblies to pick from, and within the assemblies, namespaces are available.

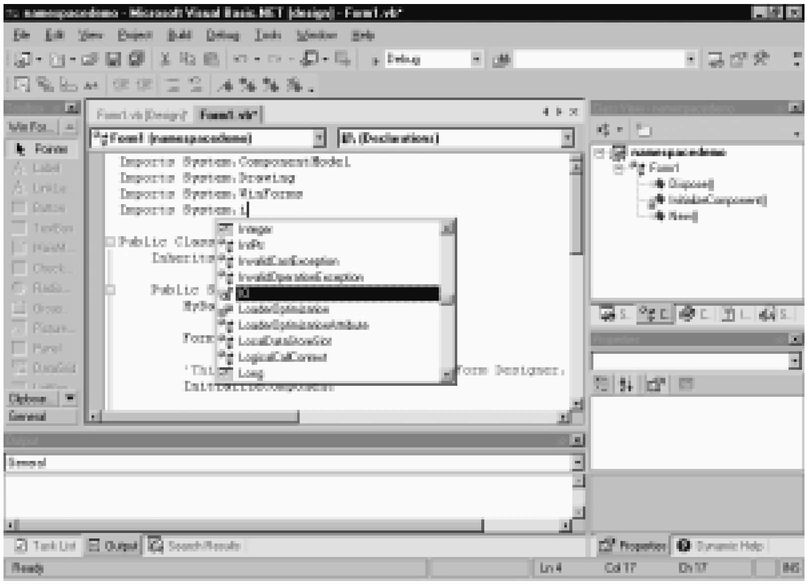
Figure 6.1 Using the **Imports** Command to Import Namespaces



```
Imports System.ComponentModel
Imports System.Drawing
Imports System.WinForms
Imports |
        Microsoft
        namespacedemo
        System
Public
        WinForms.Form

    Public Sub New()
        MyBase.New

        Form1 = Me

        'This call is required by the Win Form Designer.
        InitializeComponent
```

Figure 6.2 shows the *System.IO* namespace (as well as some of the other system namespaces). The *System.IO* namespace—which is contained in the mscorlib.dll assembly—is one of the most widely used namespaces and contains many functions. We use it later in this chapter to demonstrate file I/O.

Figure 6.2 The *System.IO* Namespace Being Imported



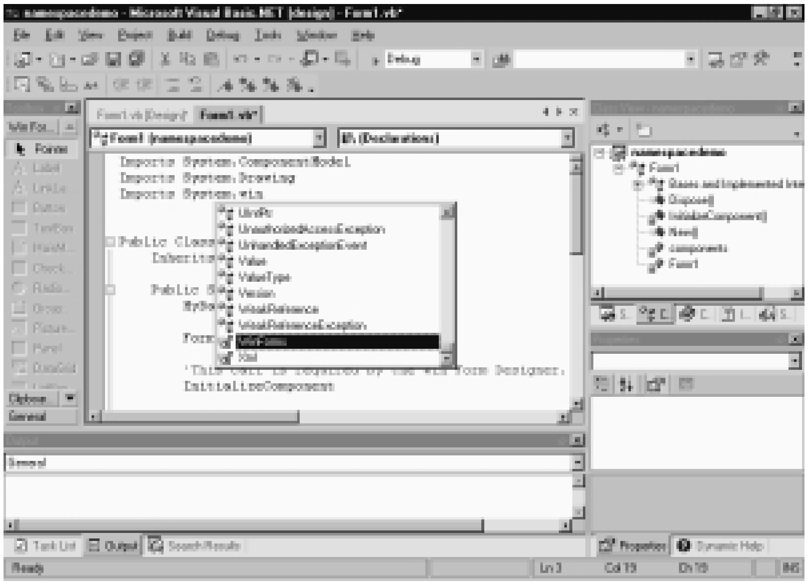
```
Imports System.ComponentModel
Imports System.Drawing
Imports System.WinForms
Imports System.I
```

Another commonly used namespace is the *System.Winforms* namespace. This namespace contains classes, interfaces, structures, delegates, enumerations, and methods for creating Windows-based applications. The form class, the clipboard class, and most of the objects you would find on a Visual Basic 6.0 form are contained in this namespace. Most Visual Basic .NET applications that use forms will require the use of this namespace. Figure 6.3 shows the implementation of the *System.Winforms* namespace with the **Imports** command.

Understanding the Imports Keyword

Imports allows a class to use a namespace that contains functionality. You must place the **Imports** statement at the beginning of a class. **Imports** goes hand in hand with namespaces—it's what allows us to use the namespaces.

Figure 6.3 Importing the *System.Winforms* Namespace



```
Imports System.ComponentModel
Imports System.Drawing
Imports System.Win

Public Class
    Inherits

    Public S
        ByVa

        Form
        'This call is required by the Win Form Designer.
        InitializeComponent
```

Restated, the **Imports** command exposes the functionality in a namespace to a class. The **Imports** command is common throughout Visual Basic .NET applications, so you need to understand how to use it. Here is an example of the **Imports** statement:

```
1      Imports REGTOOL=Microsoft.Win32.Registry
2      Imports System.Drawing

3         Protected Sub Button1_Click(ByVal sender As Object, ByVal e
             As System.EventArgs)
4       Dim img As Imaging.Metafile
' part of the system.drawing namespace

5       img.Save("c:\pic.wmf")

6        REGTOOL.CurrentUser.CreateSubKey("SomeKey")
7     End Sub
```

In the code example, when the command button **button1** is clicked, a subkey is created in the registry called **SomeKey**. The namespace

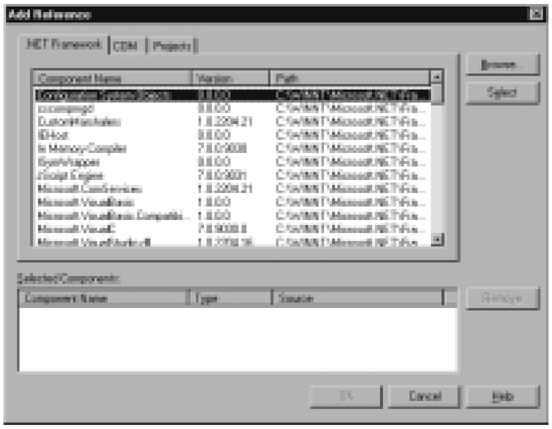
Microsoft.Win32.Registry is imported and aliased to *REGTOOL*. This means that we can use *REGTOOL* throughout the application. You don't need to alias an import; in this case though it will make the name easier to use. *REGTOOL* is much easier to type than *Microsoft.Win32.Registry*. You cannot use the same alias for more than one namespace within a class.

Additionally, you can see in line 4 prefixing an object with its namespace isn't necessary unless a conflict exists. After an object has been imported, all of its methods are available without prefixing them with the namespace. If you have the same method name within more than one namespace, you must prefix the method name with the namespace name. You can also use the **Imports** statement to add namespaces from the current project. If you had code in another class within the project you wanted to use, you could use **Imports** to access that code.

When you are working in the Visual Basic .NET development environment and invoke the **Imports** command, Microsoft's IntelliSense technology will show the available namespaces you can select. If the namespace you are looking for is not in the list, you may need to use the Add Reference dialog box (choose **Add Reference** from the **Project** menu).

The Add Reference dialog box allows you to add references to other objects (see Figure 6.4). These references will add namespaces to the list displayed when you use the **Imports** command. This enables your application to use the methods and objects exposed by the namespace.

Figure 6.4 Adding a Reference to Other Objects



Component Name	Version	Path
[illegible]	[illegible]	[illegible]
cscompmgd	8.0.0.0	C:\WINNT\Microsoft.NET\Fra...
CustomMarshalers	1.0.2204.21	C:\WINNT\Microsoft.NET\Fra...
IEHost	8.0.0.0	C:\WINNT\Microsoft.NET\Fra...
In Memory Compiler	7.0.0.9008	C:\WINNT\Microsoft.NET\Fra...
ISymWrapper	8.0.0.0	C:\WINNT\Microsoft.NET\Fra...
JScript Engine	7.0.0.9001	C:\WINNT\Microsoft.NET\Fra...
Microsoft.ComServices	1.0.2204.21	C:\WINNT\Microsoft.NET\Fra...
Microsoft.VisualBasic	1.0.0.0	C:\WINNT\Microsoft.NET\Fra...
Microsoft.VisualBasic.Compatibi...	1.0.0.0	C:\WINNT\Microsoft.NET\Fra...
Microsoft.VisualC	7.0.9000.0	C:\WINNT\Microsoft.NET\Fra...
Microsoft.VisualStudio.dll	[illegible]	C:\WINNT\Microsoft.NET\Fra...

Component Name	Type	Source

As you can see, this dialog box is somewhat different from the one presented in Visual Basic 6.0. References are broken out by type, .NET Framework, COM, and projects. This is useful when you are dealing with many objects. Most of the objects you use—along with many objects you were familiar with in Visual Basic 6.0—are under the COM tab.

Implementing Interfaces

The use of the **Implements** keyword has changed from Version 6.0 of Visual Basic. The **Implements** statement allows you to use a form of inheritance in Visual Basic .NET. You can implement a class or an interface in Visual Basic .NET with the **Implements** statement. Here is an example:

```
Public function TestFoo (ByVal sWork as String) as Integer
    Implements ImyInterface.Run
```

An interface is comprised of the methods and objects that a class exposes to its consumers. In COM programming, one of the fundamental rules is that after an interface is published you cannot change it. Interfaces are defined in a class with the **Interface** statement; the following code fragment shows an example. Note that you can define subs, functions, events, and properties in interfaces:

```
    Public Interface MyInterface
        Sub subOne()
        Sub SubTwo()
    End Interface
```

An example of an interface to a VB class might be a function as defined in the following code fragment:

```
    Public Sub subOne(ByVal sSoftDrink As String) as Integer
```

This code fragment defines a method called *subOne* that has a parameter of *sSoftdrink*. According to the COM rules, if we change the interface it becomes a new object. This would be contained within a class.

A class can inherit another class's interfaces by using the **Implements** statement. When you implement a class, you have to create all of the methods and properties contained in the implemented class or interface. Failure to do so will cause an error. The only code needed in the interface is the definition for the interface.

The **Inherits** keyword allows Visual Basic .NET to implement true polymorphism. Polymorphism is the ability to redefine methods for base classes. The **Inherits** keyword will allow a class to take on all of the objects completely from a base class. This means that the class will have all of the methods and properties of the base class.

Let's look at an example based on a restaurant, where we have a *person* class and a *customer* class. The *customer* class will inherit the *person* class. The following code shows the attributes of the *person* class:

```
Public Class person
    ' Person class.  This class simulates a customer
    Dim m_name As String    ' Declare local storage
    Dim m_fname As String
    Public Property Last_name(ByVal sNameIn As String) As String
        Get
            Last_name = m_name
        End Get
        Set
            m_name = sNameIn
        End Set
    End Property
    Public Property first_name(ByVal sFNameIn As String) As String
        Get
            first_name = m_fname
        End Get
        Set
            m_fname = sFNameIn
        End Set
    End Property
    Public Sub eat(ByVal sFood As String)
        '
        '


    End Sub
End Class
```

As we can see, the *customer* class has a *first name* property and a *last name* property, as well as a method called *eat*. Note that the syntax and implementation of the *get* and *set* methods have changed from earlier versions of Visual Basic. You can write to and read from the *first name* and *last name* while the *eat* method accepts *food* as a string variable. Now, suppose we are selling soft drinks as well to our customers. We would need a *drink* method (just for our *customer* class). In the following class, we inherit the properties and methods of the *person* class. When we create the *customer* class, we can add a *drink* class:

```
    Public Sub Public Class customer
    Inherits person

    Public Sub drink(ByVal sSoftDrink As String)
         '
         '
         '
    End Sub
End Class
```

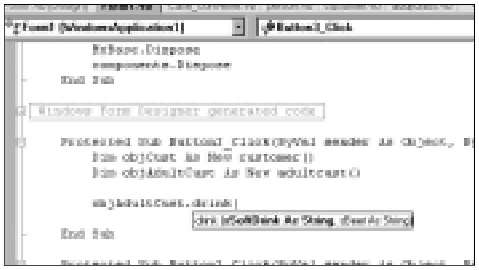
The *customer* class now contains all of the properties and methods of the *person* class, as well as the added *drink* method. This way, the coding is considerably less than it would be to rewrite the methods. Additionally, if a problem occurs with the *customer* class, we need to fix it in only one place, and the fix will propagate to all the classes that inherit the *customer* class.

Now suppose we also sell alcohol to adult customers. In this case, we would not only offer soft drinks but beer as well. We can override the *drink* class and change it. Another observation we can make is that a number of built-in methods are available within the class:

```
1      Public Class adultCustomer
2      Inherits customer

3    Overrides Sub drink(ByVal sSoftDrink As String, ByVal sBeer As
         String)
         '
         '
4    End Sub
5    End Class
```

As you can see, the *drink* sub has changed to include beer for the *adultCustomer* class. It inherits all of the properties and methods of the *customer* class. The original *customer* class contains the *drink* method that accepts only *softdrink*. See the previous code fragment (where the *customer* class is defined). The *adult customer* class contains the method that will accept both *softdrink* and *beer*. See Figure 6.5 for an example. The possibilities of what you can do with the **Inherits** command are pretty endless.

Figure 6.5 The Adult Customer Class



```
        MyBase.Dispose
        components.Dispose
    End Sub

 Windows Form Designer generated code

    Protected Sub Button2_Click(ByVal sender As Object, By
        Dim objCust As New customer()
        Dim objAdultCust As New adultcust()

        objAdultCust.drink(
                  drink(strSoftDrink As String, strBeer As String)
    End Sub
```

Another concept is that of overloaded functions. Overloaded functions are functions with the same name but different data types. When you create an overloaded function interface, you use the following syntax:

```
Overloads Function isZeroOut(ByVal strTest As String) As String
        If strTest = "" Then isZeroOut = "-"
    End Function

    Overloads Function isZeroOut(ByVal iTest As Integer) As String
        If iTest = 0 Then return("-")
    End Function
```

In this example, the same function name is called, but a different set of code executes depending on the type of variables passed into the function. You could also include other data types and have more than two (that is, another set of code if a double was passed in).

Delegates and Events

Delegates can be likened to creating a method in a class whose sole purpose in life is to call another method. Basically, the name of the procedure is passed to the delegate, and it executes the procedure on behalf of the calling procedure. Delegates are called using the *Invoke* method. A delegate is useful for specifying the name of a routine to run at runtime.

Delegates are similar to function pointers in C++. Delegates perform the task of calling methods of objects on your behalf. Delegates are classes that hold references to other classes:

- You can use a delegate to specify an event handler method to invoke when an event occurs. This is one of the most common uses for delegates in Visual Basic .NET. Delegates have signatures and hold references to methods that match their signatures.
- Delegates act as intermediaries when you are programming events to link the callers to the object called.

A declaration for a delegate is shown here:

```
Delegate Function IsZero(ByVal x As Integer) as boolean
```

The following example form and class demonstrate the use of delegates: The following code is used to build the form; the form constructor code has been removed to save space. The form has two buttons on it: One for a command we'll call **BLUE** and one for a command we'll call **RED**. Pressing the buttons causes a message box to be displayed on the screen with a message displaying *blue* or *red*. First, we take a look at the *delegatedemo* class:

```
   Public Class delegatedemo
Delegate Function ColorValue(ByVal sMessage As String) As String


Public Function showcolor(ByVal clr As ColorValue, ByVal sMessage
    As String) As String
   ' invoke
    return(clr.Invoke(sMessage))
End Function
End Class
```

In the *delegatedemo* class, the delegate *ColorValue* is declared and then used by the *ShowColor* function. In this class, the *ColorValue* delegate is invoked and returns the message from the appropriate function, either *redMessage* or *blueMessage*, whichever function address is passed.

The delegate acts as a pointer to the function. It is a multicast delegate because it can point to the *redMessage* function or the *blueMessage* function.

Next, we take a look at the form that uses the *delegatedemo* class.

```
Imports System.ComponentModel
```

```
  Imports System.Drawing
  Imports System.WinForms

 Public Class testform
   Inherits System.WinForms.Form



   Protected Sub cmdBlue_Click(ByVal sender As Object, ByVal e
        As System.EventArgs)
        Dim delMess As New delegatedemo()
        Dim sMessage As String

      delmess.showcolor(AddressOf blueMessage, "Hello World B ")
        msgbox(sMessage)
   End Sub

   Protected Sub cmdRed_Click(ByVal sender As Object, ByVal e
        As System.EventArgs)
        Dim delMess As New delegatedemo()
        Dim sMessage As String

      delmess.showcolor(AddressOf redMessage, "Hello World R")
        msgbox(sMessage)
  End Sub


   Private Function redMessage(ByVal sSmessage As String) As
        String
      ' This function returns sSmessage and 'red'.
        return  "RED " & sSmessage
   End Function

   Private Function blueMessage(ByVal sSmessage As String) As
```

```
          String
18        ' This function returns sSmessage and 'blue'.
19        return  "Blue " & sSmessage
20    End Function

21     End Class
```

This form class consumes the class *delegatedemo*. Based on the button clicked, the delegate in the *delegatedemo* class is invoked and will fire either the function *redMessage* or the function *blueMessage*, depending on the value passed in with the **AddressOf** keyword. What the **AddressOf** keyword does is return the address of the routine called. Consider the following code fragment:

```
delmess.showcolor(AddressOf redMessage, "Hello World R")
```

AddressOf redMessage instructs Visual Basic to return the address of *redMessage* so that we can use it in the routine. This is necessary in order to use delegates properly.

Lines 1 and 2 perform the housekeeping functions that build the form. Some of the initialization code for the form has been removed in order to save space. The button event starts at line 2. This event instantiates the *delegatedemo* class and then calls the *showcolor* method within the *delegatedemo* class. The address of the *blueMessage* function is passed in to the delegate so it can call the function. A string message is also passed in. The value returned is then displayed in a message box. In order to use a delegate, you need to execute its invoke method. This is what causes a delegate to fire. The other button event, the *redMessage* function is similar, except that it passed in the address of the *redMessage* function. Finally, the *redMessage* and *blueMessage* functions accept a string parameter and prefix the string with the respective color.

Simple Delegates

Simple delegates are delegates that keep a list of pointers to one function. A simple delegate serves only one function and calls only one method. It's important for the delegate's signature (function definition) to match that of the called function.

Multicast Delegates

Multicast delegates are delegates that keep a list of pointers to several functions. The preceding example uses multicast delegates. The important thing about multicast delegates is that the interfaces of all the methods that are called by the delegate need to be the same. Sometimes the interfaces for these methods are referred to as the *signature* of the delegate. The signatures all need to match in order for the delegate to function correctly.

Event Programming

An *event* is defined as a message sent by an object that signals the occurrence of an action. The action could be something a user does, like clicking on a button or typing text in a textbox, or it can be *raised* by the code you have written.

One of the issues associated with event programming is that the sender generally doesn't know which object(s) is/are going to consume its events. An event delegate is a class that allows an event sender to be connected with an event handler. An example of an event delegate is shown in this code fragment:

```
Public Delegate Sub MyEventHandler( _
   ByVal caller As Object, _
   ByVal eArgs As EventArgs _
)
```

Handles Keyword

The **Handles** keyword is used when creating an event listener to *glue* the sub to the event:

```
    Private Sub HandleEventFire( _
        ByVal sender As Object, _
        ByVal e As FireEventArgs) _
        As Boolean Handles m_SomeEvent.Fire
```

The **Handles** keyword can accept more than one event—in this way, the event handler can handle more multiple events. The only requirement is that the events it handles have the same parameter list.

Language Interoperability

Language interoperability is the ability of a class to consume other classes written in other languages. One of the benefits to programming with Visual Studio .NET is the ability to work with multiple languages. For example, the following code fragment is part of a C# class that takes an integer and squares it and returns the value (see CD folder Chapter 06/usingcsharpclasses):

```
public static int SquareNumber(int number)
{
     return number * number;
}
```

The following Visual Basic .NET class inherits this C# form class and takes on all of its attributes. The following code fragment consumes the Visual Basic .NET C# class. We removed the initialization logic in order to save space:

```
Imports cSharpDemo.cSharpCode
Public Class cSharpInheritedClass
    ' This class will take on the properties of
    '   the c# form1 class.
    Inherits cSharpDemo.cSharpCode



End Class
```

The class that follows is a C# class. Although the syntax is similar to that of a Visual Basic class, if we look closely, we can see differences in the syntax:

```
Imports System.ComponentModel
Imports System.Drawing
Imports System.WinForms



Public Class cSharpCode
    Inherits System.WinForms.Form

    Public Sub New()
        MyBase.New
        cSharpCode = Me

        'This call is required by the Win Form Designer.
        InitializeComponent

        'TODO: Add any initialization after the InitializeComponent()
            call
    End Sub

    'Form overrides dispose to clean up the component list.
    Overrides Public Sub Dispose()
        MyBase.Dispose
        components.Dispose
    End Sub

    Protected Sub Button1_Click(ByVal sender As Object, ByVal e As
        System.EventArgs)
        Dim csc As New cSharpInheritedClass()

        msgbox(CStr(csc.SquareNumber(2)))
    End Sub

End Class
```

This code displays the square of 2 (4) in a message box. Although this is a simple example, there could be a time when you need a more complex mathematical function written in one language for use in another language.

The ability to share classes between languages is particularly useful if you have a team of programmers working on a project in different languages. The team can easily share functionality across languages, and programmers can program in whatever language they are most comfortable with. Code that is already written in one language need not be rewritten in another language. This is a major boon for code reuse.

File Operations

Since the early days of BASIC, we have been using the *Open* statement for file I/O. Visual Basic 6.0 allowed us to move toward a more object-oriented approach to file I/O by using the Scripting library. The .NET platform takes this a step further by introducing the .NET *System.IO* namespace.

The *System.IO* namespace contains classes that provide developers with methods to read data from and write data to both physical files and data streams, as well as performing file system operations such as obtaining directory listings and monitoring for changes in the file system. Additionally, the ability to read from and write to files asynchronously is a new feature in Visual Basic .NET.

Directory Listing

In earlier versions of Visual Basic, we used the **Dir()** function to retrieve directory listings. This was a bit of a clumsy process that involved seeding a variable with a call to **Dir()** and then entering a loop, calling **Dir()** until it returns an empty string. The old method of obtaining directory listing follows:

```
Dim sFile As String
'Obtain an initial value for sFile
sFile = Dir("C:")
'Loop until Dir() returns empty string
Do Until sFile = vbNullString
   Debug.Print sFile
   sFile = Dir()
Loop
```

Visual Basic 6.0 introduced the Microsoft Scripting Runtime Library. This library gives us access to the Windows file system through a root object, the *Scripting.FileSystemObject* object, and its subordinate objects. This method is much more object-oriented in nature and feels much more fluid, but requires an additional reference and its associated files to be packaged with your application. The syntax and structure of the Scripting Library method of accessing the file system is very similar to that of file system access in Visual Basic .NET. Here is a sample of how you can accomplish the task demonstrated in the previous code example using Visual Basic 6.0 and the Scripting runtime:

```
'*************************************************
'This example requires a reference to be set to the
'Microsoft Scripting Runtime
'*************************************************
Dim oFS As Scripting.FileSystemObject
Dim oFolder As Scripting.Folder
Dim oFile As Scripting.File

'Create the FileSystemObject Object
Set oFS = New Scripting.FileSystemObject
'Get reference to folder through FileSystemObject
Set oFolder = oFS.GetFolder("C:\")
'Enumerate Files
For Each oFile In oFolder.Files
     Debug.Print oFile.Name
Next oFile
```

Now that we've reviewed the methods that have been used in the past to obtain directory listings, we'll look at the preferred method of obtaining directory listings in Visual Basic .NET. As mentioned earlier in the chapter, the *System.IO* namespace provides us with classes that allow us to obtain information about the Windows file system. The specific classes in the *System.IO* namespace that we will use in this demonstration are the *System.IO.Directory* class and the *System.IO.File* class.

You will notice many similarities between directory listing with the Scripting *FileSystemObject* and directory listing using Visual Basic .NET. The primary difference is that the objects needed to perform the task are now native to the development environment. Here is an example of how to perform a directory listing with Visual Basic .NET:

```
Imports SYSTEM.IO
Module Module1
    Sub Main()
     'Obtain reference to Directory
        Dim oDir As Directory = New Directory("C:\")
        Dim oFile As File
        For Each oFile In oDir.GetFiles()
```

```
            debug.WriteLine(oFile.Name)
        Next
    End Sub
End Module
```

Data Files

It is important to consider the type and use of data before storing data in local files on a client. When accessing data files, you will most likely use ADO and databases. The benefits of using databases as opposed to binary files are tremendous, with indexing and sorting and the like built-in. For small amounts of data, such as configuration data, you may want to store information in the registry. From the standpoint of debugging, you may want to store the information locally in a text file—this will allow you to view the information with a simple text editor such as Notepad. This can aid in the debugging process. That being said, you may find that sometimes you need to store information in data files on the client. Data files on the client are usually in binary format. As mentioned earlier in the chapter, the *System.IO* namespace is used to provide us with file access methods. At the top of our module, we need to provide an **Imports** statement for *System.IO* namespace. The following example shows us how to use the *System.IO* namespace to perform file I/O to and from a data file.

The *BinaryReader* and *BinaryWriter* classes may be more familiar to Visual Basic users as *DataReader* and *DataWriter* from the *filesystem* object. Although the names have been changed for the System.IO model, the functionality remains similar. *BinaryReader* is used for reading strings and primitive data types, whereas *BinaryWriter* writes strings and primitives types from a stream. The following example demonstrates reading from a binary file and writing to a binary file:

```
Dim InFile As String
Dim OutFile As String

InFile = "c:\SomeInFile.bin"
OutFile = "c:\someOutFile.Bin"

Dim inf As New System.IO.File(InFile)
Dim outf As New System.IO.File(OutFile)
Dim x As Integer
Dim aRetVal As Integer
```

```
          ' create streams for input and output
9          Dim myInstream As System.IO.FileStream
10          Dim myOutstream As system.IO.FileStream

11          Dim aFoo(100) As System.Byte  ' data to read and write
12          For x = 1 To 99
                 aFoo(x) = x
13          Next

14          Try
15              myInstream = inf.OpenRead ' Open a new stream for input.

16              myOutStream = outf.OpenWrite

17              aRetVal = myoutstream.write(aFoo, 0, 10)
18              myoutstream.Flush()
19              aRetVal = myInstream.Read(aFoo, 0, 10) ' read 10 bytes

20          Catch IOExcep As IO.IOException
21              ' Some kind of error occurred.  Display error message
22              MessageBox.Show(IOExcep.Message)
23          Finally
24              myInStream.Close() ' Close the files
25              myOutStream.Close()
26          End Try
```

In this code fragment, the file name variables are declared and assigned in lines 1 through 4. As we progress to lines 5 and 6, the objects for the files are declared and instantiated. Line 7 declares an integer that will be used later in the load logic. The stream objects for input and output are created and instantiated. Line 8 declares an integer to hold the returned value from the call. Lines 9 and 10 initialize the stream variables. In lines 11 through 13, the variable used to send and receive data is initialized and loaded. We load the variable with numeric data. Lines 15 and 16 open the streams for reading and writing and associate them with the files. Line 17 writes some data to the file and line 18 completes the operation by flushing the buffer. The data written to the file will look like this:

```
1234567891
```

Line 19 reads data from the other file (we assume that the file exists; if it doesn't exist, we would get an error). Assuming we were to use the file we had written previously, the data read from the file will look like this:

```
1234567891
```

Lines 20 through 26 contain exception-handling code, which will handle any exceptions that occur. Lines 24 and 25 the close the streams. Line 26 is the end of the exception-handling code.

That's all there is to reading and writing to files. Additionally, the *filesystem* object provides other methods for file I/O, as do many third-party products. The *filesystem* object has been available for use within Visual Basic since Visual Basic 6.0. This is an object that provides methods for manipulating a computer's files. The following code fragment demonstrates the use of the *filesystem* object:

```
Set fs = CreateObject("Scripting.FileSystemObject")
Set oFile = fs.CreateTextFile("c:\MyTestFile.txt",
True)
oFile.WriteLine("This is a test.")
oFile.Close
```

Text Files

The following example shows how to read and write from a text file. This example opens a file, reads one line at a time from the file, converts the line to uppercase, and then appends the line to the output file. Writing to and reading from text files is a common programming task in Visual Basic.

Text files are generally carriage-return delimited and are limited to ASCII-readable characters. Data files generally contain control characters:

```
Imports System.ComponentModel
Imports System.Drawing
Imports System.WinForms
Imports SYSTEM.IO
Public Class Case_Converter
```

```
Private LinesCounted As Integer = 0
Public Event Status(ByVal LinesCounted As Integer)
Public Event FinishedConverting()

Sub ToUpper(ByVal InFile As String, ByVal OutFile As String)
' first handle files
Dim inf As New SYSTEM.IO.File(InFile)
Dim outf As New SYSTEM.IO.File(OutFile)
' create streams for input and output
Dim myInstream As SYSTEM.IO.StreamReader
Dim myOutstream As SYSTEM.IO.StreamWriter

' temporary string to hold work
Dim mystr As String = " " ' initialize to not empty
Dim OutStr As String = " "

Try
myInstream = inf.OpenText ' Open a new stream for input.
' Do until the stream returns Nothing at end of file.
myOutStream = outf.CreateText

Do Until isnothing(mystr)
mystr = myInstream.ReadLine
' perform conversion
OutStr = ucase(mystr)
myoutstream.WriteLine(OutStr)
LinesCounted += 1 ' Increment line count
' raise an event so the form can monitor progress
RaiseEvent Status(LinesCounted)
Loop
Catch eof As IO.EndOfStreamException
' No action is necessary, the end of the stream has been reached.
Catch IOExcep As IO.IOException
' Some kind of error occurred.
MessageBox.Show(IOExcep.Message)
```

```
29  Finally
30  myInStream.Close() ' Close the files
31  myOutStream.Close()
32  End Try
33  RaiseEvent FinishedConverting()
34 End Sub
35  End Class
```

In this example, we can see that the class *Case_Converter* contains a method called *ToUpper*, which includes two parameters: *InFile* and *OutFile*. Note that importing the *System.IO* namespace (shown in line 4 of the code) is very important. This allows us to use the methods and functions contained in that namespace.

Next, as we progress through the code, line 6 declares a local variable for use within the class. Lines 7 and 8 declare public events that are exploited later in this chapter. It is good programming practice to declare everything in Visual Basic. NET for type safety as well as to help understand the data in a particular variable.

Line 9 is the beginning of the method. When we declare the method, notice that the parameters (*InFile* and *Outfile*) include both a type (*String*) and a calling method (*ByVal*). This is very important in Visual Studio .NET because the default calling type has changed from *Byref* to *Byval*, and you may not get the expected results using the default calling type.

NOTE

Remember to include the *Imports System.IO* command at the beginning of your code when you are working with file I/O. If you don't include it, your code won't compile. This is not so much a problem if you are writing a routine from scratch, because Microsoft's IntelliSense won't work, and you will quickly be aware of the issue. However, look out for if you are pasting code in from another project.

Lines 10 through 15 declare the objects we use to work with file I/O. Some important things to note here are the *System.IO.File* object, the *System.IO.StreamReader* object, and the *System.IO.StreamWriter* object. The *System.IO.File* class is used to help create *FileStream* objects and provides routines for creating, copying, deleting, moving, and opening of files. In lines 10 and 11, file objects are created for the input and output files. These are used later with the *FileStream* objects that are created in lines 12 and 13. The *System.IO .StreamReader* class implements a text reader that will read characters in from the file. This class does the work when it comes to reading data in from the file. The *System.IO.StreamWriter* class implements a text writer that will read characters in from the file.

Appending to Files

Appending to files is pretty simple. If we use the code shown in the text files example, only a minor change is necessary. Appending means that we have an existing file, and we want to add data to the end of it, which is a common programming task. Oftentimes, you'll need to write information out to files for logging, troubleshooting, and saving information.

```
17  myInstream = inf.OpenText ' Open a new stream for input.
' Do until the stream returns Nothing at end of file.
18  myOutStream = outf.CreateText
```

Line 18 would change to:

```
17  myInstream = inf.OpenText ' Open a new stream for input.
' Do until the stream returns Nothing at end of file.
18  myOutStream = outf.AppendText
```

Changing the stream type from *CreateText* to *AppendText* causes text to be appended to the file as opposed to overwriting the file.

Collections

Collections are groups of like objects. Collections are similar to arrays, but they don't have to be redimensioned. You can use the *Add* method to add objects to a collection. Collections take a little more code to create than arrays do, and sometimes accessing a collection can be a bit slower than an array, but they offer significant advantages because a collection is a group of objects whereby an array is a data type. Consider the following code fragment:

```
Dim colPeople As New Collections.StringCollection()
Dim x As Integer
colPeople.Add("Mark")
colPeople.Add("Debbie")
colPeople.Add("Marissa")
```

```
For x = 0 To colPeople.Count - 1
            msgbox(colpeople.Item(x))
      Next x
```

When this code is executed, it displays three message boxes. The first contains "Mark," the second contains "Debbie," and the third contains "Marissa." We can easily remove Debbie from the collection by using the following code:

```
colpeople.RemoveAt(1)
```

This removes Debbie from the collection, and *colPeople.Count* will be equal to 2. The reason we remove at element 1 is because the collection is zero-based, the first item in the collection is item 0. We can also remove Debbie from the collection by using the following code:

```
colpeople.Remove("Debbie")
```

This code produces the same result: It removes **"Debbie"** from the collection. If we run the following code fragment, it displays two message boxes: the first contains "Mark," the second contains "Marissa":

```
For x = 0 To colPeople.Count - 1
            msgbox(colpeople.Item(x))
       Next x
```

Some of the more commonly used methods and properties of collections are shown in Table 6.1.

Table 6.1 Collection Parameters

Parameter	Description
colpeople.Add()	Adds an element to the collection.
colpeople.AddRange()	Copies the elements of a string array to the end of a collection.
colpeople.Clear()	Removes all of the elements from the collection.
colpeople.Contains()	Gets a value indicating whether the collection contains the specified value.
colpeople.CopyTo()	Copies the collection values to a one-dimensional array instance at the specified index.
colpeople.Count()	Returns the number of elements in the collection.

Continued

Table 6.1 Continued

Parameter	Description
colpeople.Equals()	Determines whether the specified object is the same instance as the current object (*colpeople*).
colpeople.GetEnumerator()	Returns an enumerator that can iterate through the collection.
colpeople.GetType()	Gets the type of the object.
colpeople.IndexOf()	Gets the zero-based index of the collection.
colpeople.Insert()	Inserts an object in the middle of the collection (at the specified location).
colpeople.IsReadOnly()	Determines whether the collection is read-only.
colpeople.IsSynchronized()	Determines if the collection is synchronized (thread safe).
colpeople.Item()	The index into the collection.
colpeople.Remove()	Removes an element from a named spot in the collection.
colpeople.RemoveAt()	Removes an element from the collection at a named specified location in the collection.
colpeople.SyncRoot()	Gets the object used to synchronize access to the collection.
colpeople.ToString()	Returns a string representation of the collection.

The Drawing Namespace

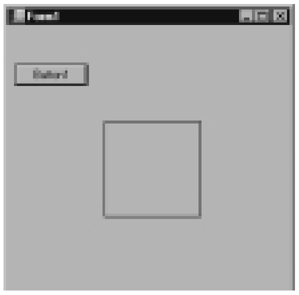
The *System.Drawing* object provides methods for drawing and performing graphic operations. It is a very powerful namespace that exposes quite a few classes, methods, and child namespaces. The following code fragment demonstrates how easy it is to use the *System.Drawing* namespace to draw a simple graphic (a rectangle) on a form (see CD folder Chapter 06/drawingapp):

```
    Protected Sub Button1_Click(ByVal sender As Object, ByVal
    e As System.EventArgs)
         ' create graphics object
1         Dim grp As System.Drawing.Graphics
2         grp = Me.CreateGraphics
```

```
        ' Create pen object
3        Dim oPen As New System.Drawing.Pen(system.Drawing.Color.Black)

        ' Draw up a rectangle
4        grp.DrawRectangle(oPen, 100, 100, 100, 100)


    End Sub
```

In this code fragment, line 1 creates a graphics object. The form in line 2 then instantiates this graphics object. Line 3 creates a pen object to be used for our drawing. The color should be specified in the parameter list. The output of the code is shown in Figure 6.6.

Figure 6.6 The Results of the Drawrectangle Function



As you can see, a rectangle is drawn on the form. If you are familiar with the drawing techniques in earlier versions of Visual Basic, you will realize quickly that the code needs to go in the *form.paint* event. Otherwise, the drawings will be lost whenever the form is repainted. The drawn figure will disappear if you don't put the drawing code (so that it is redrawn) every time the form is repainted. This event fires whenever the form is redrawn.

With the *System.Drawing* object, the ability to create different line graphic images is limitless. This may not be the method of choice when you can choose from so many different commercially available graphics packages, but the *System.Drawing* object is built-in and lightweight. Line graphics, which are usually mathematical in nature, can easily be represented by programmatic functions. You can also use functions for things such as filling in graphics that have been drawn by the drawing objects.

Take some time to review Table 6.2, which shows the different methods and descriptions of the *System.Drawing* object.

Table 6.2 Classes of the *System.Drawing* Namespace

Method	Description
System.Drawing.Bitmap()	The bitmap class encapsulates a GDI+ bitmap.
System.Drawing.Brush()	The brush class is a base class used to fill the interior of shapes such as circles, rectangles, and the like.
System.Drawing.Brushes()	This class contains brushes for all of the standard colors.
System.Drawing.BrushStyle()	This is an enumeration that contains the brush styles. These are typically applied to brush objects.
System.Drawing.Color()	This structure represents a color.
System.Drawing.ColorConverter()	This class is used to convert a color from one data type to another.
System.Drawing.ColorTranslator()	This class translates colors to and from GDI+ objects.
System.Drawing.ContentAlignment()	This enumeration specifies the alignment of content on a drawing object.
System.Drawing.Cursor()	This class contains the image used to create the mouse cursor.
System.Drawing.CursorConverter()	This class is used to convert cursors from one data type to another.
System.Drawing.Cursors()	This class contain the standard cursors.
System.Drawing.Font()	This class defines a format for a font with attributes such as type, size, and style.
System.Drawing.FontConverter()	This class is used to convert fonts from one data type to another.

Continued

Table 6.2 Continued

Method	Description
System.Drawing.FontFamily()	This class is used to class of fonts that have similar attributes but some variation.
System.Drawing.FontStyle()	This enumeration contains style information for fonts.
System.Drawing.Graphics()	This class encapsulates a GDI+ drawing surface.
System.Drawing.Icon()	This class contains the icon class; a small bitmap image.
System.Drawing.IconConverter()	This class is used to convert an icon from one data type to another.
System.Drawing.Image()	This is an abstract class that offers functionality for icons, bitmaps cursors, and metafile classes.
System.Drawing.ImageAnimator()	This class is used to work with images and had members to animate multiple frame images.
System.Drawing.ImageConverter()	This class is used to convert an image from one data type to another.
System.Drawing.ImageFormatConverter()	This class is used to convert colors from one data type to another.
System.Drawing.Pen()	This defines a class to be used for drawing lines and shapes.
System.Drawing.Pens()	This includes pens for all the standard colors.
System.Drawing.PenStyle()	This enumeration defines different styles that a pen can be defined with.
System.Drawing.Point()	This structure represents a pair of X, Y coordinates on a drawing surface.
System.Drawing.PointConverter()	This point converter can be used to convert a point from one data type to another.

Continued

Table 6.2 Continued

Method	Description
System.Drawing.PointF()	This structure represents a pair of X, Y coordinates on a drawing surface.
System.Drawing.PolyFillMode()	This enumeration specifies how overlapping objects will be filled.
System.Drawing.Rectangle()	This structure stores the location and size of a rectangular region.
System.Drawing.RectangleConverter()	This class is used to convert a rectangle from one data type to another.
System.Drawing.Region()	This class shows the interior of a rectangle.
System.Drawing.Size()	This represents the size of a rectangle. Uses an order pair to represent size.
System.Drawing.SizeConverter()	This class is used to convert a size from one data type to another.
System.Drawing.SizeF()	This represents the size of a rectangle. Uses an order pair to represent size.
System.Drawing.SolidBrush()	This class defines a brush that is comprised of one color. Brushes are used to fill objects with a color.
System.Drawing.StringAlignment()	This enumeration is used to define the alignment of a string.
System.Drawing.StringDigitSubstitute()	This enumeration is used to specify information for string digit substitution.
System.Drawing.StringFormat()	This class contains string layout information and manipulation functions.
System.Drawing.StringFormatFlags()	This enumeration is used to specify the layout information describing how a string is to be laid out.

Continued

Table 6.2 Continued

Method	Description
System.Drawing.StringTrimming()	This enumeration is used to specify how to trim characters that don't fit in an object like a rectangle or polygon.
System.Drawing.StringUnit()	This enumeration is used to specify the unit of measurement for a text string.
System.Drawing.SystemBrushes()	This class contains brushes for some of the Windows colors.
System.Drawing.SystemColors()	This class contains Windows systemwide colors.
System.Drawing.SystemIcons()	This class contains Windows systemwide icons.
System.Drawing.SystemPens()	This class contains pens for Windows systemwide colors.
System.Drawing.TextureBrush()	This class contains a brush that can be used to fill the interior of an object, such as a polygon or circle.
System.Drawing.ToolboxBitmapAttribute()	This class defines the images that are associated with a particular component.

Images

A powerful child namespace of the *System.Drawing* namespace is the *System.Drawing.Imaging* namespace. It provides methods for loading, saving, and manipulating image files. Quite a few types of image files are supported by the *images* namespace. Most of the common types (JPEG, GIF, TIF, and the like) are supported.

Table 6.3 Classes of the *System.Drawing.Imaging* Namespace

Method	Description
System.Drawing.Imaging.APMFileHeader()	This class contains objects to Define an APM file header.
System.Drawing.Imaging.BitmapData()	This class contains objects to specify the attributes of a bitmap file.
System.Drawing.Imaging.ColorAdjustType()	This enumeration is used to specify which GDI+ uses color adjustment information.
System.Drawing.Imaging.ColorChannelFlags()	This enumeration is used to specify CMYK channels.
System.Drawing.Imaging.ColorMap()	This class is used to define a map for converting colors.
System.Drawing.Imaging.ColorMapType()	This enumeration is used to specify types of color maps.
System.Drawing.Imaging.ColorMatrix()	This class defines a 5x5 matrix used for coordinates in an RGB color space (defines the mix for colors).
System.Drawing.Imaging.ColorMatrixFlags()	This enumeration is used to specify options for adjusting the color matrix for a GDI+ object.
System.Drawing.Imaging.ColorMode()	This enumeration specifies the two-color modes for color components.
System.Drawing.Imaging.ColorPalette()	This class defines objects for an array to make up a color pallet.
System.Drawing.Imaging .EmfPlusRecordType()	This enumerator defines the methods available in a metafile to read and write graphics commands.
System.Drawing.Imaging.EmfType()	This enumeration is used to specify the metafile type.
System.Drawing.Imaging.Encoder()	This class contains functions that can be used to describe an image and can be passed to an image codec.

Continued

Table 6.3 Continued

Method	Description
System.Drawing.Imaging.EncoderParameter()	This class contains functions that can be used to work with *encoderparameter* types.
System.Drawing.Imaging .EncoderParameters()	This class contains functions which can be used to work with standard encoder parameters.
System.Drawing.Imaging .EncoderParameterValueType()	This enumeration specifies an *encoderparameter* data type.
System.Drawing.Imaging.FrameDimension()	This class contains functions that can be used to work with page frames and dimensions. Some of the shared properties of this class are resolution, page, and time.
System.Drawing.Imaging.ImageAttributes()	This class contains information about how colors are manipu-lated during the rendering process.
System.Drawing.Imaging.ImageCodecFlags()	This enumeration specifies an image codec flag's data type.
System.Drawing.Imaging.ImageFlags()	This enumeration specifies the attributes of the pixel data in an image object.
System.Drawing.Imaging.ImageFormat()	This class contains objects for manipulating the format of an image.
System.Drawing.Imaging.ImageLockMode()	This enumeration is used to specify the lock mode of an image.
System.Drawing.Imaging.Metafile()	This class is used to define a graphic metafile. This can be used to record and play back a metafile.
System.Drawing.Imaging.MetafileFrameUnit()	This enumeration specifies the unit of measurement used by a metafile frame unit.

Continued

Table 6.3 Continued

Method	Description
System.Drawing.Imaging.MetafileHeader()	This class contains information about the header attributes of a metafile.
System.Drawing.Imaging.MetafileType()	This enumeration specifies the type of metafile.
System.Drawing.Imaging.METAHEADER()	This class is used for working with metafile headers.
System.Drawing.Imaging.PaletteFlags()	This enumeration specifies the type of color data in the system palette.
System.Drawing.Imaging.PixelFormat()	This enumeration specifies the format for each pixel.
System.Drawing.Imaging.PropertyItem()	This class is used to encapsulate an item to be used with an image file.

Printing

Printing in Visual Basic .NET is an entirely different concept than in previous versions of Visual Basic, which utilized the intrinsic *Printer* object that was used in a straight line manner. In Visual Basic .NET, printing is handled by classes in the *System.Drawing.Printing* namespace.

To print a document in Visual Basic .NET, we create an instance of the *System.Drawing.Printing.PrintDocument* object (this object must be declared *WithEvents*), set properties to determine the characteristics of our print job, and then call the **Print()** method of the *PrintDocument* object to begin the printing process. As the printing engine prepares to render each page of our document, the *delegate* method that we assigned to handle the *PrintPage* event of the *PrintDocument* object will be called. In this routine is where we need to provide instructions for how the next page is to be printed.

We can best understand this new concept in printing by walking through a simple example. In the example, we create an application that prints the contents of a text file. To minimize overhead and keep the code in our project focused on printing the file, we use a console application for our sample. Create a new console application project named *FilePrinter*. At the top of *Module1*, add the following **Imports** statements:

```
Imports System.Drawing
Imports System.Drawing.Printing
Imports SYSTEM.IO
```

The namespaces we reference in these **Imports** statements contain the classes that allow us to read our text file and print the contents. Now, we need to declare our module level variables. We need a variable to represent the file that we read, and we need a variable to represent the actual print job. Add the following code to the Declarations section of *Module1*:

```
Private WithEvents m_oDoc As PrintDocument
Private m_oFile As StreamReader
```

With our variables declared and our namespaces imported, we are ready to jump into coding the example. We start out by opening our file and starting the print job in *Sub Main*. In the interest of keeping our code as simple as possible, error handling will not be used. It is imperative to make sure that the file you specify for the *StreamReader* object actually exists. Modify *Sub Main* in your project to read as follows:

```
Sub Main()
        m_oFile = file.OpenText("C:\myFile.txt")
        m_oDoc = New PrintDocument()
        m_oDoc.Print()
End Sub
```

At this point, our application opens a text file and starts a print job, but none of the actual printing code is written. The true work in this application is done in the method assigned as the delegate for the *PrintPage* method of *m_oDoc*. We first look at the code for our delegate, and then we discuss the code line by line. Place the following code in *Module1*:

```
Private Sub OnPagePrint(ByVal Sender As Object, _
  ByVal arg As System.Drawing.Printing.PrintPageEventArgs) _
  Handles m_oDoc.PrintPage
    Dim sngCurY As Single
    Dim sngLineHeight As Single
    Dim oFont As Font


    'Set the font for printing
    oFont = New System.Drawing.Font("Courier New", 12)

    'Determine the height of font for printing
    sngLineHeight = oFont.GetHeight(arg.Graphics)

    'Move to top of page
    sngCurY = arg.MarginBounds.Top

    'Make sure data is available in file
    If m_oFile.Peek() <> -1 Then
        Do
            'move to next line on page
            sngCurY += sngLineHeight
            'print the next line of the file
            arg.Graphics.DrawString(m_oFile.ReadLine(), _
              oFont, brushes.Black, _
              arg.MarginBounds.Left, sngCurY)
        Loop Until sngCurY >= arg.MarginBounds.Bottom Or _
         m_oFile.Peek() = -1
    End If

    'determine whether we should continue
    'on the next page
    If m_oFile.Peek <> -1 Then
        arg.HasMorePages = True
    Else
        arg.HasMorePages = False
    End If
End Sub
```

In the first line, we declare the sub *OnPagePrint*. This routine must accept two arguments: the first is of type *Object* and the second is of type *System.Drawing.Printing.PrintPageEventArgs*. The second argument, which we declared as *arg*, provides us with the ability to discover information about the print job and to write our output to the printer. We use the *Handles* keyword to indicate that this method provides the event handling for the *PrintPage* method of the *PrintDocument* object.

On the next several lines, we declare variables that are important for the spacing of our printer output. The single, *sngCurY*, contains our current vertical position on the page. We use this to indicate when we need to go to the next page. The single, *sngLineHeight*, contains the height of the lines on our page. This is calculated by calling the *GetHeight* method of the *Font* object used to render our text. Finally, we declare our *Font* object, *oFont*, which determines the characteristics of the font used in our print job.

Following the declaration of our variables, we instantiate *oFont*, setting the font family to *Courier New* and the font size to *12*. We then use the *GetHeight* method of *oFont* to set *sngLineHeight* to the height of each line in our document. By setting *sngCurY* to the value of *arg.MarginBounds.Top*, we move our starting position to the topmost position in the printable area of our page.

Our next step is to determine whether we have data left inside the file to print. This is done by calling the *Peek()* method of our *StreamReader* object. The *Peek()* method returns the next available character (or –1 if no more characters are available) without actually moving the pointer within the file. If we have data in the file, we proceed into our loop.

Inside the loop, the first action we need to take is to move our vertical position on the paper. We do this by adding the value of *sngLineHeight* to *sngCurY*. This, in effect, brings our "pen" down one line. We call the *DrawString* method of the *System.Drawing* object so that we can access through the variable *arg*. Our loop continues this process until we have reached the bottom of the page or we have run out of data in our file.

This process concludes by determining whether we finished the loop because we ran out of data in the file or because we reached the bottom of the page. If there is still data in our file, then we know that we reached the bottom of the page, and we set the *HasMorePages* property of *arg* to *True* to indicate that the print engine needs to request another page. If not, we set the *HasMorePages* property to *False* to indicate that the print engine can complete the job. Our *OnPagePrint* method is called until the print engine is instructed that there are no more pages to print.

Table 6.4 Classes of the *System.Drawing.Printing* Namespace

Method	Description
System.Drawing.Printing.Duplex()	This enumeration defines the printer's duplex setting.
System.Drawing.Printing .InvalidPrinterException()	This class is the base for the error that gets thrown when trying to use a printer with invalid settings.
System.Drawing.Printing.Margins()	This class is used to specify the margins of a page.
System.Drawing.Printing.MarginsConverter()	This class is used to convert the margins to and from different types.
System.Drawing.Printing.PageSettings()	This class can be used to specify settings for a page.
System.Drawing.Printing.PaperKind()	This enumeration is used to specify the standard paper size.
System.Drawing.Printing.PaperSize()	This class is used to specify the paper size.
System.Drawing.Printing.PaperSource()	This enumeration is used to specify the standard paper source.
System.Drawing.Printing.PaperSourceKind()	This enumeration is used to specify the standard paper source.
System.Drawing.Printing.PreviewPageInfo()	This class contains objects for a single page. This class cannot be inherited.
System.Drawing.Printing .PreviewPrintController()	This class contains objects for a *printcontroller*.
System.Drawing.Printing.PrintController()	This class contains objects that control how a document is printed.
System.Drawing.Printing.PrintDocument()	This class contains objects to send documents to the printer.
System.Drawing.Printing.PrinterResolution()	This class contains objects to represent the printer resolution.

Continued

Table 6.4 Continued

Method	Description
System.Drawing.Printing .PrinterResolutionKind()	This class contains objects to represent the standard printer resolutions.
System.Drawing.Printing.PrinterSettings()	This class contains objects to represent how a document is printed.
System.Drawing.Printing.PrinterUnit()	This enumeration contains representations that are used to define the unit of measure for the printer.
System.Drawing.Printing .PrinterUnitConvert()	This class contains objects that are used for working with the WIN32 printing API.
System.Drawing.Printing.PrintEventArgs()	This class contains objects for printing; used with the *begin-print* and *endprint* events.
System.Drawing.Printing.PrintEventHandler()	This delegate handles the *beginprint*, *endprint*, and *querypagesettings* events
System.Drawing.Printing .PrintPageEventArgs()	This class contains objects for printing; used with the *printpage* event.
System.Drawing.Printing .PrintPageEventHandler()	This delegate handles the *print-page* event.
System.Drawing.Printing.PrintRange()	This enumeration is used to specify the options buttons on the Print dialog box.
System.Drawing.Printing .QueryPageSettingsEventArgs()	This class provides data for the *querypagesettings* event.
System.Drawing.Printing .QueryPageSettingsEventHandler()	This delegate is for the *querypagesettings* event.
System.Drawing.Printing .StandardPrintController()	This class is used to specify a print controller to send data to a printer.

Understanding Free Threading

In order to understand free threading, you need to first understand what a thread is. A *thread* is an independent flow of control that operates within the same address space as other independent flows of controls within a process. It can also be defined as instructions executed by a process. A *process* is typically an application or part of an application running under Windows.

Windows is a pre-emptive multitasking system. The operating system works with threads, and it switches them between processors. In a single processor system, this means at any given instant only one thread can be running. In a multiprocessor system, more than one thread can run at once.

Free threading allows the application you write to perform tasks independently. As a programmer, you can create an independent thread for a process. This can cause an application to be more responsive. Although this is a very powerful addition to Visual Basic, it can also wreak havoc if it is not properly implemented and managed.

Additionally, debugging free-threaded applications can be a nightmare. When more than one process is running, and processes are sharing memory, this can create really complicated bugs. On the positive side, quite a few applications could benefit from the use of free threading, such as the following:

- Applications that have long computational processes.
- Applications that are communicating over the Internet.
- Applications that are performing data access.
- Applications that are using MSMQ (Microsoft Message Queue Services).
- Applications that perform process control.

In order to use free threading in Visual Basic .NET, you must create a thread object using the *System.Threading* namespace. It's a good idea to import the *System.Threading* namespace at the beginning of your class so that you have access to the following:

```
Dim thread As System.Threading.Thread
```

Another implementation of threading is shown in the following code fragment, where a thread is created for a sub called **SomeSub**:

```
Dim othread As New System.Threading.thread(AddressOf SomeSub)
```

The following statement kicks off the thread:

```
othread.Start()
```

Because Visual Basic has made creating threads so easy, you must be careful about synchronization with threads as well as the creation of so many threads that the system performance suffers. If you need to terminate a thread, you can do so by implementing the following code:

```
oThread.Stop
```

SyncLock

You can use the **SyncLock** command to help prevent problems when working with objects in a multithreaded environment. The **SyncLock** command accepts an object as a *key* and locks that object from being accessed by other threads. By *key* we mean a unique identifier or license plate that is used to identify an object . In this way, a function can be marked as off-limits to other threads. The reason why this is so critical is that in the case of multiple threads trying to access an object at the same time, it could cause the system to become unstable or crash. The **SyncLock** command helps us to prevent this from happening.

If another thread were to try to execute the locked block of code, it would be suspended until the **SyncLock** cleared (at the end **SyncLock** statement). **SyncLock** is basically a mutex (mutual exclusion) preventing a critical section of code from being executed by two threads at the same time. The following code fragment demonstrates the **SyncLock** command:

```
Private Sub dosomething()
        SyncLock (button1)
            button1.Text = "Something Else"
        End SyncLock
    End Sub
```

In the procedure, the **SyncLock** command uses the *button1* object as the key. In this case, a key is used as a unique identifier for a block of code. This prevents other threads from accessing *button1* while we were changing the text. If another thread changed the text while we were changing it, a system crash, program error, or some other undesired effect may result.

Good threadsafe code is written by properly using multithreading and paying attention to designing your application properly. Threadsafe code operates properly when more than one thread executes it.

You can use events to simplify thread synchronization. If a process needs to wait for another process to finish, it can wait for the event to raise. This is done my designing the program so that functions wait for other functions to finish using events.

NOTE

Although the **SyncLock** command is helpful, it is not a substitute for writing robust, threadsafe code. When you are developing a multithreaded application, proper design consideration to threading is still required.

An example of this would be asking a coworker to do something and waiting for them to finish (them telling you they are done could be likened to an event) before performing a task that needs the work they were doing.

Summary

In this chapter, we have covered namespaces and how they benefit us by giving us logical groupings of functionality. We have seen how namespaces can make the programming task easier through code reuse, as more and more functions are written. We have also explored the use of the **Imports** keyword. This is the method by which namespaces are made available to a program. Remember to choose wisely when deciding to use an array or a collection. Collections offer many advantages over arrays. We have begun to exploit the power of free threading, and how it can make our applications more responsive. Some of the risks associated with free threading have also been covered. This will be a boon to applications which need to process information asynchronously. The **SyncLock** command was introduced to help manage multithreaded object access.

Solutions Fast Track

Using Modules

- ☑ In Visual Basic .NET, modules are treated like classes.
- ☑ Shared methods take the place of functions and subs in modules.

Utilizing Namespaces

- ☑ Namespaces are one of the key new concepts in Visual Basic .NET. Be sure that you understand how they can help you with code reuse.
- ☑ Take the time to familiarize yourself with Visual Basic .NET's built-in namespaces.

Understanding the Imports Keyword

- ☑ The **Imports** keyword is one of the most important new features of Visual Basic .NET.
- ☑ **Imports** is the method by which we use namespaces.

Implementing Interfaces

☑ Visual inheritance is now supported in Visual Basic .NET, which is a major benefit when creating a lot of similar forms.

☑ The **Implements** command requires all interfaces of the base class to be created in the inherited class.

Delegates and Events

☑ Delegates are similar to pointers in C.

☑ The **Handles** keyword accepts multiple events.

The Advantages of Language Interoperability

☑ Language interoperability can help to ease development on large projects.

☑ It can aid in code reuse across languages.

File Operations

☑ File handling has changed somewhat—it's now stream oriented.

☑ The *System.IO* namespace should be imported when doing file I/O.

☑ Consider the type of data and the use of the data before deciding on a storage type.

Collections

☑ The implementation of collections has changed slightly from earlier versions of Visual Basic, but the principles are the same.

☑ Collections don't require redimensioning like arrays do.

☑ Removing an object from within a collection is easier than it is within an array.

The Drawing Namespace

☑ When drawing, use the *paint* event of your form where appropriate.

Understanding Free Threading

☑ Consider synchronization of processes when creating threads.

☑ Creating too many threads can negatively affect performance instead of speeding things up.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to www.syngress.com/solutions and click on the "Ask the Author" form.

Q: I'm trying to use a namespace that is part of an assembly I know is installed on my system, but it doesn't show up in the Intellisense list. What should I do?

A: If you know the name of the assembly (in this case we'll say it's **MHFUNCS.DLL**, you can go to the Add References selection from the Project menu and add **MHFUNCS.DLL** to the list.

Q: I created a thread for every function in my application, and now it runs really slow. Why didn't free threading speed things up?

A: You created too many threads. Consider creating threads for tasks that take a long time to return control to the calling function.

Q: I create graphics to draw a square on the screen, but when I Alt-Tab to my e-mail program and come back, the graphics are gone. What can I do to prevent this?

A: Add code to draw the graphics (or redraw the graphics) in the form paint event. This will cause the graphics to be redisplayed whenever the form needs to be redrawn.

Q: My application has three threads that all use the same object. I'm not sure why, but sometimes the results are not what I expect from the object. What could be causing this?

A: It sounds like the threads may all be accessing the object at once. Wrapping the code in a **SyncLock** statement would be a good idea. For more information, review the SyncLock section in this chapter.

Chapter 7

Creating Windows Forms

Solutions in this chapter:

Introduction

The .NET Framework provides a common object-based framework for creating forms. All programming languages will use the same forms which are called *Windows Forms*. This gives C++ users an easier way to create forms, and Visual Basic users gain more control over their forms. Windows Forms are classes inherited from the *Forms* class. You can also inherit existing forms to extend or change their functionality. The process of working with forms has undergone some fundamental changes from the process used in previous versions of Visual Basic. We discuss how to create Windows Forms at design time and how to programmatically manipulate them at runtime. It is also important to understand the events for Windows Forms and how they can be utilized. Visual Basic uses the following types of forms: standard forms, MDI forms, and dialog boxes. We discuss working with each of these types of forms.

Another change that will take a little getting used to is adding and using controls on forms. We won't go into much detail on the controls themselves until the next chapter, though. You use menus to allow users easy access to functionality within your program. Users are also accustomed to context menus as well. You will need to understand how to create these menus at design time and manipulate them at runtime. You use toolbars for frequently used commands, and you use status bars to indicate various items of interest to the user.

Binding controls on a form to data sources was a cumbersome task in previous Visual Basic versions. Visual Basic .NET comes with a wizard to ease this process. You can bind controls on a form to a data source in different ways. We also discuss how changes made to controls by the user are updated in the data source. This chapter does not go into detail on ADO and data access. This is covered in Chapter 9.

Windows Forms recognize only Windows Controls. However, you can still use existing ActiveX controls on Windows Forms. Visual Studio .NET includes the Windows Forms ActiveX Control Importer. It converts the type definitions of an ActiveX Control to make it look like a Windows Control to the form.

Application Model

Windows Forms is the new platform for Microsoft Windows–based application development. It is based on the .NET Framework and provides a clear, object-oriented, extensible set of classes that enable you to develop rich Windows-based applications. Additionally, in a multi-tier distributed solution, Windows Forms can act as the local user interface.

A *form* is a representation of a window. Most forms are used to display controls that display information to the user or collect input from the user. A form is an object with properties that define its appearance, methods that define its behavior, and events that define its interaction with the user. You can use the properties, methods, and events of a form to suit your needs. For example, you can change forms to create standard windows, dialog boxes, multiple document interface (MDI) windows, or display surfaces for graphical routines. We take a closer look at some of these applications of forms in the following sections.

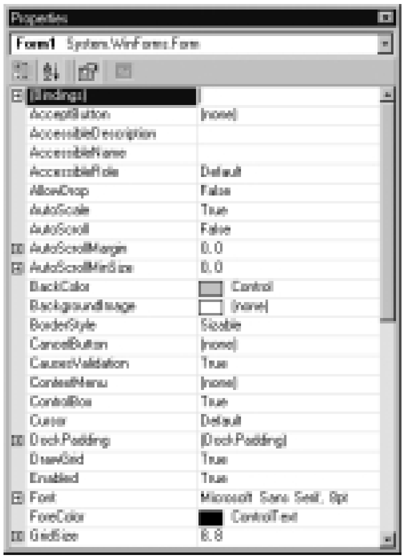
The .NET Framework also allows you to inherit from existing forms to add functionality or modify existing behavior. When you add a form to your project, you can choose whether it inherits from the *Form* class provided by the framework, or from a form you've previously created.

The .NET Framework provides a common, object-oriented framework for all languages in Visual Studio .NET—this framework is Windows Forms. Windows Forms give the Visual Basic programmer control that was previously available only in other Visual Studio languages.

Properties

The properties of a form determine its appearance and behavior. You can use the Properties window to change properties of a form at design-time (see Figure 7.1).

Figure 7.1 The Properties of a Form Displayed in the Properties Window



Form1 System.WinForms.Form

(Bindings)	
AcceptButton	(none)
AccessibleDescription	
AccessibleName	
AccessibleRole	Default
AllowDrop	False
AutoScale	True
AutoScroll	False
AutoScrollMargin	0, 0
AutoScrollMinSize	0, 0
BackColor	▨ Control
BackgroundImage	☐ (none)
BorderStyle	Sizable
CancelButton	(none)
CausesValidation	True
ContextMenu	(none)
ControlBox	True
Cursor	Default
DockPadding	(DockPadding)
DrawGrid	True
Enabled	True
Font	Microsoft Sans Serif, 8pt
ForeColor	■ ControlText
GridSize	8, 8

You can change many properties of a form at runtime as well. Table 7.1 shows properties of a form. You will see how to change these properties both at design-time and runtime in the following sections.

Table 7.1 Form Properties

Property	Description
(Bindings)	This collection holds all the bindings of properties of the form to data sources.
AcceptButton	The accept button of the form. If this is set, the button is clicked whenever the user presses **Enter**.
AccessibleDescription	The description that will be reported to accessibility clients.
AccessibleName	The name that will be reported to accessibility clients.
AccessibleRole	The role that will be reported to accessibility clients.
AllowDrop	Determines if the controls will receive drag-and-drop notifications.
AutoScale	Determines whether the form will automatically scale with the screen font.
AutoScroll	Determines whether scroll bars will automatically appear if controls are placed outside the form's client area.
AutoScrollMargin	The margin around controls during autoscroll.
AutoScrollMinSize	The minimum logical size for the autoscroll region.
BackColor	The background color used to display text and graphics in the form.
BackgroundImage	The background image used for the form.
BorderStyle	Controls the appearance of the border of the form. This will also affect how the caption bar is displayed and what buttons are allowed to appear on it.
CancelButton	The Cancel button of the form. If this is set, the button is clicked whenever the user presses **Esc**.
CausesValidation	Indicates whether the form causes and raises validation events.
ContextMenu	The shortcut menu to display when the user right-clicks the form.
ControlBox	Determines whether the form has a Control/System menu box.

Continued

Table 7.1 Continued

Property	Description
Cursor	The cursor that appears when the mouse passes over the form.
DockPadding	Determines the size of the border for docked controls.
DrawGrid	Indicates whether to draw the positioning grid.
Enabled	Indicates whether the form is enabled.
Font	The font used to display text in the form.
ForeColor	The foreground color used to display text and graphics in the form.
GridSize	Determines the size of the positioning grid.
HelpButton	Determines whether a form has a Help button on the caption bar.
Icon	Indicates the icon for a form. This is displayed in the form's system menu box and when the form is minimized.
IMEMode	Determines the Input Method Editor (IME) status of the form when selected.
IsMDIContainer	Determines whether the form is an MDI container.
KeyPreview	Determines whether keyboard events for controls on the form are registered with the form.
Language	Indicates the current localizable language.
Localizable	Determines if localizable code will be generated for the form.
Location	The position of the top-left corner of the form with respect to its container.
Locked	Determines whether the form can be moved or resized.
MaximizeBox	Determines whether the form has a Maximize box in the upper-right corner of its caption bar.
Menu	The main menu of the form. This should be set to a component of type *MainMenu*.
MinimizeBox	Determines whether a form has a Minimize box in the upper-right corner of its caption bar.
Opacity	Determines how opaque or transparent the form is; 0% is transparent, 100% is opaque.

Continued

Table 7.1 Continued

Property	Description
RightToLeft	Indicates whether the form should draw right-to-left for RTL languages.
ShowInTaskbar	Determines whether the form appears in the Windows taskbar.
Size	The size of the form in pixels.
SizeGripStyle	Determines when the size grip will be displayed for the form.
SnapToGrid	Determines if controls should snap to the positioning grid.
StartPosition	Determines the position of the form when it first appears.
Text	The text contained in the form.
TopMost	Determines whether the form is above all other non-topmost forms, even when deactivated.
TransparencyKey	A color that will appear transparent when painted on the form.
WindowState	Determines the initial visual state of the form.

When you add a form to a project by clicking **Add Windows Form** from the **Project** menu, Visual Basic .NET prompts you for a name for the form. You can also change the name of a form after you have added it to your project. You can change the name of a form by using the Solution Explorer window. Perform the following steps to change the name of a form:

1. From the **View** Menu, click **Solution Explorer**.
2. Right-click the form in **Solution Explorer** and then click **Rename**.
3. Enter a new name for the form, including a **.vb** extension.

You can also use the Properties window to change the name of a form. Follow these steps to change a form's name using the Properties window:

1. From the **View** menu, click **Solution Explorer**.
2. Right-click the form in **Solution Explorer** and then click **Properties**.
3. In **Properties Window**, type a new name in the **FileName** property box, including a **.vb** extension.

You will want to change many properties of a form besides its name to get the appearance that suits your needs. In the following section, we look at how to change key properties of a form and how those changes affect its appearance and behavior.

Manipulating Windows Forms

When you add a Windows form to your project, many of the form's properties are set to commonly used values by default. For example, the *Opacity* property is set to 100% by default because forms are generally opaque and not transparent. Also, the *TopMost* property is set to False by default because inactive forms are not commonly above other forms. Although these values are convenient, they will not always suit your needs. For example, a Help window often allows the user to make it stay on top of other windows. Let's look at how to change the properties of a form.

Properties of Windows Forms

You can change the properties of a form at design time or at runtime. You can virtually avoid compile-time errors by changing properties by point-and-click at design time. Also, changing properties at design time by using the Properties window is often quicker. For example, you could change the font used to display text on a form by following these steps:

1. From the **View** menu, click **Properties Window**.
2. In the **Properties Window**, click **Font** and then click the ellipsis box.
3. In the **Font** dialog box, select the appropriate settings.

Because the Font property consists of many subproperties, quickly changing it at design time is convenient. Similarly, you can use the Properties window to change other properties at design time.

At times, you will want to manipulate a form based on a user input. Like other properties, you can change the caption of a form at runtime. This is handy when the caption of your form includes the text in an editable control on the form. For example, in an employee database application, a form's caption may include the name of an employee and read "John Doe Properties." If the user changes the employee's name using a text box named *txtName*, you want to change the form's caption as shown in the following code:

```
frmEmployee.Text = txtName.Text & "Properties"
```

Likewise, you can change other properties at runtime. You can also use a form's methods at runtime to change its behavior. In the following section, you will see how to do just that.

Methods of Windows Forms

The methods of a form determine its behavior. As you write code in the Code window, methods show differently from properties in the Complete Word box: methods appear as purple diamonds. You can use methods to tailor the behavior of a form to your needs. For example, to make a form invisible you can use the *Hide* method as follows:

```
frmEmployee.Hide()
```

You can also use many other form methods, as shown in Table 7.2.

Table 7.2 Form Methods

Method	Description
Activate	Activates the form and gives it focus.
ActivateControl (inherited from *ContainerControl*)	Activates a specified control.
AddOwnedForm	Adds an owned form to this form.
AdjustFormScrollbars (inherited from *ScrollableControl*)	Adjusts the autoscroll bars on the container based on the current control positions and the control currently selected.
AssignParent (inherited from *RichControl*)	Assigns a new parent control. Sends out the appropriate property change notifications for properties that are affected by the change of parent.
BeginInvoke (inherited from *RichControl*)	Overloaded. Executes a delegate on the thread that owns the control's underlying handle.
BringToFront (inherited from *Control*)	Brings this control to the front of the z-order.
CallWndProc (inherited from *Control*)	Dispatch the method to this window's *wndProc* directly.
Close	Closes the form.
Contains (inherited from *Control*)	Verifies if a control is a child of this control.

Continued

Table 7.2 Continued

Method	Description
CreateAccessibilityInstance (inherited from *RichControl*)	Constructs the new instance of the accessibility object for this control. Subclasses should not call *base.CreateAccessibilityObject*.
CreateControl (inherited from *Control*)	Forces the creation of the control. This includes the creation of the handle, and any child controls.
CreateControlsInstance (inherited from *RichControl*)	Constructs the new instance of the Controls collection objects. Subclasses should not call *base*.*CreateControlsInstance*.
CreateGraphics (inherited from *RichControl*)	Overloaded. Creates the Graphics object for the control.
CreateHandle (inherited from *Control*)	Creates a handle for this control. This method should not be called; it is only called by the .NET Framework. Inheriting classes should always call *base*.*createHandle* when overriding this method.
DefWndProc (inherited from *Control*)	Sends the message to the default window *proc*.
DestroyHandle (inherited from *RichControl*)	Destroys the handle associated with this control.
Dispose (inherited from *ContainerControl*)	Disposes of the resources (other than memory) used by the *ContainerControl*.
DoDragDrop (inherited from *RichControl*)	Begins a drag operation. The *allowedEffects* determine which drag operations can occur. If the drag operation needs to interact with applications in another process, data should either be a base managed class (String, Bitmap, or Metafile) or some Object that implements *System*.*ComponentModel*.*IPersistable*. Data can also be any Object that implements *System*.*WinForms*.*IDataObject*.
EndInvoke (inherited from *RichControl*)	Retrieves the return value of the asynchronous operation represented by the *IAsyncResult* interface passed. If the asynchronous operation has not been completed, this function will block until the result is available.
Equals (inherited from *Object*)	Determines whether the specified *Object* is the same instance as the current *Object*.

Continued

Table 7.2 Continued

Method	Description
Finalize (inherited from *Object*)	Allows an *Object* to attempt to free resources and perform other cleanup operations before the Garbage Collector (GC) reclaims the *Object*. This method may be ignored by the Common Language Runtime; therefore, necessary cleanup operations should be done elsewhere.
FindForm (inherited from *RichControl*)	Retrieves the form that this control is on. The control's parent may not be the same as the form.
Focus (inherited from *Control*)	Sets focus to the control. Attempts to set focus to this control.
GetChildAtPoint (inherited from *Control*)	Retrieves the child control that is located at the specified client coordinates.
GetContainer (inherited from *MarshalByRefComponent*)	Gets the container for the component.
GetContainerControl (inherited from *Control*)	Returns the closest *ContainerControl* in the control's chain of parent controls and forms.
GetDesignMode (inherited from *MarshalByRefComponent*)	Gets a value indicating whether the component is currently in design mode.
GetHashCode (inherited from *Object*)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetLifetimeService (inherited from *MarshalByRefObject*)	This method is used to return a lifetime service object that is used to control the lifetime policy to the object. For the default Lifetime service, this will be an object of type ILease.
GetNextControl (inherited from *Control*)	Retrieves the next control in the tab order of child controls.
GetServiceObject (inherited from *MarshalByRefComponent*)	Gets the implementer of the *IServiceObjectProvider*.
GetStyle (inherited from *Control*)	Retrieves the current value of the specified bit in the control's style. NOTE: This is a control style, not the Win32 style of the *hwnd*.
GetType (inherited from *Object*)	Gets the *Type* of the *Object*.

Continued

Table 7.2 Continued

Method	Description
Hide (inherited from *Control*)	Hides the control by setting the visible property to false.
InitializeLifetimeService (inherited from *MarshalByRefObject*)	Object can provide their own lease and so control their own lifetime. They do this by overriding the *InitializeLifetimeService* method provided on *MarshalByRefObject*.
InitLayout (inherited from *RichControl*)	Called after the control has been added to another container.
Invalidate (inherited from *Control*)	Overloaded. Invalidates a specific region of the control and causes a paint message to be sent to the control.
Invoke (inherited from *RichControl*)	Overloaded. Executes a delegate on the thread that owns this control's underlying window handle.
InvokeGotFocus (inherited from *Control*)	Raises the *GotFocus* event.
InvokeLostFocus (inherited from *Control*)	Raises the *LostFocus* event.
InvokeOnClick (inherited) from *Control*	Raises the *Click* event.
InvokePaint (inherited from *RichControl*)	Raises the *Paint* event for a specific control.
InvokePaintBackground (inherited from) *RichControl*	Raises the *PaintBackground* event for a specific control.
IsInputChar (inherited from *Control*)	Determines if *charCode* is an input character that the control wants.
IsInputKey (inherited from *Control*)	Determines if *keyData* is an input key that the control wants.
LayoutMDI	Arranges the Multiple Document Interface (MDI) child forms of this form.
MemberwiseClone (inherited from *Object*)	Creates a shallow copy of the current *Object*.
OnChangeUICues (inherited from *Control*)	Raises the *ChangeUICues* event.
OnClick (inherited from *Control*)	Raises the *Click* event.

Continued

Table 7.2 Continued

Method	Description
OnClosed	Raises the *Closed* event.
OnClosing	Raises the *Closing* event.
OnControlAdded (inherited from *Control*)	Raises the *ControlAdded* event.
OnControlRemoved (inherited from *Control*)	Raises the *ControlRemoved* event.
OnCreateControl (inherited from *Control*)	Called when the control is first created.
OnDeactivate	Raises the *Deactivate* event.
OnDoubleClick (inherited from *Control*)	Raises the *DoubleClick* event.
OnDragDrop (inherited from *RichControl*)	Inheriting classes should override this method to handle this event. Call *base*.*onDragDrop* to send this event to any registered event listeners.
OnDragEnter (inherited from *RichControl*)	Inheriting classes should override this method to handle this event. Call *base*.*onDragEnter* to send this event to any registered event listeners.
OnDragLeave (inherited from *RichControl*)	Inheriting classes should override this method to handle this event. Call *base*.*onDragLeave* to send this event to any registered event listeners.
OnDragOver (inherited from *RichControl*)	Inheriting classes should override this method to handle this event. Call *base*.*onDragOver* to send this event to any registered event listeners.
OnEnter (inherited from *Control*)	Raises the *Enter* event.
OnGiveFeedback (inherited from *RichControl*)	Inheriting classes should override this method to handle this event. Call *base*.*onGiveFeedback* to send this event to any registered event listeners.
OnGotFocus (inherited from *RichControl*)	Raises the *GotFocus* event.
OnHandleCreated (inherited from *RichControl*)	Inheriting classes should override this method to find out when the handle has been created. Call *base*.*OnHandleCreated* first.
OnHandleDestroyed (inherited from *RichControl*)	Inheriting classes should override this method to find out when the handle is about to be destroyed. Call *base*.*OnHandleDestroyed* last.

Continued

Table 7.2 Continued

Method	Description
OnHelpRequested (inherited from *RichControl*)	Inheriting classes should override this method to handle this event. Call *base.onHelp* to send this event to any registered event listeners.
OnInputLangChange	Raises the *InputLangChange* event.
OnInputLangChangeRequest	Raises the *InputLangChangeRequest* event.
OnInvalidated (inherited from *RichControl*)	Inheriting classes should override this method to handle this event. Call *base.OnInvalidate* to send this event to any registered event listeners.
OnKeyDown (inherited from *Control*)	Raises the *KeyDown* event.
OnKeyPress (inherited from *Control*)	Raises the *KeyPress* event.
OnKeyUp (inherited from *Control*)	Raises the *KeyUp* event.
OnLayout (inherited from *RichControl*)	Core layout logic. Inheriting controls should override this function to do any custom layout logic. It is not necessary to call *base.layoutCore*, however for normal docking and anchoring functions to work, *base.layoutCore* must be called.
OnLeave (inherited from *Control*)	Raises the *Leave* event.
OnLostFocus (inherited from *RichControl*)	Raises the *LostFocus* event.
OnMDIChildActivate	Raises the *MdiChildActivate* event.
OnMenuComplete	Raises the *MenuComplete* event.
OnMenuStart	Raises the *MenuStart* event.
OnMouseDown (inherited from *Control*)	Raises the *MouseDown* event.
OnMouseEnter (inherited from *Control*)	Raises the *MouseEnter* event.
OnMouseHover (inherited from *Control*)	Raises the *MouseHover* event.
OnMouseLeave (inherited from *Control*)	Raises the *MouseLeave* event.
OnMouseMove (inherited from *Control*)	Raises the *MouseMove* event.

Continued

Table 7.2 Continued

Method	Description
OnMouseUp (inherited from *Control*)	Raises the *MouseUp* event.
OnMouseWheel (inherited from *Control*)	Raises the *MouseWheel* event.
OnMove (inherited from *Control*)	Raises the *Move* event.
OnPaint (inherited from *RichControl*)	Inheriting classes should override this method to handle this event. Call *base*.*onPaint* to send this event to any registered event listeners.
OnPaintBackground (*inherited* from *RichControl*)	Inheriting classes should override this method to handle the erase background request from windows. It is not necessary to call *base*.*onPaintBackground*, however if you do not want the default Windows behavior you must set *event*.*handled* to true.
OnParentPropertyChanged (inherited from *RichControl*)	This method is called by the parent control when any property changes on the parent. This can be overridden by inheriting classes; however, they must call *base*.*OnParentPropertyChanged*.
OnPropertyChanged (inherited from *RichControl*)	Occurs when *AccessibleObject* is providing help to accessibility applications.
OnQueryContinueDrag (inherited from *RichControl*)	Inheriting classes should override this method to handle this event. Call *base*.*onQueryContinueDrag* to send this event to any registered event listeners.
OnResize (inherited from *Control*)	Raises the *Resize* event.
OnValidated (inherited from *Control*)	Raises the *Validated* event.
OnValidating (inherited from Control)	Raises the *Validating* event.
ParentChanged (inherited from Control)	Called by the .NET Framework after a control's parent changes. This allows (for example) child controls to automatically hook events on their parent, giving better encapsulation.
PerformLayout (inherited from Control)	Overloaded. Forces the control to apply layout logic to child controls.

Continued

Table 7.2 Continued

Method	Description
PointToClient	Gets the client coordinates for a specified location.
PointToScreen	Gets the screen coordinates for a specified location.
PreProcessMessage (inherited from *Control*)	Called by the application's message loop to preprocess input messages before they are dispatched.
ProcessCmdKey (inherited from *RichControl*)	Processes a command key.
ProcessDialogChar (inherited from *Control*)	Processes a dialog character.
ProcessDialogKey (inherited from *Control*)	Processes a dialog key.
ProcessKeyEventArgs (inherited from *Control*)	Processes a key message.
ProcessKeyPreview (inherited from *Control*)	Previews a keyboard message.
ProcessMnemonic (inherited from *Control*)	Processes a mnemonic character.
ProcessTabKey (inherited from *ContainerControl*)	Selects the next available control and makes it the active control.
RaiseDragEventArgs (inherited from *RichControl*)	Raises the event associated with key with the event data of e and a sender of this control.
RaiseKeyEventArgs (inherited from *Control*)	Raises the event associated with key with the event data of e and a sender of this control.
RaiseMouseEventArgs (inherited from *Control*)	Raises the event associated with key with the event data of e and a sender of this control.
RaisePaintEventArgs (inherited from *RichControl*)	Raises the event associated with key with the event data of e and a sender of this control.
RaisePropertyChangedEvent (inherited from *Control*)	Raises the property changed event. This creates the needed event data and then calls *OnPropertyChanged*.
RecreateHandle (inherited from *Control*)	Forces the recreation of the handle for this control. Inheriting controls must call *base.recreateHandle*.

Continued

Table 7.2 Continued

Method	Description
RectangleToClient	Gets the client coordinates and size of a specified rectangle.
RectangleToScreen	Gets the client coordinates and size for a specified rectangle.
Refresh (inherited from *Control*)	Forces the control to invalidate and immediately repaint itself and any children.
RemoveOwnedForm	Removes a form from the list of owned forms. Also sets the owner of the removed form to a null reference (in Visual Basic Nothing).
ResetBackColor (inherited from *RichControl*)	Resets the back color to be based on the parent's back color.
ResetBindings (inherited from *RichControl*)	Resets the *DataBindings* property to its default value.
ResetCursor (inherited from *RichControl*)	Resets the *Cursor* property to its default value.
ResetFont (inherited from *RichControl*)	Resets the font to be based on the parent's font.
ResetForeColor (inherited from *RichControl*)	Resets the fore color to be based on the parent's fore color.
ResetRightToLeft (inherited from *RichControl*)	Resets *RightToLeft* to be the default.
ResetText (inherited from *Control*)	Resets the text to its default value.
ResumeLayout (inherited from *Control*)	Overloaded. Resumes normal layout logic.
ResumeLayout (inherited from *Control*)	Overloaded. Resumes normal layout logic.
RTLTranslateAlignment (inherited from *RichControl*)	Overloaded Converts the current alignment to the appropriate alignment to support right-to-left text.
RTLTranslateContent (inherited from *RichControl*)	[Overloaded. Converts the current alignment to the appropriate alignment to support right-to-left text.
RTLTranslateHorizontal (inherited from *RichControl*)	Converts the specified *HorizontalAlignment* to the appropriate *HorizontalAlignment* to support right-to-left text.

Continued

Table 7.2 Continued

Method	Description
RTLTranslateLeftRight (inherited from *RichControl*)	Converts the specified *LeftRightAlignment* to the appropriate *LeftRightAlignment* to support right-to-left text.
Scale (inherited from *Control*)	Overloaded. Scales the control and any child controls.
ScaleCore (inherited from *Control*)	Performs the work of scaling the entire control and any child controls.
Select (inherited from *Control*)	Activates this control.
SelectNextControl (inherited from *Control*)	Selects the next control following ctl.
SendMessage (inherited from *Control*)	Overloaded. Sends a Win32 message to the control.
SendMessage (inherited from *Control*)	Overloaded. Sends a Win32 message to the control.
SendToBack (inherited from *Control*)	Sends this control to the back of the z-order.
SetAutoScrollMargin (inherited from *ScrollableControl*)	Sets the size of the auto-scroll margins.
SetBounds (inherited from *Control*)	Overloaded. Sets the bounds of the control.
SetBoundsCore (inherited from *RichControl*)	Performs the work of setting the bounds of the control.
SetClientSizeCore (inherited from *Control*)	Performs the work of setting the size of the client area of the control.
SetDesktopBounds	Sets the bounds of the form in desktop coordinates.
SetDesktopLocation	Sets the location of the form in desktop coordinates.
SetLocation (inherited from *Control*)	Sets the location of this control.
SetNewControls	Arranges an array of controls on a form.
SetSize (inherited from *Control*)	Sets the size of this control.

Continued

Table 7.2 Continued

Method	Description
SetStyle (inherited from *Control*)	Sets the current value of the specified bit in the control's style. NOTE: This is control style, not the Win32 style of the *hwnd*.
ShouldPersistAutoScrollMargin (inherited from *ScrollableControl*)	Indicates whether the *AutoScrollMargin* property should be persisted.
ShouldPersistAutoScrollMinSize (inherited from *ScrollableControl*)	Indicates whether the *AutoScrollMinSize* property should be persisted.
ShouldPersistAutoScrollPosition (inherited from *ScrollableControl*)	Indicates whether the *AutoScrollPosition* property should be persisted.
ShouldPersistBackColor	Indicates whether the *BackColor* property should be persisted.
ShouldPersistBindings (inherited from RichControl)	Indicates whether bindings should be persisted.
ShouldPersistCursor (inherited from *RichControl*)	Returns true if the cursor should be persisted in code gen.
ShouldPersistFont (inherited from *RichControl*)	Returns true if the font should be persisted in code gen.
ShouldPersistForeColor	Indicates whether the *ForeColor* property should be persisted.
ShouldPersistIcon	Indicates whether the Icon property should be persisted.
ShouldPersistLocation (inherited from *Control*)	Determines if the *Location* property needs to be persisted.
ShouldPersistRightToLeft (inherited from *RichControl*)	Returns true if the *RightToLeft* should be persisted in code gen.
ShouldPersistSize (inherited from *Control*)	Determines if the *Size* property needs to be persisted.
ShouldPersistText (inherited from *Control*)	Determines if the *Text* property needs to be persisted.
ShouldPersistTransparencyKey	Indicates whether the *TransparencyKey* property should be persisted.
Show (inherited from *Control*)	Makes the control display by setting the *visible* property to true.

Continued

Table 7.2 Continued

Method	Description
ShowDialog	Overloaded. Displays this form as a modal dialog box.
SuspendLayout (inherited from *Control*)	Suspends the layout logic for the control.
ToString (inherited from *Object*)	Returns a String that represents the current *Object*.
Update (inherited from Control)	Forces the control to paint any currently invalid areas.
UpdateBounds (inherited from *Control*)	Overloaded. Updates the bounds of the control.
UpdateStyles (inherited from *Control*)	Forces styles to be reapplied to the handle. This function will call *CreateParams* to get the styles to apply.
UpdateZOrder (inherited from *Control*)	Updates this control in its parent's z-order.
Validate (inherited from *ContainerControl*)	Validates the last unvalidated control and its ancestors up through, but not including, the current control.
WndProc	Processes Windows messages.
WndProcException (inherited from RichControl)	Processes Windows exceptions.

Similarly, you can use other methods to achieve the behavior that you need. In the following sections, we will look how to create forms that have a specific application.

Creating Windows Forms

The form is the primary vehicle for user interaction within a Windows-based application. You can combine controls and code to collect information from the user and respond to it, work with data stores, and query and write to the Registry and file system on the user's computer. To achieve these results, Visual Basic .NET allows you to create modal, modeless, and top-most forms; we will discuss these form types in the following sections. Visual Basic .NET also allows you to create new instances of a form in different ways. For example, each of the following snippets creates a new instance *frmNewDialog* of a form *frmDialog*:

```
Dim frmNewDialog As frmDialog()
frmNewDialog = New frmDialog()
```

Or:

```
Dim frmNewDialog As New frmDialog()
```

Or:

```
Dim frmNewDialog As frmDialog = New frmDialog()
```

Notice how the *Set* keyword is conspicuously absent. Also notice the parentheses following the form class. In Visual Basic .NET, parentheses are added to the names of forms, classes, and collections—if you omit them, the code editor will add them for you. In the first snippet, declaring the new form *frmNewDialog* as type *frmDialog* allows to you access the properties and methods of the *frmDialog* class using the Complete Word window. However, a new instance of the form is not created until the second statement, which includes the *New* keyword.

As with all objects in the .NET Framework, forms are instances of classes. When you add a form, you can choose whether it inherits from the *Form* class provided by the framework, or from a form you've previously created. The framework also allows you to inherit from existing forms to add functionality or modify existing behavior.

Displaying Modal Forms

A modal form must be closed before you can continue working with the rest of the application. In many Windows-based applications, the user needs to click **OK** or **Cancel** on a dialog box to be able to switch to another form: The dialog box is *modal*. Modal dialog boxes are useful when displaying important messages because they require an acknowledgement from the user. You can display a form as a modal dialog box by using the *ShowDialog* method. The following snippet shows just how:

```
Dim frmProperties As frmDialog = New frmDialog()
frmProperties.ShowDialog()
```

You should be familiar with code execution following the *ShowDialog* method. If a form is shown modally, the code following the *ShowDialog* method does not execute until the form is closed. This differs from code execution if a form is shown as a modeless, as you will see in the next section.

Displaying Modeless Forms

Contrary to a modal form, a *modeless* form allows the user to shift the focus between the form and another form without closing the initial form. Modeless forms are useful when you want the user to refer to one form from another, such as with tool windows and Help windows. However, modeless forms can be a handful because the user can access them in an unpredictable order. This complicates your task as the developer to keep the state of your application consistent. You can easily display a form as a modeless dialog box. To display a modeless dialog box, use the *Show* method, as shown in the following code:

```
Dim frmToolbox As frmDialog = New frmDialog()
frmToolbox.Show()
```

Execution of code following the *Show* method differs from execution of code following the *ShowDialog* method. When a form is shown modelessly, the code following the *Show* method is executed immediately after the form is displayed. When showing multiple forms, at times you may want to keep a form on top of other windows. The following section discusses top-most forms.

Displaying Top-Most Forms

A top-most form stays in front of non-topmost forms even when inactive. In Windows 2000, a top-most form stays in front of other forms within its application. In Windows 98, a top-most form stays in front of all forms in all applications. Top-most forms are useful for creating floating tool windows and Help windows in front of other windows in your application. In a Windows Forms application, you can easily make a form the top-most form by using the *TopMost* property. The following snippet shows how:

```
frmToolbox.TopMost = True
```

After creating a form, you will often want to make stylistic changes to it, such as changing its borders, resizing it, or setting its location. We cover these topics in the following sections.

Changing the Borders of a Form

After you add a form to your project at design time or create a form in code, you can choose from several border styles to determine its look. Apart from controlling the look of the borders of a form, the *BorderStyle* property influences how the caption bar is displayed along with the buttons that appear on it. Moreover, the *BorderStyle* property also affects the resizing behavior of a form. Table 7.3 describes the settings for the *BorderStyle* property.

Table 7.3 Settings for the *BorderStyle* Property

Setting	Description
None	No border or border-related elements. Used for startup forms.
Fixed 3D	Used when 3D border effect is desired. Not resizable. Can include control-menu box, title bar, and Maximize and Minimize buttons on the title bar. Creates a raised border relative to the body of the form.
Fixed Dialog	Used for dialog boxes. Not resizable. Can include control-menu box, title bar, and Maximize and Minimize buttons on the title bar. Creates a recessed border relative to the body of the form.
Fixed Single	Not resizable. Can include control-menu box, title bar, and Maximize and Minimize buttons. Resizable using only Maximize and Minimize buttons. Creates a single line border.
Fixed Tool Window	Used for tool windows. Displays a nonsizable window with a Close button and title bar text in a reduced font size. The form does not appear in the Windows taskbar.
Sizable	(Default) Often used as main window. Resizable. Can include control-menu box, title bar, Maximize button, and Minimize button. Can be resized using control-menu box, Maximize and Minimize buttons on the title bar, or by using the mouse pointer at any edge.
Sizable Tool Window	Used for tool windows. Displays a sizable window with a Close button and title bar text in a reduced font size. The form does not appear in the Windows taskbar.

NOTE

All border styles except the *None* setting feature the Close button on the right-hand side of the title bar.

You can set the border style of a form at design time or at runtime. To set the border style of a form at design time:

1. From the **View** menu, click **Properties Window**.
2. In the **Properties Window**, click **BorderStyle** and select the appropriate border style.

You can also change the border style at runtime using one of the values of the *FormBorderStyle* enumeration. For example, the following sample code set the border style of a form to *FixedDialog*:

```
frmProperties.BorderStyle = FormBorderStyle.FixedDialog
```

If you choose a border style that allows a Maximize and Minimize button in the title bar, you can choose to disable either or both of the buttons. This is handy when you are satisfied with all attributes of a particular border style except the Maximize or Minimize button. You can disable the Maximize and Minimize buttons using the *MaximizeBox* and *MinimizeBox* properties. The following snippet disables the Maximize button of a form:

```
frmProperties.MaximizeBox = False
```

You can disable the Minimize button in similar fashion. The following section discusses resizing forms.

Resizing Forms

As in previous version of Visual Basic, you can use the *Width* and *Height* properties to resize a form. However, Visual Basic .NET also allows you to resize a form by setting its *Size* property. In addition, in Visual Basic .NET you can quickly change form size by increments. Which method you use to resize a form depends largely on your preference.

First, let's look at how to resize a form the old-fashioned way, by setting the *Width* and *Height* properties. This is useful when you want to change either form width or form height, and not both. The following snippet sets the form height to 50 pixels:

```
frmPalette.Height = 50
```

You can achieve the same result by using the *Size* object, which specifies the width and the height in that order. The following code also changes only the form height to 50 pixels. The *frmPalette.Width* parameter maintains the current width of the form:

```
frmPalette.Size = New Size(frmPalette.Width, 50)
```

We have now revealed the true power of the *Size* object: to change both height and width in one statement. For example, you could set the form size to 50 by 50 pixels as follows:

```
frmPalette.Size = New Size(50, 50)
```

In Visual Basic .NET you can also quickly change form size by increments. The following example sets the form height to 50 pixels higher than the current setting:

```
frmPalette.Height += 50
```

WARNING

Do not try to implicitly set the width and height of the *Size* object to quickly change the form size by increments. The following code will **not** change the form size. The *Size* property returns a *Size* structure containing a copy of the form width and height, and the height member of this copied structure is incremented by 50. However, the copied and incremented structure is then discarded:

```
frmPalette.Size.Height += 50
```

Setting Location of Forms

After you create a form, you can specify where it is to be displayed on the computer screen. When a form first appears, the *StartPosition* property determines the position of the form. The default setting of the *StartPosition* is *WindowsDefaultLocation*, which allows the operating system to compute the best location for the form at startup based on the hardware. For example, the user may have a system with multiple monitors or a different screen size and resolution, which can cause the form location to change unpredictably.

NOTE

A form's location as you see it may differ from the form's location as the user sees it.

To an extent, you can control the location of a form using its *Location* property. You can change the x-coordinate and the y-coordinate of a form by using the *Left* and *Top* properties, as in previous versions of Visual Basic. The following example changes the form's y-coordinate to the 100-pixel point:

```
frmPalette.Top = 100
```

In Visual Basic .NET, you can also achieve the same result by using the *Location* object and its X and Y properties. The following snippet also adjusts the form's y-coordinate to the 100-pixel point:

```
frmPalette.Location.Y = 100
```

However, the power of the *Location* object lies in that you can use it to change both the x-coordinate and the y-coordinate of a form simultaneously. The following code adjusts both the x-coordinate and the y-coordinate to the respective 100-pixel points:

```
frmPalette.Location = New Point(100, 100)
```

WARNING

Do not try to implicitly set the x-coordinate and y-coordinate of the *Location* object to quickly change the form's location by increments. The following code will **not** change the form's location. The *Location* property returns a *Location* structure containing a copy of the form's x-coordinate and y-coordinate, and the y-coordinate of this copied structure is incremented by 100. However, the copied and incremented structure is then discarded:

```
frmPalette.Location.Y += 100
```

In Visual Basic .NET, you can also quickly change a form's location by increments. The following example adjusts the form's y-coordinate to 100 pixels farther than the current setting:

```
frmPalette.Top += 100
```

You can use the *DesktopLocation* property instead of the *Location* property to adjust a form's location. The *DesktopLocation* property determines the location of a form relative to the Windows taskbar. This is useful if the taskbar is not automatically hidden and has been docked to the left or top of the monitor, which obscures the desktop coordinates (0, 0). Setting the desktop location of a form to (0, 0) ensures that it appears flush with and not covered by the taskbar, as the following example shows:

```
frmPalette.DesktopLocation = New Point(0, 0)
```

Form Events

Events occur for forms when the user open or closes a form, moves between forms, or interacts with the surface of a form. Events that occur when the user interacts with a form can be triggered by using the mouse or keyboard. The Windows Form framework exposes many events of the *Form* class. Table 7.4 describes these events.

Table 7.4 Form Events

Event	Description
Activated	Occurs when the form is activated in code or by the user.
ChangeUICues (inherited from *Control*)	Occurs when the focus or keyboard or both cues have changed.
Click (inherited from *Control*)	Occurs when the form is clicked.
Closed	Occurs when the form is closed.
Closing	Occurs when the form is closing.
ControlAdded (inherited from *Control*)	Occurs when a new form is added.
ControlRemoved (inherited from *Control*)	Occurs when a form is removed.

Continued

Table 7.4 Continued

Event	Description
Deactivate	Occurs when the form loses focus and is not the active form.
DoubleClick (inherited from *Control*)	Occurs when the form is double-clicked.
DragDrop (inherited from *RichControl*)	Occurs when a drag-and-drop operation is completed.
DragEnter (inherited from *RichControl*)	Occurs when an object is dragged into the control's bounds.
DragLeave (inherited from *RichControl*)	Occurs when an object has been dragged into and out of the control's bounds.
DragOver (inherited from *RichControl*)	Occurs when an object has been dragged over the control's bounds.
Enter (inherited from *Control*)	Occurs when the form is entered.
GiveFeedback (inherited from *RichControl*)	Occurs during a drag operation.
GotFocus (inherited from *Control*)	Occurs when the form receives focus.
HandleCreated (inherited from Control)	Occurs when a handle is created for the form.
HandleDestroyed (inherited from *Control*)	Occurs when the form's handle is destroyed.
HelpRequested (inherited from *RichControl*)	Occurs when the user requests Help for a control.
InputLangChange	Occurs after the input language of the form has changed.
InputLangChangeRequest	Occurs when the user attempts to change the input language for the form.
Invalidated (inherited from *RichControl*)	Occurs when a control's display is updated.
KeyDown (inherited from *Control*)	Occurs when a key is pressed down while the form has focus.
KeyPress (inherited from *Control*)	Occurs when a key is pressed while the form has focus.
KeyUp (inherited from *Control*)	Occurs when a key is released while the form has focus.

Continued

Table 7.4 Continued

Event	Description
Layout (inherited from *Control*)	Occurs when a form's layout properties have been changed.
Leave (inherited from *Control*)	Occurs when the form is left.
LostFocus (inherited from *Control*)	Occurs when the form loses focus.
MDIChildActivate	Occurs when an MDI child form is activated or closed within an MDI application.
MenuComplete	Occurs when a menu in a form loses focus.
MenuStart	Occurs when a menu in a form receives focus.
MouseDown (inherited from *Control*)	Occurs when the mouse pointer is over the form and a mouse button is pressed.
MouseEnter (inherited from *Control*)	Occurs when the mouse pointer enters the form.
MouseHover (inherited from *Control*)	Occurs when the mouse pointer hovers over the form.
MouseLeave (inherited from *Control*)	Occurs when the mouse pointer leaves the form.
MouseMove (inherited from *Control*)	Occurs when the mouse pointer is moved over the form.
MouseUp (inherited from *Control*)	Occurs when the mouse pointer is over the form and a mouse button is released.
MouseWheel (inherited from *Control*)	Occurs when the mouse wheel moves while the form has focus.
Move (inherited from *Control*)	Occurs when the form is moved.
Paint (inherited from *RichControl*)	Occurs when the control is redrawn.
PropertyChanged (inherited from *Control*)	Occurs when a property of the form has changed.
QueryAccessibilityHelp (inherited from *RichControl*)	Occurs when AccessibleObject is providing help to accessibility applications.
QueryContinueDrag (inherited from *RichControl*)	Occurs during a drag-and-drop operation and allows the drag source to determine whether the drag-and-drop operation should be canceled.

Continued

Table 7.4 Continued

Event	Description
Resize (inherited from *Control*)	Occurs when the form is resized.
Validated (inherited from *Control*)	Occurs when the form is done validating.
Validating (inherited from *Control*)	Occurs when the form is validating.

Creating Multiple Document Interface Applications

MDI applications allow simultaneous display of multiple documents, with each document displayed in its own window. MDI applications consist of an MDI parent form and MDI child forms. An MDI application allows you to determine the child form that has the focus. Often MDI applications also allow the user to quickly switch between child windows and to tile, cascade, and arrange child windows. In the following sections, we discuss these topics in detail. First, let's look closely at how to create an MDI parent form.

Creating an MDI Parent Form

The MDI parent form is at the heart of an MDI application. It is the container for the multiple documents—the child forms—within an MDI application. You can use the *IsMDIContainer* property to create an MDI parent form. Follow these steps to create an MDI parent form:

1. Create a new form and open it in the **Code** window.
2. In the constructor for your form, add the following code:

```
Me.IsMDIContainer = True
```

It is convenient for the user to interact with MDI child forms when the parent form is maximized. You can maximize the parent form by setting its *WindowState* property to Maximized.

Creating MDI Child Forms

MDI child forms are forms that operate within an MDI parent form in an MDI application. In an MDI application, these are often the forms with which the user interacts the most. Creating MDI child forms is a step-by-step procedure that we walk through in Exercise 7.1. The following exercise creates MDI child forms via a button on a parent form.

Exercise 7.1 Creating an MDI Child Form

In this exercise, you will create an MDI parent form and an MDI child form. New instances of the child form will be displayed through a button on the parent form.

Creating an MDI Parent Form

1. From the **File** menu, select **New Project**.
2. In the **Visual Basic Projects** list, select the **Windows Application** template and then click **OK**.
3. In the **Properties Window**, set the **IsMDIContainer** property to **True**, and then set the **WindowState** property to **Maximized**.

Creating an MDI Child Form

1. From the **Project** menu, select **Add Windows Form**.
2. In the **Local Project Items** list, select the **Windows Form** template and then click **Open**.

Displaying an MDI Child Form

1. Select the MDI parent form.
2. On the **Toolbox**, select the **Win Forms** tab and double-click the **Button** control to put it on the form.
3. On the MDI parent form, double-click the button. Replace the event handler for the **Click** event with the following code to create a new MDI child form when the button is clicked:

```
Protected Sub Button1_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs)
```

```
        Dim frmNewMDIChild As New Form2()
        frmNewMDIChild.MDIParent = Me
        frmNewMDIChild.Show()
    End Sub
```

The user can now click the button on the MDI parent form to create new child forms. As child forms are created, your task as the developer becomes to manage them. Fortunately, the Windows Forms framework exposes properties and methods to make that an easy task.

Determining the Active MDI Child Form

In an MDI application, the active child form is the child form that has the focus or was most recently active. At times, you will need to identify the active child form. For example, suppose you have a Close menu item in the File menu on your parent form. Because your application can have many instances of the same child form, you need to set apart the child form to be closed: the active child form. You can use the *ActiveForm* property of the parent form to distinguish the active child form. The following code on the parent form closes the active child form:

```
Protected Sub mnuFileClose_Click(ByVal sender as System.Object, _
    ByVal e as System.EventArgs)
    frmMDIParent.ActiveForm.Close()
End Sub
```

MDI applications often offer other ways to interact with child forms as well. In the next section, we look closely at arranging child forms.

Arranging MDI Child Forms

MDI parent forms often sport a Window menu with Arrange, Cascade, Tile Horizontal, and Tile Vertical submenus. The user can click these menus to arrange child forms. You can provide this functionality by using the *LayoutMDI* method of the parent form and the *MDILayout* enumeration. You can choose from four values of the *MDILayout* enumeration, which are described in Table 7.5.

Table 7.5 Settings of the *MDILayout* Enumeration

Setting	Description
ArrangeIcons	Displays child form icons arranged along the lower portion of the parent form.
Cascade	Displays cascading child forms.
TileHorizontal	Displays horizontally tiled child forms.
TileVertical	Displays vertically tiled child forms.

Suppose that you want to tile child forms horizontally when the user clicks the appropriate menu. The following snippet shows just that:

```
Protected Sub mnuWindowTileHorizontal_Click _
(ByVal sender as System.Object, ByVal e as System.EventArgs)
    frmMDIParent.LayoutMDI(MDILayout.TileHorizontal)
End Sub
```

We have now discussed creating and manipulating forms. Generally, forms provide only a framework for the objects with which the user interacts the most: the controls. In the following sections, we discuss adding controls to forms.

Adding Controls to Forms

Most forms contain controls that display information to the user or collect information from the user. These controls are most often added to the form at design time. You can add a control to a form at design time in several ways:

1. From the **View** menu, select **Toolbox**.
2. On the **Toolbox** window, select the **Win Forms** tab.
3. Double-click the appropriate control.

Or:

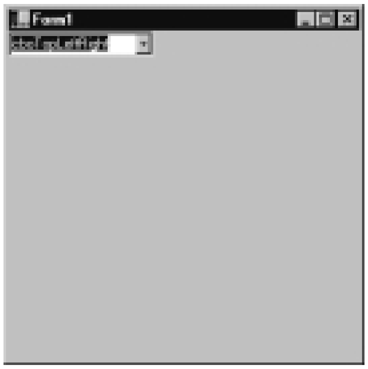
1. Click the appropriate control.
2. On the form, click or drag the mouse.

You can arrange controls on forms in many ways. You can anchor, dock, layer, and position controls on forms. In the following sections, we discuss these different ways to arrange controls on forms.

Anchoring Controls on Forms

The controls on a resizable form should resize and reposition properly when the user resizes the form. In previous versions of Visual Basic, this required extensive coding or a custom component to carry out control resizing and repositioning. In Visual Basic .NET, you can use the *Anchor* property of Windows Forms controls. The *Anchor* property determines to which edges of the container a control is bound. When a control is anchored to an edge, the distance between the control's closest edge and the specified edge will remain constant. Say for example that you have a combo box that is anchored to the top, left, and right edges of a form (see Figure 7.2).

Figure 7.2 A Combo Box Anchored to the Top, Left, and Right Edges of a Form



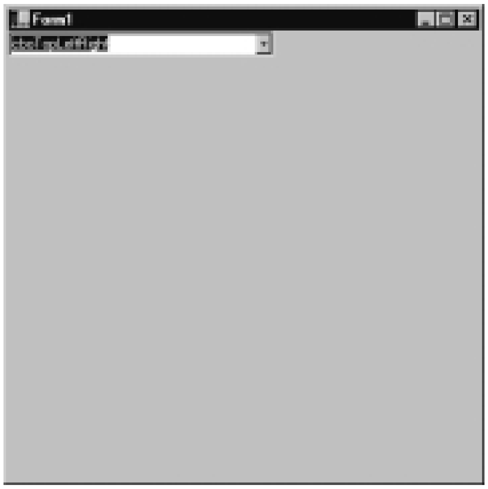
When the user resizes the form, the combo box resizes horizontally to maintain the same distance from the left and right edges of the form—its width increases to maintain the same distance from the right edge. The combo box also repositions itself vertically to maintain the same distance from the top edge of the form (see Figure 7.3). The code would look like the following snippet:

```
cboTopLeftRight.Anchor = AnchorStyles.TopLeftRight
```

NOTE

Windows Forms controls are anchored to the top and left form edges by default.

Figure 7.3 The Combo Box Anchored to the Top, Left, and Right Edges of a Form after Resizing



You can choose from 16 different anchor styles, including None and All. Table 7.6 describes the different control anchor styles. You can also dock controls on forms, which we will discuss in the following section.

Table 7.6 Anchor Styles for Controls

Setting	Description
All	Each edge of the control anchors to the corresponding edge of its container.
Bottom	The control is anchored to the bottom edge of its container.
BottomLeft	The control is anchored to the bottom and left edges of its container.
BottomLeftRight	The control is anchored to the bottom, left, and right edges of its container.
BottomRight	The control is anchored to the bottom and right edges of its container.
Left	The control is anchored to the left edge of its container.
LeftRight	The control is anchored to the left and right edges of its container.
None	The control is not anchored to any edges of its container.
Right	The control is anchored to the right edge of its container.

Continued

Table 7.6 Continued

Setting	Description
Top	The control is anchored to the top edge of its container.
TopBottom	The control is anchored to the top and bottom edges of its container.
TopBottomLeft	The control is anchored to the top, left, and bottom edges of its container.
TopBottomRight	The control is anchored to the top, right, and bottom edges of its container.
TopLeft	The control is anchored to the top and left edges of its container.
TopLeftRight	The control is anchored to the left, top, and right edges of its container.
TopRight	The control is anchored to the top and right edges of its container.

NOTE

Some controls have a limit to their height. If you anchor a control with a height limit to the bottom of its form, the control will not exceed its height limit.

Docking Controls on Forms

At times you may want to dock a control to an edge of its form. For example, status bars are often docked to the bottom form edge. You can dock controls using the *Dock* property. The *Dock* property determines to which form edges a control is docked. Of special note is the Fill setting of the *Dock* property, which makes a control fill its container (either a form or a container control). You can choose from six different dock styles, including None and Fill. Table 7.7 describes the different control dock styles.

Table 7.7 Dock Styles for Controls

Member Name	Description
Bottom	The control's bottom edge is docked to the bottom of its containing control.
Fill	All the control's edges are docked to all edges of its containing control and sized appropriately.
Left	The control's left edge is docked to the left edge of its containing control.
None	The control is not docked.
Right	The control's right edge is docked to the right edge of its containing control.
Top	The control's top edge is docked to the top of its containing control.

Layering Objects on Forms

When your form contains a number of controls, you may need to manipulate their visual layering. You can layer controls visually using their z-order. Z-ordering is the visual layering of controls on a form along its depth, or z-axis. The control at the top of the z-order overlaps all other controls. All other controls overlap the control at the bottom of the z-order. Use the *BringToFront* method to bring a control to the top of the z-order. To send a control to the bottom of the z-order, use the *SendToBack* method of the control as shown in the following example:

```
lblFileSystem.SendToBack()
```

Similarly, you can layer MDI child forms on an MDI parent form using the *BringToFront* and *SendToBack* methods of the child forms.

Positioning Controls on Forms

We have seen how to position forms previously in this chapter. You can position controls on forms in the same fashion. As you can with forms, you can position controls using the *Location* property. The following code sets the location of a text box to the pixel point (50, 50):

```
txtLabel.Location = New Point(50, 50)
```

You can also use the *Left* and *Right* properties or the X and Y properties of the *Location* object to change one control coordinate at a time. Both of the following statements adjust the x-coordinate of the text box to the 50-pixel point:

```
txtLabel.Left = 50
txtLabel.Location.X = 50
```

You can also quickly change a control's location by increments. The following example adjusts the x-coordinate of our text box to 50 pixels farther than the current setting:

```
txtLabel.Left += 50
```

WARNING

Do not try to implicitly set the x-coordinate and y-coordinate of the *Location* object to quickly change the control's location by increments. The following code will **not** change the control's location. The *Location* property returns a *Location* structure containing a copy of the control's x-coordinate and y-coordinate, and the y-coordinate of this copied structure is incremented by 50. However, the copied and incremented structure is then discarded:

```
txtLabel.Location.X += 50
```

Dialog Boxes

Dialog boxes display information to the user and collect information from the user. They are useful because they present visual cues that are familiar to the Windows user. Technically, a dialog box is merely a form with a border style of fixed dialog. As we have seen, this adjusts the appearance of the dialog box in several ways:

- A dialog box is not resizable.
- A dialog box can include a title bar, a control-menu box, and Maximize and Minimize buttons (but they usually do not include the latter three).
- A dialog box has a recessed border relative to the body of the form.

You can use the dialog boxes that are predefined in the .NET Framework or create your own.

Displaying Message Boxes

A message box displays application-related information to the user and collects an acknowledgement or a choice from the user. For example, when you delete a file in Windows Explorer, a message box confirms whether you want to delete the file and collects your choice.

You can display a message box using the *Show* method of the *MessageBox* class. At a minimum, the *Show* method takes a message parameter. The following code displays a message box informing the user of the completion of a backup:

```
Messagebox.Show("The backup of 'My C Drive (C:)' is complete.")
```

Often message boxes collect a choice from the user. The *Show* method returns a value that you can use to determine the user's choice. The following snippet displays a message box confirming the deletion of a file:

```
If Messagebox.Show("Are you sure you want to send 'Error.log' to the " _
    & "Recycle Bin?", "Confirm File Delete", MessageBox.YesNo _
    + MessageBox.IconQuestion) = DialogResult.Yes Then
'Send file to Recycle Bin


End If
```

The .NET Framework includes other preformatted dialog boxes, the likes of which are used throughout Windows. In the next section, we discuss those dialog boxes.

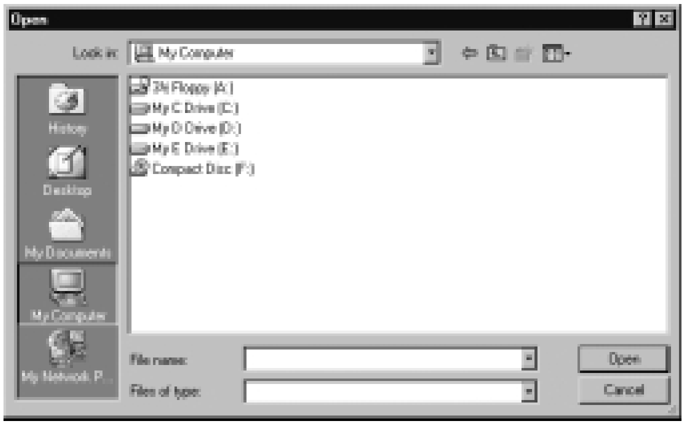
Common Dialog Boxes

At times, you can use preconfigured dialog boxes that are included in the Windows Forms framework in lieu of creating your own. When you use standard Windows dialog boxes, the user can easily recognize the functionality of the dialog box.

The *OpenFileDialog* Control

The Windows Forms *OpenFileDialog* control is the same Open File dialog box that you have used throughout Windows—for example, when opening a document in Microsoft Word. By using this preconfigured dialog box, you can present functionality that your users are already familiar with. By default, the Open File dialog box displays a Look In box, an Outlook bar, and a list box displaying the contents of the current folder. The dialog box also displays a File Name box and a Files Of Type box (see Figure 7.4).

Figure 7.4 The Open File Dialog Box



The Open File dialog box exposes several properties that you can use to write your file-opening logic. For example, you can use the *FileName* property to set the file first shown in the dialog or to check the last file selected by the user. Table 7.8 shows the other properties of the *OpenFileDialog* control.

Table 7.8 Properties of the *OpenFileDialog* Control

Property	Description
(Name)	Indicates the name used in code to identify the dialog.
AddExtension	Controls whether extensions are automatically added to filenames.
CheckFileExists	Checks that the specified file exists before returning from the dialog.
CheckPathExists	Checks that the specified path exists before returning from the dialog.
DefaultExt	The default filename extension. If the user types in a filename, this extension is added at the end of the filename if one isn't specified.

Continued

Table 7.8 Continued

Property	Description
DereferenceLinks	Controls whether shortcuts are dereferenced before returning from the dialog.
FileName	The file first shown in the dialog, or the last one selected by the user.
Filter	The file filters to display in the dialog.
FilterIndex	The index of the file filter selected in the dialog. The first item has an index of 1.
InitialDirectory	The initial directory for the dialog.
Modifiers	Indicates the visibility level of the dialog.
Multiselect	Controls whether multiple files can be selected in the dialog.
ReadOnlyChecked	The state of the read-only check box in the dialog.
RestoreDirectory	Controls whether the dialog restores the current directory before closing.
ShowHelp	Enables the Help button.
ShowReadOnly	Controls whether to show the read-only check box in the dialog.
Title	The string to display in the title bar of the dialog.
ValidateNames	Controls whether or not the dialog ensures that the filenames do not contain invalid characters or sequences.

As with all preconfigured dialog boxes provided by the Windows Forms framework, you can display the Open File dialog box by using the *ShowDialog* method. For example, say that you wanted to display the Open File dialog box and set the file first displayed by the dialog box to *File1.txt*.Your code would look like the following snippet:

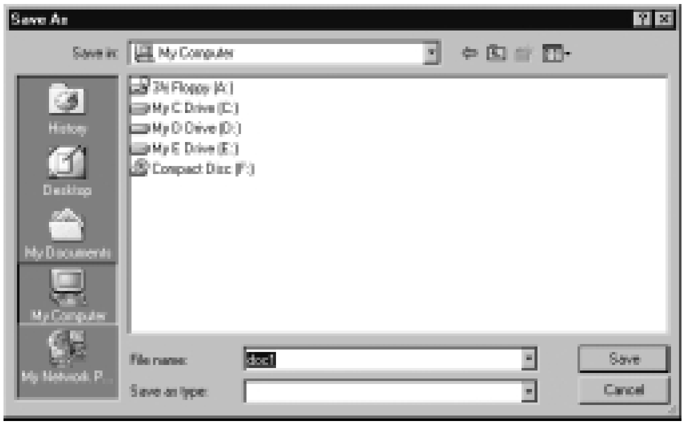
```
With OpenFileDialog1
    .FileName = "File1.txt"
    .ShowDialog()
End With
```

Similarly, you can use the other properties to access functionality provided by the Open File dialog box.You will see more examples as we discuss the other preconfigured dialog boxes. Let's look at the Save File dialog box next.

The *SaveFileDialog* Control

The Save File dialog box is similar to the Open File dialog box. The Save File dialog box allows the user to specify options for saving a file. You have also seen this dialog box throughout Windows—for example, when saving an unsaved document in Microsoft Word. The Save File dialog box displays a Save In box, an Outlook bar, and a list box showing the contents of the current folder. The dialog box also displays a File Name box and a Save As Type box (see Figure 7.5).

Figure 7.5 The Save File Dialog Box



File name: doc1
Save as type:

Like the Open File dialog box, the Save File dialog box also exposes several properties that you can use to write your file-saving logic. Table 7.9 describes the properties of the *SaveFileDialog* control.

Table 7.9 Properties of the SaveFileDialog Control

Property	Method
(Name)	Indicates the name used in code to identify the dialog.
AddExtension	Controls whether extensions are automatically added to filenames.
CheckFileExists	Checks that the specified file exists before returning from the dialog.
CheckPathExists	Checks that the specified path exists before returning from the dialog.
CreatePrompt	Controls whether to prompt the user when a new file is about to be created. It is only applicable if the *ValidateNames* property is set to True.

Continued

Table 7.9 Continued

Property	Method
DefaultExt	The default filename extension. If the user types in a filename, this extension is added at the end of the filename if one is not specified.
DereferenceLinks	Controls whether shortcuts are dereferenced before returning from the dialog.
FileName	The file first shown in the dialog, or the last one selected by the user.
Filter	The file filters to display in the dialog.
FilterIndex	The index of the file filter selected in the dialog. The first item has an index of 1.
InitialDirectory	The initial directory for the dialog.
Modifiers	Indicates the visibility level of the dialog.
OverwritePrompt	Controls whether to prompt the user when an existing file is about to be overwritten. It is only applicable if the *ValidateNames* property is set to True.
RestoreDirectory	Controls whether the dialog restores the current directory before closing.
ShowHelp	Enables the Help button.
Title	The string to display in the title bar of the dialog.
ValidateNames	Controls whether or not the dialog ensures that filenames do not contain invalid characters or sequences.

You can use these properties to write your file-saving logic. For example, say that you wanted to enforce a filename filter in your dialog box to first display only text files with the extension .txt and also allow the user the option to see all files. You can use the *Filter* property to specify the filename filter string, which determines the choices that appear in the Save As Type box. Let's see how this would appear in code:

```
With SaveFileDialog1
    .Filter = "Text files (*.txt)|*.txt|All files (*.*)|*.*"
    .ShowDialog()
End With
```

The first part of the filter displays the *Text files (*.txt)* in the Save As Type box and specifies the mask for these filenames, namely *.txt:

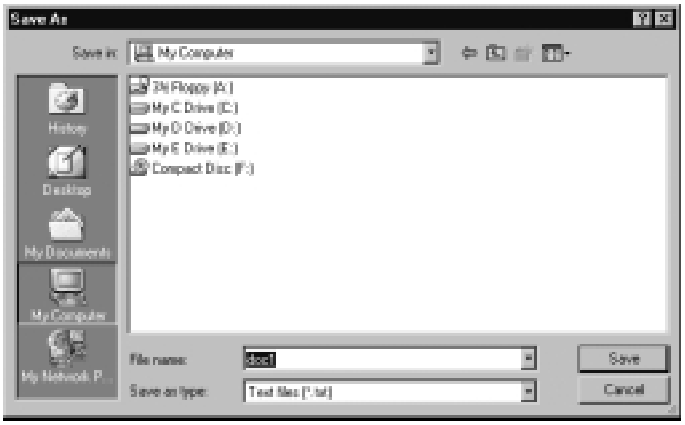
```
Text files (*.txt)|*.txt
```

The second part of the filter displays the text *All files (*.*)* in the Save As Type box and specifies the mask for all filenames, namely *.*:

```
  All files (*.*)|*.*
```

Notice that the two masks are separated by the pipe symbol. The two filename masks appear in the order specified, text files first and all files second (see Figure 7.6).

Figure 7.6 Using the Filter Property of the *SaveFileDialog* Control



File name: doc1

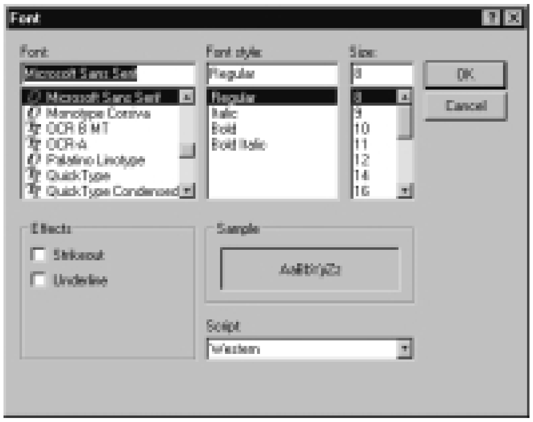
Save as type: Text files (*.txt)

The *OpenFileDialog* control exposes the *Filter* property as well. As you can see, the preconfigured dialog boxes have several properties in common. We discuss how to use more of these properties as we look at the other preconfigured dialog boxes. Let's look at the font dialog box next.

The *FontDialog* Control

The Windows Forms *FontDialog* control is another preconfigured dialog box. The Font dialog box displays the fonts that are installed on the user's computer. The dialog box allows the user to select a font, font style, and size. The user can also select effects such as Strikeout and Underline, and a script. In addition, the Font dialog box displays a sample of how the font will appear (see Figure 7.7).

Figure 7.7 The Font Dialog Box


Font: Microsoft Sans Serif
Font style: Regular
Size: 8
Script: Western

The *FontDialog* control exposes several methods that you can use to dynamically manipulate the dialog box. Table 7.10 displays the properties of the *FontDialog* control.

Table 7.10 Properties of the *FontDialog* Control

Property	Description
(Name)	Indicates the name used in code to identify the dialog.
AllowScriptChange	Controls whether the character set of the font can be changed.
AllowSimulations	Controls whether GDI font simulations are allowed.
AllowVectorFonts	Controls whether vector fonts can be selected.
AllowVerticalFonts	Controls whether vertical fonts can be selected.
Color	The color selected in the dialog.
FixedPitchOnly	Controls whether only fixed-pitch fonts can be selected.
Font	The font selected in the dialog.
FontMustExist	Controls whether to report an error if the selected font does not exist.
MaxSize	The maximum point size that can be selected (or 0 to disable).
MinSize	The minimum point size that can be selected (or 0 to disable).
Modifiers	Indicates the visibility level of the dialog.

Continued

Table 7.10 Continued

Property	Description
ScriptsOnly	Controls whether to exclude OEM and Symbol character sets.
ShowApply	Controls whether to show the Apply button.
ShowColor	Controls whether to show a color choice.
ShowEffects	Controls whether to show the underline, strikeout, and font color selections.
ShowHelp	Controls whether to show the Help button.

You can use these properties to control which buttons and selections are shown on the Font dialog box. For example, the following snippet specifies that the Apply button, a color choice, the underline, strikeout, and color selections, and the Help button be shown. As with other preconfigured dialog boxes, you can display the Font dialog box using the *ShowDialog* method:

```
With FontDialog1
    .ShowApply = True
    .ShowColor = True
    .ShowEffects = True
    .ShowHelp = True
    .ShowDialog()
End With
```

You can see the results of this snippet in Figure 7.8.

The *ColorDialog* Control

The Windows Forms *ColorDialog* control allows the user to select a color from a palette and to add custom colors to that palette. You may have seen it in other Windows applications, such as the *Display* control panel. The color dialog box displays an array of basic colors, an array of custom colors, and a Define Custom Colors button (see Figure 7.9).

Figure 7.8 Using the *ShowApply*, *ShowColor*, *ShowEffects*, and *ShowHelp* Properties of the *FontDialog* Control

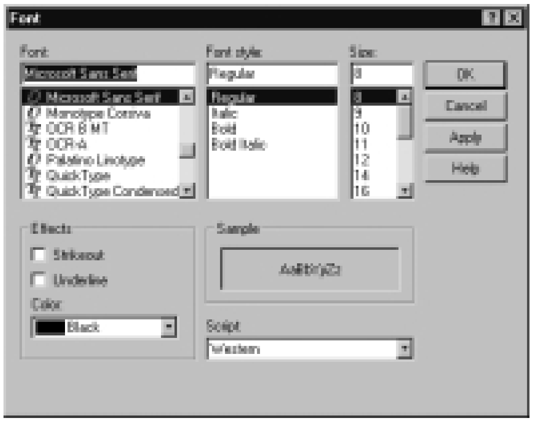
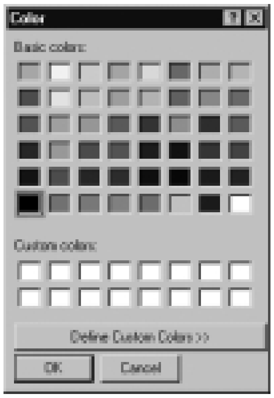


Figure 7.9 The Color Dialog Box



The color dialog box has a unique set of properties. You can use the *Color* property to determine the color the user has selected and then take appropriate action. Other properties of the *ColorDialog* control are described in Table 7.11.

Table 7.11 Properties of the *ColorDialog* Control

Property	Description
(Name)	Indicates the name used in code to identify the dialog.
AllowFullOpen	Enables and disables the Define Custom Colors button.
AnyColor	Controls whether any color can be selected.

Continued

Table 7.11 Continued

Property	Description
Color	The color selected in the dialog.
FullOpen	Controls whether the custom color section of the dialog is initially displayed.
Modifiers	Indicates the visibility level of the dialog.
ShowHelp	Controls whether the Help button is displayed.
SolidColorOnly	Controls whether only solid colors can be selected.

The *PrintDialog* Control

The Windows Forms *PrintDialog* control is another preconfigured dialog box that you can use in lieu of creating your own. The Print dialog box allows the user to select a printer, choose the pages to print, and determine other print-related settings in Windows applications. The dialog box also allows users to print many parts of their documents: print all, print a selected page range, or print a selection.

You can use the properties of the *PrintDialog* control to configure the appearance of your Print dialog box. Table 7.12 describes the properties of the *PrintDialog* control.

Table 7.12 Properties of the *PrintDialog* Control

Property	Description
(Name)	Indicates the name used in code to identify the dialog.
AllowPrintToFile	Enables and disables the Print To File check box.
AllowSelection	Enables and disables the Selection radio button.
AllowSomePages	Enables and disables the Pages radio button.
Document	The *PrintDocument* from which to get printer settings.
Modifiers	Indicates the visibility level of the dialog.
PrintToFile	Controls whether the Print To File check box is checked.
ShowHelp	Controls whether the Help button is displayed.
ShowNetwork	Controls whether the Network button is displayed.

For example, you can use the *AllowPrintToFile* property to enable the Print To File check box. Let's look at how this would appear in code:

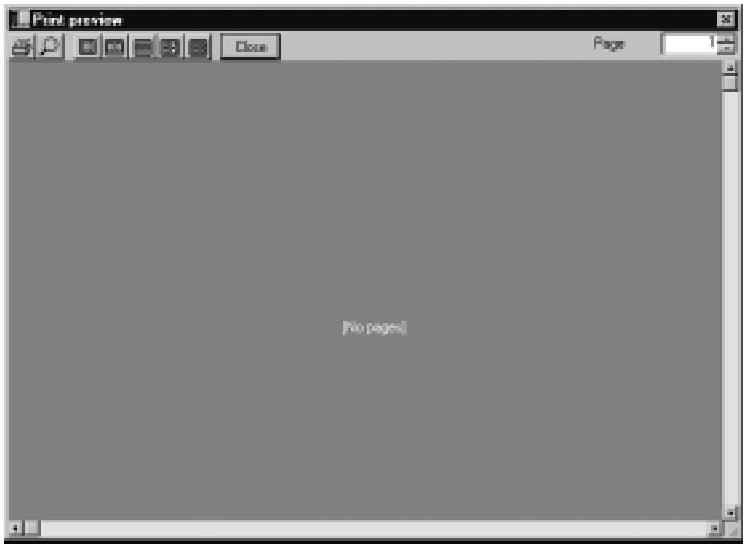
```
With PrintDialog1
     .AllowPrintToFile = True
     .ShowDialog()
End With
```

The Print dialog box is related to the Print Preview dialog box, which we discuss in the next section.

The *PrintPreviewDialog* Control

The *PrintPreviewDialog* control displays how a document will appear when printed. The Print Preview dialog box contains buttons for printing, zooming in, displaying one or multiple pages, and closing the dialog box (see Figure 7.10).

Figure 7.10 The Print Preview Dialog Box



The *PrintPreviewDialog* control is unique in that it contains another control: *PrintPreviewControl*. The contained *PrintPreviewControl* exposes properties of its own, such as the *Columns* and *Rows* properties, which determine the number of pages displayed horizontally and vertically on the control. (You can access the *Columns* property using the syntax *PrintPreviewDialog1.PrintPreviewControl .Columns*.) Because the *PrintPreviewControl* is automatically contained within the *PrintPreviewDialog* control when you add the dialog to your form, you do not have to add the *PrintPreviewControl* to the form. Table 7.13 describes the properties of the *PrintPreviewControl*.

Table 7.13 Properties of the *PrintPreviewDialog* Control

Property	Description
(Bindings)	This collection holds all the bindings of properties of the dialog to data sources.
(Name)	Indicates the name used in code to identify the dialog.
AccessibleDescription	The description that will be reported to accessibility clients.
AccessibleName	The name that will be reported to accessibility clients.
AccessibleRole	The role that will be reported to accessibility clients.
AllowDrop	Determines if the control will receive drag-and-drop notifications.
Anchor	The anchor of the control.
AutoZoom	Determines whether to automatically zoom to fill available space.
BackColor	The background color used to display text and graphics in the control.
BackgroundImage	The background image used for the control.
CausesValidation	Indicates whether the control causes and raises validation events.
Columns	The number of pages across.
ContextMenu	The shortcut menu to display when the user right-clicks the dialog.
Cursor	The cursor that appears when the mouse passes over the dialog.
Dock	The docking location of the dialog, indicating which borders are docked to the container.
Document	The *PrintDocument* to be previewed.
Enabled	Indicates whether the control is enabled.
Font	The font used to display text in the control.
ForeColor	The foreground color used to display text and graphics in the control.
IMEMode	Determines the IME status of the control when selected.
Location	The position of the top-left corner of the control with respect to its container.
Locked	Determines if the user can move or resize the control.

Continued

Table 7.13 Continued

Property	Description
Modifiers	Indicates the visibility level of the control.
RightToLeft	Indicates whether the control should draw right-to-left for RTL languages.
Rows	The number of pages down.
Size	The size of the control in pixels.
StartPage	The first page displayed by the control.
TabIndex	Determines the index in the Tab order that the control will occupy.
TabStop	Indicates whether the user can use the **Tab** key to give focus to the control.
Text	The text contained in the control.
Visible	Determines whether the control is visible or hidden.
Zoom	The magnification applied by the control.

However, like other preconfigured dialog boxes, the *PrintPreviewDialog* control also exposes properties of its own. These properties are described in Table 7.14.

Table 7.14 Properties of the *PrintPreviewDialog* Control

Property	Description
(Bindings)	This collection holds all the bindings of properties of the dialog to data sources.
(Name)	Indicates the name used in code to identify the dialog.
AcceptButton	The accept button of the form. If this is set, the button is clicked whenever the user presses **Enter**.
AccessibleDescription	The description that will be reported to accessibility clients.
AccessibleName	The name that will be reported to accessibility clients.
AccessibleRole	The role that will be reported to accessibility clients.
AllowDrop	Determines if the dialog will receive drag-and-drop notifications.
Anchor	The anchor of the dialog.
AutoScale	If set to True, the dialog will automatically scale with the screen font.

Continued

Table 7.14 Continued

Property	Description
AutoScroll	Determines whether scroll bars will automatically appear if controls are placed outside the dialog's client area.
AutoScrollMargin	The margin around controls during autoscroll.
AutoScrollMinSize	The minimum logical size for the autoscroll region.
BackColor	The background color used to display text and graphics in the dialog.
BackgroundImage	The background image used for the dialog.
BorderStyle	Controls the appearance of the border for the dialog. This will also affect how the caption bar is displayed, and what buttons appear on it.
CancelButton	The cancel button of the dialog. If this is set, the button is clicked whenever the user presses the **Esc** key.
CausesValidation	Indicates whether the dialog causes and raises validation events.
ContextMenu	The shortcut menu to display when the user right-clicks the dialog.
ControlBox	Determines whether the dialog has a Control/System menu box.
Cursor	The cursor that appears when the mouse passes over the dialog.
Dock	The docking location of the dialog, indicating which borders are docked to the container.
DockPadding	Determines the size of the border for docked controls.
Document	The *PrintDocument* to be previewed.
Enabled	Indicates whether the dialog is enabled.
Font	The font used to display text in the dialog.
ForeColor	The foreground color used to display text and graphics in the dialog.
HelpButton	Determines whether the dialog has a Help button on the caption bar.
Icon	Indicates the icon for the dialog. This is displayed in the dialog's System menu box and when the dialog is minimized.

Continued

Table 7.14 Continued

Property	Description
IMEMode	Determines the IME status of the dialog when selected.
IsMDIContainer	Determines whether the dialog is an MDI container.
KeyPreview	Determines whether keyboard events for controls on the dialog are registered with the dialog.
Location	The position of the top-left corner of the dialog with respect to its container.
MaximizeBox	Determines whether the dialog has a Maximize box in the upper-right corner of its caption bar.
Menu	The main menu of the dialog. This should be set to a component of type *MainMenu*.
MinimizeBox	Determines whether the dialog has a Minimize box in the upper-right corner of its caption bar.
Modifiers	Indicates the visibility level of the dialog.
PrintPreviewControl	The *PrintPreviewControl* to use as the dialog's core.
RightToLeft	Indicates whether the dialog should draw right-to-left for RTL languages.
ShowInTaskbar	Determines whether the dialog appears in the Windows Taskbar.
Size	The size of the dialog in pixels.
SizeGripStyle	Determines when the SizeGrip will be displayed for the dialog.
StartPosition	Determines the position of the dialog when it first appears.
TabStop	Indicates whether the user can use the **Tab** key to give focus to the dialog.
Text	The text contained in the dialog.
TopMost	Determines whether the dialog is above all other non-topmost forms, even when deactivated.
TransparencyKey	A color that will appear transparent when painted on the dialog.
Visible	Determines whether the dialog is visible or hidden.
WindowState	Determines the initial visual state of the dialog.

You can use these properties to configure the appearance of the Print Preview dialog box. For instance, you can use the *WindowState* property to show the dialog box as maximized in code that would appear like the following snippet:

```
With PrintPreviewDialog1
     .WindowState = FormWindowState.Maximized
     .ShowDialog()
End With
```

Another related dialog box is the Page Setup dialog box, which we discuss in the next section.

The *PageSetupDialog* Control

The Windows Forms *PageSetupDialog* control displays a dialog box that allows the user to set page details for printing in Windows applications. The Page Setup dialog box allows the user to set border and margin adjustments, headers and footers, and page orientation (portrait or landscape).

You can also use the properties of the *PageSetupDialog* control to configure the behavior of the Page Setup dialog box. Table 7.15 describes properties of the *PageSetupDialog* control.

Table 7.15 Properties of the *PageSetupDialog* Control

Property	Description
(Name)	Indicates the name used in code to identify the dialog.
AllowMargins	Enables and disables editing of margins.
AllowOrientation	Enables and disables the Orientation radio buttons.
AllowPaper	Enables and disables editing of paper size.
AllowPrinter	Enables and disables the Printer button.
Document	The PrintDocument from which to get printer settings.
MinMargins	The smallest margin the user is allowed to select.
Modifiers	Indicates the visibility level of the dialog.
ShowHelp	Controls whether the Help button is displayed.
ShowNetwork	Controls whether the Network button is displayed.

For example, you can use the *AllowMargins* property to enable editing of margins and the *ShowNetwork* property to display the Network button. The code would appear as in the following snippet:

```
With PageSetupDialog1
    .AllowMargins = True
    .ShowNetwork = True
    .ShowDialog()
End With
```

We have now discussed the preconfigured dialog boxes provided by the Windows Forms framework. These dialog boxes provide a lot of functionality, but at times they may not suit your needs. You can create your own dialog boxes to provide exactly the functionality that you require. You will learn how to do so in the following section.

Creating Dialog Boxes

If the preformatted dialog boxes included in the .NET Framework do not suit your needs, you can create your own. Creating a dialog box is another step-by-step procedure, which is outlined here:

1. Create a form.
2. Set the **BorderStyle** property of the form to **FixedDialog**.
3. Set the **ControlBox**, **MinimizeBox**, and **MaximizeBox** properties of the form to **False**.
4. Customize the appearance of the form appropriately.

Customize event handlers in the Code window appropriately.

NOTE

Dialog boxes do not usually include sizeable borders, menu bars, Minimize and Maximize buttons, window scroll bars, or status bars.

Dialog boxes are displayed modally to prevent the user from performing tasks outside of the dialog box. You can display a dialog box modally by using the

ShowDialog method, as we have seen earlier in this chapter. Like the *Show* method of the *MessageBox* class, the *ShowDialog* method also returns input from the user in the form of a dialog result. For example, the following snippet determines the input from the user and handles it accordingly:

```
Dim frmNewEmployee As frmDialogBox = New frmDialogBox()


If frmNewEmployee.DialogResult = DialogResult.OK Then
    'Handle form data


End If
```

Creating and Working with Menus

Menus hold commands grouped by a common topic. Menus make it easy for your users to navigate your application as they see menus they have already used in Windows, such as the File menu. As an added benefit, using a menu to hold commands avoids using precious real estate on your forms. Let's look at creating menus.

Adding Menus to a Form

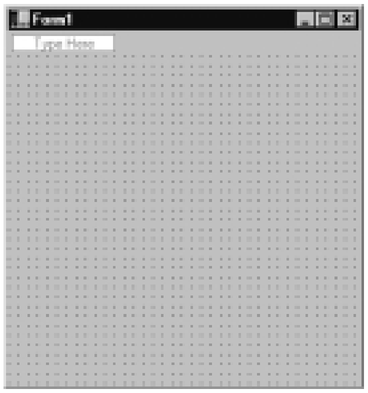
You can add menus to a form using the *MainMenu* control. Menus are often added to a form at design time. You can use the new Menu Designer in Visual Basic .NET to add menus to your forms at design time.

Exercise 7.2 Adding a Menu to a Form at Design Time

In this exercise, you will add a File menu with an Exit menu item to a form at design time:

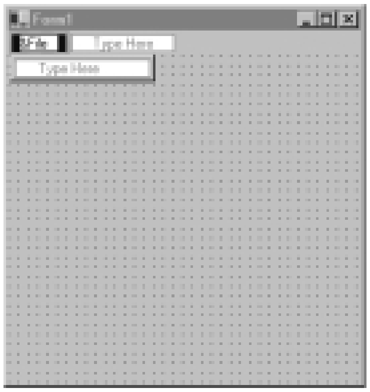
1. From the **File** menu, select **New Project**.
2. In the **Visual Basic Projects** list, select the **Windows Application** template, and then click **OK**.
3. In the **Toolbox**, select the **Win Forms** tab, and then double-click the **MainMenu** control. A menu is added to the form displaying the text *Type Here* (see Figure 7.11), and the **MainMenu** control is added to the component tray.

Figure 7.11 The Menu Designer



4. In the **Menu Designer**, click the text **Type Here** to select the menu, and then type **&File** (see Figure 7.12).

Figure 7.12 The File Menu



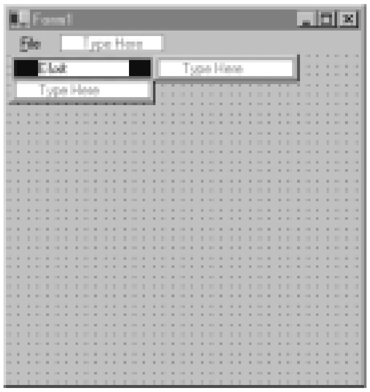
5. Click the area below the **File** menu item to add another entry to the same menu, and type **E&xit**. (see Figure 7.13).

The Windows Forms framework includes four menu enhancements that you can use to convey information to the user. Table 7.16 describes these menu enhancements.

Table 7.16 Menu Enhancements

Enhancement	Description
Check marks	Indicate whether a feature is turned on or off (such as whether a ruler is displayed along the margin of a word-processing application) or to indicate which of a list of files is being displayed (such as in a Window menu).
Shortcut keys	Allow access to menu items using keyboard commands.
Access keys	Allow keyboard navigation of menus (pressing the **Alt** key, and the underlined access key chooses the desired menu or menu item).
Separator bars	Used to group related commands within a menu and make menus easier to read.

Figure 7.13 The File Menu and the Exit Menu Item



Check marks allow the user to conveniently toggle a feature on or off in your application. In an MDI application, they are also useful when you want to use a Window menu to indicate which MDI child form has the focus. What if you wanted to add a check mark to a menu at design time? Complete the following steps:

1. In the **Menu Designer**, select the menu item.
2. Click the area to the left of the menu item.

   A check mark appears. You can remove the check mark by repeating the same steps.

Shortcut keys allow the user to use keyboard command to access menu items. For example, in many applications you can save your work by pressing **Ctrl+S**. To add a shortcut key to a menu item at design time, perform the following steps:

1. Use the **View** menu to open the **Properties window**.
2. In the **Menu Designer**, select the menu item.
3. In the **Properties window**, set the **Shortcut** property to one of the values offered in the drop-down list.

Access keys allow the user to navigate menus using the keyboard—pressing the ALT key and the underlined access key. When the menu opens and shows items with access keys, the user just needs to press the access key to select the menu item. Use the following steps to add an access key to a menu item at design time:

1. In the **Menu Designer**, select the menu item.
2. When setting the **Text** property, enter an ampersand (**&**) prior to the letter you want to be underlined as the access key. For example, typing **&File** as the **Text** property of a menu item will result in a menu item that appears as *File*.

Separator bars are used to group related commands within a menu and make menus easier to read. To add a separator bar as a menu item at design time, you should do the following: In the **Menu Designer**, right-click the location where you want a separator bar, and choose **New Separator**.

You can also add a menu to a form and add menu enhancements to menu items at runtime. The next section covers working with menus at runtime.

Dynamically Creating Menus

We have seen how to add a menu to a form at design time. You can also add a menu to a form at runtime. We walk through this process in Exercise 7.3.

Exercise 7.3 Adding a Menu to a Form at Design Time

In this exercise, you will add a File menu with an Exit menu item to a form at runtime:

1. From the **File** menu, select **New Project**.

2. In the **Visual Basic Projects** list, select the **Windows Application** template, and then click **OK**.
3. In the **Code** window, type the following code to add a **MainMenu** control to **Form1** within a public method **AddMenu**:

```
Public Sub AddMenu()
    Dim mmMainMenu As New MainMenu()
    Menu = mmMainMenu
End Sub
```

4. Within the **AddMenu** method, use the following code to create **MenuItem** objects to add to the **MainMenu** object's collection:

```
    Dim mnuFile As New MenuItem()
    Dim mnuFileExit As New MenuItem()
```

5. Within the **AddMenu** method, set the **Text** property for each of these menu items:

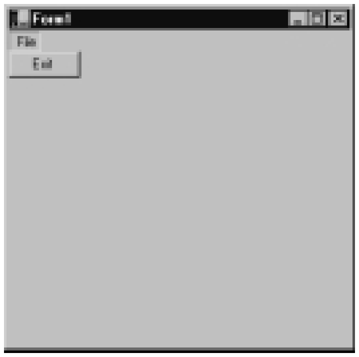
```
    mnuFile.Text = "&File"
    mnuFileExit.Text = "E&xit"
```

6. Within the **AddMenu** method, create the top-level **File** menu item and add the **Exit** menu item:

```
    mmMainMenu.MenuItems.Add(mnuFile)
    mnuFile.MenuItems.Add(mnuFileExit)
```

When you call the *AddMenu* method, Form1 is displayed as in Figure 7.14.

Figure 7.14 The File Menu and the Exit Menu Item at Runtime



Adding Status Bars to Forms

A status bar is a horizontal control that is usually positioned at the bottom of a form. Status bars are used to display textual information such as date and time or descriptions of menu items. Status bars also displays modes of the keyboard, such as when the user presses the **Insert**, **Num Lock**, or **Scroll Lock** keys. Table 7.17 shows the properties of the *StatusBar* control.

Table 7.17 Properties of the *StatusBar* Control

Property	Description
(Bindings)	This collection holds all the bindings of properties of the status bar to data sources.
(Name)	Indicates the name used in code to identify the status bar.
AccessibleDescription	The description that will be reported to accessibility clients.
AccessibleName	The name that will be reported to accessibility clients.
AccessibleRole	The role that will be reported to accessibility clients.
AllowDrop	Determines if the status bar will receive drag-and-drop notifications.
Anchor	The anchor of the status bar.
CausesValidation	Indicates whether the status bar causes and raises validation events.
ContextMenu	The shortcut menu to display when the user right-clicks the status bar.
Cursor	The cursor that appears when the mouse passes over the status bar.
Dock	The docking location of the status bar, indicating which borders are docked to the container.
Enabled	Indicates whether the status bar is enabled.
Font	The font used to display text in the status bar.
IMEMode	Determines the IME status of the status bar when selected.
Location	The position of the top-left corner of the status bar with respect to its container.
Locked	Determines if the status bar can be moved or resized.
Modifiers	Indicates the visibility level of the status bar.

Continued

Table 7.17 Continued

Property	Description
Panels	The panels in the status bar.
RightToLeft	Indicates whether the status bar should draw right-to-left for RTL languages.
ShowPanels	Determines if the status bar displays panels, or if it displays a single line of text.
Size	The size of the status bar in pixels.
SizingGrip	Determines whether the status bar has a sizing grip.
TabIndex	Determines the index in the Tab order that the status bar will occupy.
TabStop	Indicates whether the user can use the **Tab** key to give focus to the status bar.
Text	The text contained in the status bar.
Visible	Determines whether the status bar is visible or hidden.

You can add panels to a status bar at design time or at runtime. We discuss both methods. First, let's take a look at adding panels to a status bar at design time. For example, to add a panel with the text *Spell Check* to a status bar:

1. In the **Toolbox**, select the **Win Forms** tab, and then double-click the **StatusBar** control.
2. In the **Properties Window**, set the **ShowPanels** property to **True**.
3. In the **Properties Window**, select the **Panels** property, and then choose the ellipsis box.
4. In the **StatusBarPanel Collection Editor**, select the **Add** button.
5. In the **Properties** box, set the **Text** property to **Spell Check** and then set the **AutoSize** property to **Contents**.

You can also add a panel to a status bar or change a panel's text at runtime. This is useful when you want to describe the functionality of menu items as the mouse moves over them. Typically, you would create a panel at design time and change only its text at runtime, but let's see how you would add a panel to a status bar dynamically:

```
Dim sbpStatusBarPanel As New StatusBarPanel()
```

```
With StatusBar1
     .ShowPanels = True
     .Panels.Add(sbpStatusBarPanel)
End With
```

Now you can change the text of the panel dynamically. For example, as the user moves the mouse over the New menu item in the File menu, you can describe the functionality as follows:

```
StatusBar1.Panels(0).Text = "Creates a new file."
```

Adding Toolbars to Forms

A toolbar is another control that is often docked to an edge of a form. In the .NET Framework, toolbars display buttons that can appear as standard buttons, toggle-style buttons, or drop-down style buttons. Toolbar buttons can appear as raised or flat—when the mouse pointer moves over a flat button, its appearance changes to three-dimensional. Toolbars can also display drop-down menus that activate commands. As is common throughout Windows, a button can display an image along with text. Table 7.18 describes the properties of the Toolbar control:

Table 7.18 Properties of the Toolbar Control

Property	Description
(Bindings)	This collection holds all the bindings of properties of the toolbar to data sources.
(Name)	Indicates the name used in code to identify the toolbar.
AccessibleDescription	The description that will be reported to accessibility clients.
AccessibleName	The name that will be reported to accessibility clients.
AccessibleRole	The role that will be reported to accessibility clients.
AllowDrop	Determines if the toolbar will receive drag-and-drop notifications.
Anchor	The anchor of the toolbar.
Appearance	Controls the appearance of the toolbar.
AutoSize	Controls whether the toolbar will automatically size itself based on button size.

Continued

Table 7.18 Continued

Property	Description
BackgroundImage	The background image used for the toolbar.
BorderStyle	Controls what type of border the toolbar will have.
Buttons	The collection of toolbar buttons that make up the toolbar.
ButtonSize	Suggests the size of buttons in the toolbar. Button sizes might still be different based on text, drop-down arrows, and others.
CausesValidation	Indicates whether the toolbar causes and raises validation events.
ContextMenu	The shortcut menu to display when the user right-clicks the toolbar.
Cursor	The cursor that appears when the mouse passes over the toolbar.
Divider	Controls whether the toolbar will display a 3D line at the top of its client area.
Dock	The docking location of the toolbar, indicating which borders are docked to the container.
DropDownArrows	Controls whether the toolbar will display an arrow on the side of drop-down buttons.
Enabled	Indicates whether the control is enabled.
Font	The font used to display text in the control.
ImageList	The *ImageList* control from which the toolbar will get button images.
IMEMode	Determines the IME status of the toolbar when selected.
Location	The position of the top-left corner of the toolbar with respect to its container.
Locked	Determines if the toolbar can be moved or resized.
Modifiers	Indicates the visibility level of the toolbar.
ShowToolTips	Indicates whether tool tips will be shown for each button, if available.
Size	The size of the toolbar in pixels.
TabIndex	Determines the index in the Tab order that the toolbar will occupy.

Continued

Table 7.18 Continued

Property	Description
TabStop	Indicates whether the user can use the **Tab** key to give focus to the toolbar.
TextAlign	Controls how the text is positioned relative to the image in each button.
Visible	Determines whether the toolbar is visible or hidden.
Wrappable	Indicates if more than one row of buttons is allowed.

This amount of functionality may seem daunting, but you do not need to be familiar with all these properties to create a toolbar. As a minimum, you should be aware of the *Buttons* property, which is the collection of buttons that that make up a toolbar. You can use the *Buttons* property to add buttons to a toolbar at design time or at runtime. Let's discuss the most challenging method: how to add a button to a toolbar at runtime.

As we have seen with other collections, you can use the *Add* method of the Buttons collection to add a button to a toolbar. For example, to add a Save button to a toolbar, use the following code:

```
Dim tbbSave As New ToolBarButton()


tlbToolbar.Buttons.Add(tbbSave)
```

We have now seen how to create forms and add controls to forms. In the next section, we discuss binding data sources to forms.

Data Binding

We have now discussed many ways of displaying information to the user and collecting information from the user. In most applications, the information displayed comes from a data source and the information collected goes to a data source. The Windows Forms framework allows you to bind data sources to forms, which is a very convenient way to open and save datasets. There are two types of data binding, and we discuss them in the following sections.

Simple Data Binding

In *simple data binding,* a single value within a data set is bound to a property of a component. For example, the *Text* property of a text box can be bound to the

FirstName column of an Employees table. The following snippet shows the code for this scenario:

```
Dim dtEmployee As DataTable
Dim txtFirstName As New Textbox

dtEmployee = dsDataSet.Tables("Employee")
txtFirstName.Bindings.Add("Text", dtEmployee, "FirstName")
```

Because the binding is simple, only one first name will be shown at a time. This makes a text box a good choice for simple data binding—text boxes contain only one piece of information at a time: the value of the *Text* property. Other controls such as combo boxes and list boxes expose an Items collection that can contain more than one piece of information at a time. These controls are good candidates for complex data binding, which we discuss in the next section.

Complex Data Binding

In *complex data binding,* a whole dataset is bound to a component. For example, a combo box can be data bound to the same dataset and display all first names in its drop down box. Let's look at the code:

```
Dim dtEmployee As DataTable
Dim cboFirstName As New ComboBox

dtEmployee = dsDataSet.Tables("Employee")
cboFirstName.DataSource = dtEmployee
cboFirstName.DisplayMember = "FirstName"
```

In contrast to the text box in the previous section, the combo box we used in this example was bound using complex data binding. As you know, the drop-down of the combo box can contain more than one item. In this example, all employee first names will appear in the drop-down. Controls with an Items collection make good candidates for complex data binding—grids are an especially popular choice. When you bind data to a component, you can choose from a number of data sources, which we discuss in the following section.

Data Sources for Data Binding

When you bind data to a component you can choose from several data sources. In Visual Basic .NET, a data source is any grouping of data that implements the *IList* interface. The *IList* interface represents a collection of objects that can be individually indexed. This means that you can use regular collections and even arrays as data sources for data binding. Although not a comprehensive list, Table 7.19 describes possible data sources that are commonly used for data binding.

Table 7.19 Data Sources for Data Binding

Data Source	Description
DataTable	The representation of a table. DataTable contains two collections: DataColumn, representing the columns of data in a given table (which ultimately determine the kinds of data that can be entered into that table), and DataRow, representing the rows of data in a given table. This is the actual data within the table.
DataView	A customized view of a single DataTable that may be filtered, searched, or sorted. A DataView is the data snapshot used by complex bound controls.
DataSet	The in-memory cache that consists of tables, relations, and constraints. Each table has a collection of columns. These columns represent the arrangement of the DataSet. Each table can then have multiple rows, representing the data within the DataSet, which are aware of their original state along with their current state. In this manner, the DataSet can track changes that have occurred.
DataSetView	A customized view of the entire DataSet, analogous to a DataView, but with relations included. A TableSettings collection allows you to set default filters and sort options for any views that the DataSetView has for a given table.
Array	An ordered collection of data contained in a variable and referenced by a single variable name. Each element of the array can be referenced by a numerical subscript.
Collection	An object that contains zero or more objects. Collections normally contain objects of the same class.

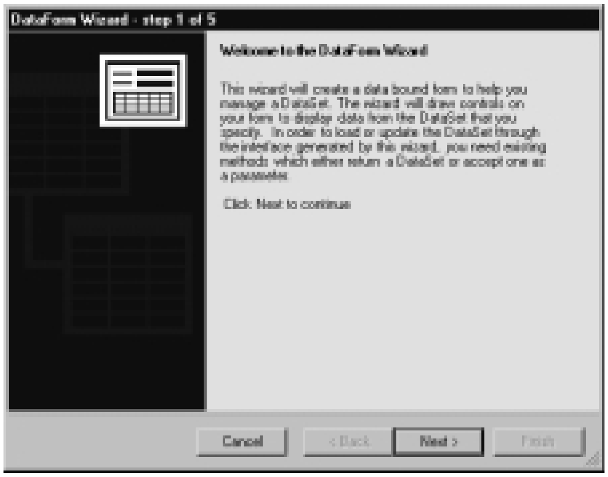
In the next section, we discuss the Data Form Wizard, an easy way to quickly generate a data-bound form.

Using the Data Form Wizard

The Data Form Wizard allows you to quickly generate a data-bound form. The wizard allows you to specify a dataset, tables and fields, and other display options. The wizard then produces and binds controls on the form to display data. After you produce the controls, you can change their properties to suit your needs. Let's walk through producing a data-bound form:

1. From the **Data** tab of the **Toolbox**, drag a **DataFormWizard** onto the form. The wizard is launched (see Figure 7.15).

Figure 7.15 The Data Form Wizard



DataForm Wizard - step 1 of 5

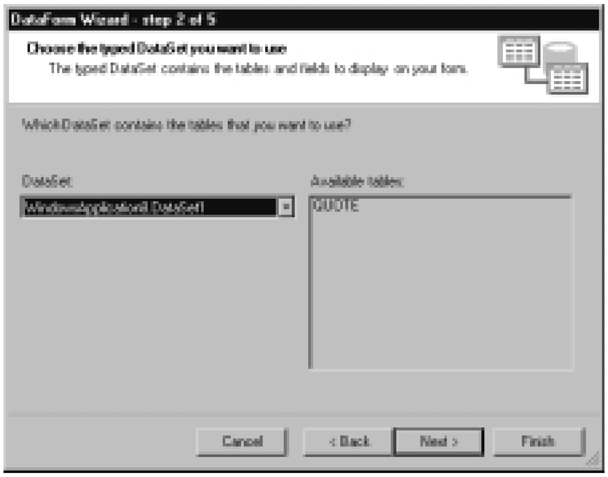
Welcome to the DataForm Wizard

This wizard will create a data bound form to help you manage a DataSet. The wizard will draw controls on your form to display data from the DataSet that you specify. In order to load or update the DataSet through the interface generated by this wizard, you need existing methods which either return a DataSet or accept one as a parameter.

Click Next to continue

2. In the wizard's second pane (shown in Figure 7.16), select a previously created dataset.

Figure 7.16 The Data Form Wizard—The Second Pane



DataForm Wizard - step 2 of 5

Choose the typed DataSet you want to use

The typed DataSet contains the tables and fields to display on your form.

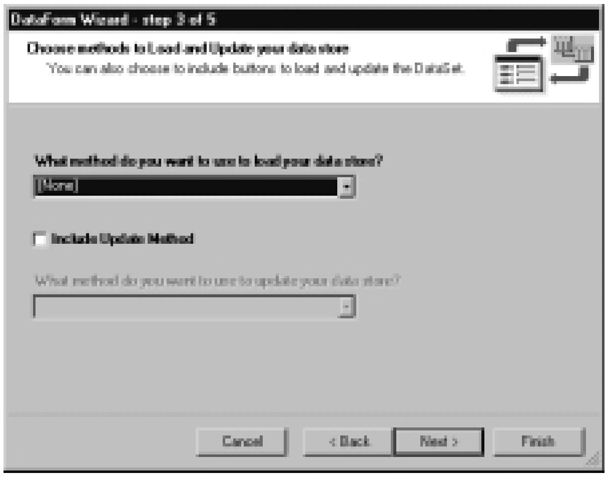
Which DataSet contains the tables that you want to use?

DataSet: WindowsApplication8.DataSet1

Available tables: QUOTE

3. In the third pane, do not check the **Include Update Method** box (see Figure 7.17).

Figure 7.17 The Data Form Wizard—The Third Pane



DataForm Wizard - step 3 of 5

Choose methods to Load and Update your data store

You can also choose to include buttons to load and update the DataSet.

What method do you want to use to load your data store?

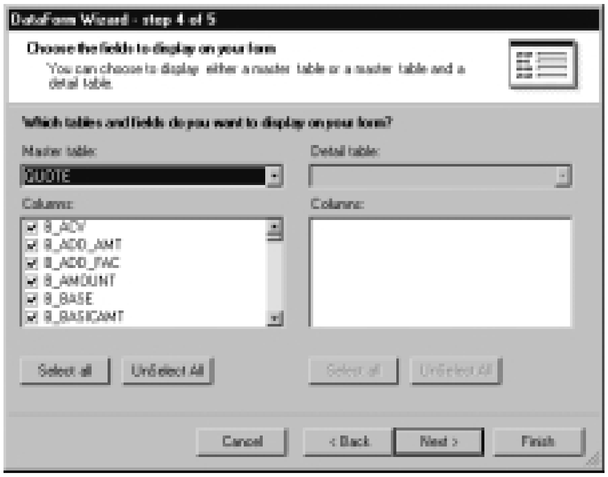
[None]

☐ Include Update Method

What method do you want to use to update your data store?

4. In the fourth pane, select the appropriate table and the columns (see Figure 7.18).

Figure 7.18 The Data Form Wizard—The Fourth Pane



DataForm Wizard - step 4 of 5

Choose the fields to display on your form

You can choose to display either a master table or a master table and a detail table.

Which tables and fields do you want to display on your form?

Master table: QUOTE

Detail table:

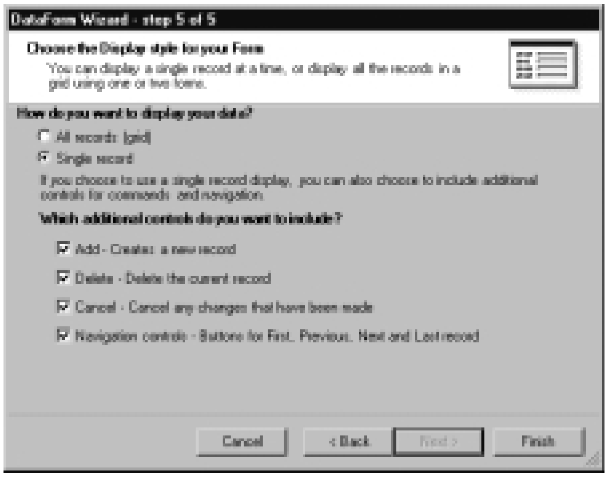
Columns:

☑ B_ACV
☑ B_ADD_AMT
☑ B_ADD_FAC
☑ B_AMOUNT
☑ B_BASE
☑ B_BASICAMT

Columns:

5. In the fifth pane, under **How do you want to display your data?**, select **Single records** (see Figure 7.19).

Figure 7.19 The Data Form Wizard—The Fifth Pane



DataForm Wizard - step 5 of 5

Choose the Display style for your Form

You can display a single record at a time, or display all the records in a grid using one or two forms.

How do you want to display your data?

- All records (grid)
- [✓ Single record]

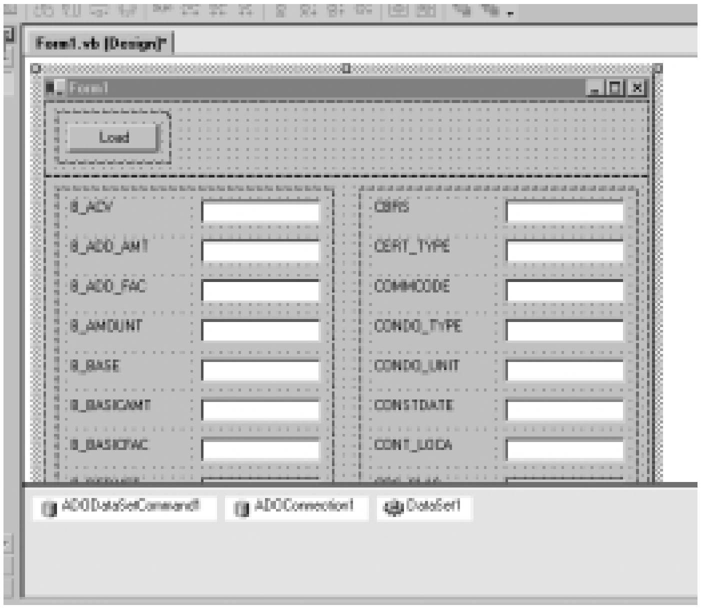
If you choose to use a single record display, you can also choose to include additional controls for commands and navigation.

Which additional controls do you want to include?

- ☑ Add - Creates a new record
- ☑ Delete - Delete the current record
- ☑ Cancel - Cancel any changes that have been made
- ☑ Navigation controls - Buttons for First, Previous, Next and Last record

6. Click **Finish**. On the form, the wizard creates a Load button and one control for each column. When the Data Form Wizard has finished, you have a form that is ready to run and display data (see Figure 7.20).

Figure 7.20 The Form Created by the Data Form Wizard



7. Press **F5**.
8. When the form is displayed, click the **Load** button.

The dataset is populated with records from the database, and the data-bound controls display what is in the dataset. You can easily change the appearance and the behavior of the controls created by the Data Form Wizard by changing their properties. In the next chapter, we discuss changing control properties to suit your needs. The Windows Forms Class Viewer is another useful tool included in the .NET Framework. We discuss the class viewer in the next section.

Using the Windows Forms Class Viewer

The Windows Forms class viewer allows you to quickly look up information about a class or series of classes based on a search pattern. The class viewer displays information by reflecting on the type using the Common Language Runtime reflection API.

To use the Windows Forms class viewer, start **wincv.exe** from the command line and type part or all of a type name into the text box at the top of the form. The list box on the left hand side of the form will then display a list of all the types that **wincv** finds based on the name you entered. When you select a type from the list, the type definition is displayed in the area on the right.

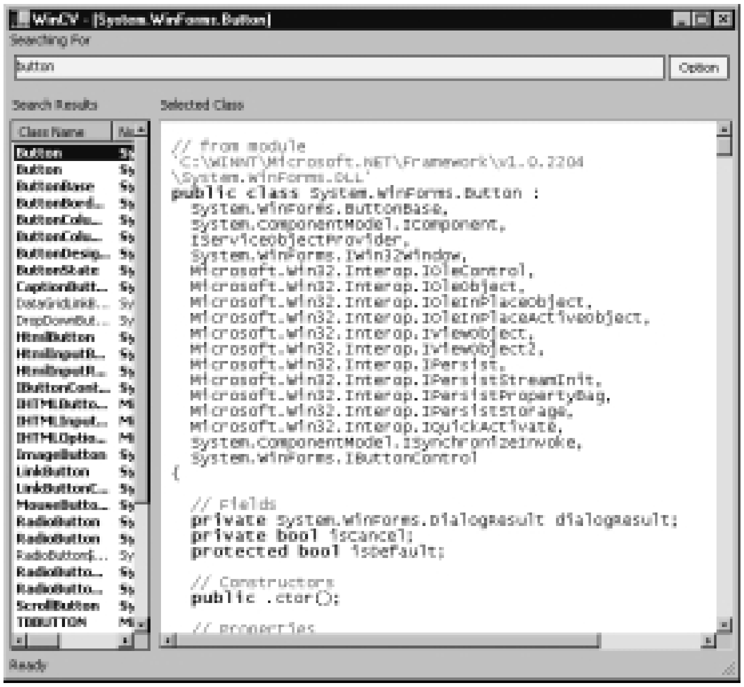
You can use the Options button to copy the contents of the display to the clipboard. You can then conveniently paste the clipboard contents in a Windows application. For example, the Figure 7.21 shows the definition for the type *System.WinForms.Button*.

We have now discussed the Data Form Wizard and the Windows Forms class viewer, two handy .NET Framework tools. Another useful .NET Framework tool is the Windows Forms ActiveX Control Importer, which we discuss in the next section.

Using the Windows Forms ActiveX Control Importer

As was mentioned at the beginning of this chapter, Windows Forms can host only Windows Forms Controls—classes that are derived from *System.WinForms .Control*. In order to host an ActiveX Control on a form, it must appear to be a Windows Forms Control. Also, an ActiveX Control requires hosting in an ActiveX Control Container.

Figure 7.21 The Definition for the Type *System.WinForms.Button* in the Windows Forms Class Viewer



Class Name	Na
Button	Sy
Button	Sy
ButtonBase	Sy
ButtonBord...	Sy
ButtonColu...	Sy
ButtonColu...	Sy
ButtonDesig...	Sy
ButtonState	Sy
CaptionButt...	Sy
DataGridLinkB...	Sy
DropDownBut...	Sy
HtmlButton	Sy
HtmlInputB...	Sy
HtmlInputR...	Sy
IButtonCont...	Sy
IHTMLButto...	Mi
IHTMLInput...	Mi
IHTMLOptio...	Mi
ImageButton	Sy
LinkButton	Sy
LinkButtonC...	Sy
MouseButto...	Sy
RadioButton	Sy
RadioButton	Sy
RadioButton$...	Sy
RadioButto...	Sy
RadioButto...	Sy
ScrollButton	Sy
TBBUTTON	Mi

```
// from module
'C:\WINNT\Microsoft.NET\Framework\v1.0.2204
\System.WinForms.DLL'
public class System.WinForms.Button :
  System.WinForms.ButtonBase,
  System.ComponentModel.IComponent,
  IServiceObjectProvider,
  System.WinForms.IWin32Window,
  Microsoft.Win32.Interop.IOleControl,
  Microsoft.Win32.Interop.IOleObject,
  Microsoft.Win32.Interop.IOleInPlaceObject,
  Microsoft.Win32.Interop.IOleInPlaceActiveObject,
  Microsoft.Win32.Interop.IViewObject,
  Microsoft.Win32.Interop.IViewObject2,
  Microsoft.Win32.Interop.IPersist,
  Microsoft.Win32.Interop.IPersistStreamInit,
  Microsoft.Win32.Interop.IPersistPropertyBag,
  Microsoft.Win32.Interop.IPersistStorage,
  Microsoft.Win32.Interop.IQuickActivate,
  System.ComponentModel.ISynchronizeInvoke,
  System.WinForms.IButtonControl
{

  // Fields
  private System.WinForms.DialogResult dialogResult;
  private bool isCancel;
  protected bool isDefault;

  // Constructors
  public .ctor();

  // Properties
```

All is not lost if you created ActiveX controls in previous versions of Visual Basic. You can use the ActiveX Control Importer to convert the type definitions found within the COM type library for an ActiveX Control into a Windows Forms Control.

In order to achieve hosting, there is a base class *System.WinForms.AxHost* that derives from *System.WinForms.RichControl*. This control appears as a Windows Forms control to your Windows Form and an ActiveX Control Container to your ActiveX control. To host the ActiveX Control, use the Windows Forms ActiveX Control Importer to create a wrapper control that derives from *System.WinForms .AxHost*. This generated control hosts the ActiveX Control and exposes its properties, methods, and events (PMEs) as PMEs of the generated control.

Summary

The Windows Forms framework exposes forms, their properties, methods, and events. MDI applications make use of unique forms: MDI parent and child forms. A dialog boxes is also a specific type of form with a particular border style and title bar. You can add controls to your forms to display information to the user and collect information from the user. You can also add menus, status bars, and toolbars to your forms. You can open a dataset to display information from the user. After you collect information from the user you can save the dataset. Windows Forms data binding makes this all quick and easy.

Solutions Fast Track

Application Model

☑ Windows Forms is the new platform for Microsoft Windows–based application development, based on the .NET Framework.

☑ Windows Forms provides a clear, object-oriented, extensible set of classes that enable you to develop rich Windows-based applications. Additionally, in a multi-tier distributed solution, Windows Forms can act as the local user interface.

☑ The properties of a form determine its appearance and behavior. You can use the Properties window to change properties of a form at design-time. You can change many properties of a form at runtime as well.

Manipulating Windows Forms

☑ The form is the primary vehicle for user interaction within a Windows-based application. You can combine controls and code to collect information from the user and respond to it, work with data stores, and query and write to the Registry and file system on the user's computer.

☑ When you add a Windows form to your project, many of the form's properties are set to commonly used values by default. Although these values are convenient, they will not always suit your needs.

☑ A modal form must be closed before you can continue working with the rest of the application.

- ☑ Contrary to a modal form, a modeless form allows the user to shift the focus between the form and another form without closing the initial form.
- ☑ A top-most form stays in front of non-topmost forms even when inactive.

Form Events

- ☑ Events occur for forms when the user open or closes a form, moves between forms, or interacts with the surface of a form.
- ☑ Events that occur when the user interacts with a form can be triggered by using the mouse or keyboard.

Creating Multiple Document Interface Applications

- ☑ The MDI parent form is at the heart of an MDI application. It is the container for the multiple documents, the child forms, within an MDI application.
- ☑ MDI child forms are forms that operate within an MDI parent form in an MDI application. In an MDI application, these are often the forms with which the user interacts the most.
- ☑ In an MDI application, the active child form is the child form that has the focus or was most recently active.

Adding Controls to Forms

- ☑ Most forms contain controls that display information to the user or collect information from the user.
- ☑ When a control is anchored to an edge, the distance between the control's closest edge and the specified edge will remain constant.
- ☑ The *Dock* property determines to which form edges a control is docked.
- ☑ Z-ordering is the visual layering of controls on a form along its depth, or z-axis. The control at the top of the z-order overlaps all other controls. All other controls overlap the control at the bottom of the z-order.

Dialog Boxes

- ☑ Dialog boxes display information to the user and collect information from the user. They are useful because they present visual cues that are familiar to the Windows user.
- ☑ Technically, a dialog box is merely a form with a border style of fixed dialog.
- ☑ A message box displays application-related information to the user and collects an acknowledgement or a choice from the user.
- ☑ At times you can use preconfigured dialog boxes that are included in the Windows Forms framework in lieu of creating your own. When you use standard Windows dialog boxes, the user can easily recognize the functionality of the dialog box.
- ☑ If the preformatted dialog boxes included in the .NET Framework do not suit your needs, you can create your own.

Creating and Working with Menus

- ☑ Menus hold commands grouped by a common topic.
- ☑ Menus make it easy for your users to navigate your application because they see menus they have already used in Windows, such as the File menu.
- ☑ As an added benefit, using a menu to hold commands avoids using precious real estate on your forms.
- ☑ You can add menus to a form using the *MainMenu* control.

Adding Status Bars to Forms

- ☑ A status bar is a horizontal control that is usually positioned at the bottom of a form.
- ☑ Status bars are used to display textual information such as date and time, or descriptions of menu items.
- ☑ Status bars also displays modes of the keyboard, such as when the user presses the **Insert**, **Num Lock**, or **Scroll Lock** keys.

Adding Toolbars to Forms

- ☑ A toolbar is another control that is often docked to an edge of a form.
- ☑ In the .NET Framework, toolbars display buttons that can appear as standard buttons, toggle-style buttons, or drop-down style buttons.
- ☑ Toolbars can also display drop-down menus that activate commands.

Data Binding

- ☑ The Windows Forms framework allows you to bind data sources to forms, which is a very convenient way to open and save datasets.
- ☑ In simple data binding, a single value within a data set is bound to a property of a component.
- ☑ In complex data binding, a whole dataset is bound to a component.

Using the Windows Forms Class Viewer

- ☑ The Windows Forms class viewer allows you to quickly look up information about a class or series of classes based on a search pattern.
- ☑ The class viewer displays information by reflecting on the type using the Common Language Runtime reflection API.
- ☑ To use the Windows Forms class viewer, start **wincv.exe** from the command line and type part or all of a type name into the text box at the top of the form.

Using the Windows Forms ActiveX Control Importer

- ☑ In order to host an ActiveX Control on a form, it must appear to be a Windows Forms Control. Also, an ActiveX Control requires hosting in an ActiveX Control Container.
- ☑ In order to achieve hosting there is a base class *System.WinForms.AxHost* that derives from *System.WinForms.RichControl.* This control appears as a Windows Forms control to your Windows Form and an ActiveX Control Container to your ActiveX control.

☑ To host the ActiveX Control, use the Windows Forms ActiveX Control Importer to create a wrapper control that derives from *System.WinForms .AxHost*. This generated control hosts the ActiveX Control and exposes its properties, methods, and events (PMEs) as PMEs of the generated control.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: What is a form?

A: A form is a representation of a window. Most forms are used to display controls that display information to the user or collect input from the user.

Q: What are modal and modeless forms?

A: A modal form must be closed before you can continue working with the rest of the application. A modeless form allows the user to shift the focus between the form and another form without closing the initial form.

Q: What is an MDI application?

A: Multiple Document Interface applications allow simultaneous display of multiple documents, with each document displayed in its own window. MDI applications consist of an MDI parent form and MDI child forms.

Q: What is a dialog box?

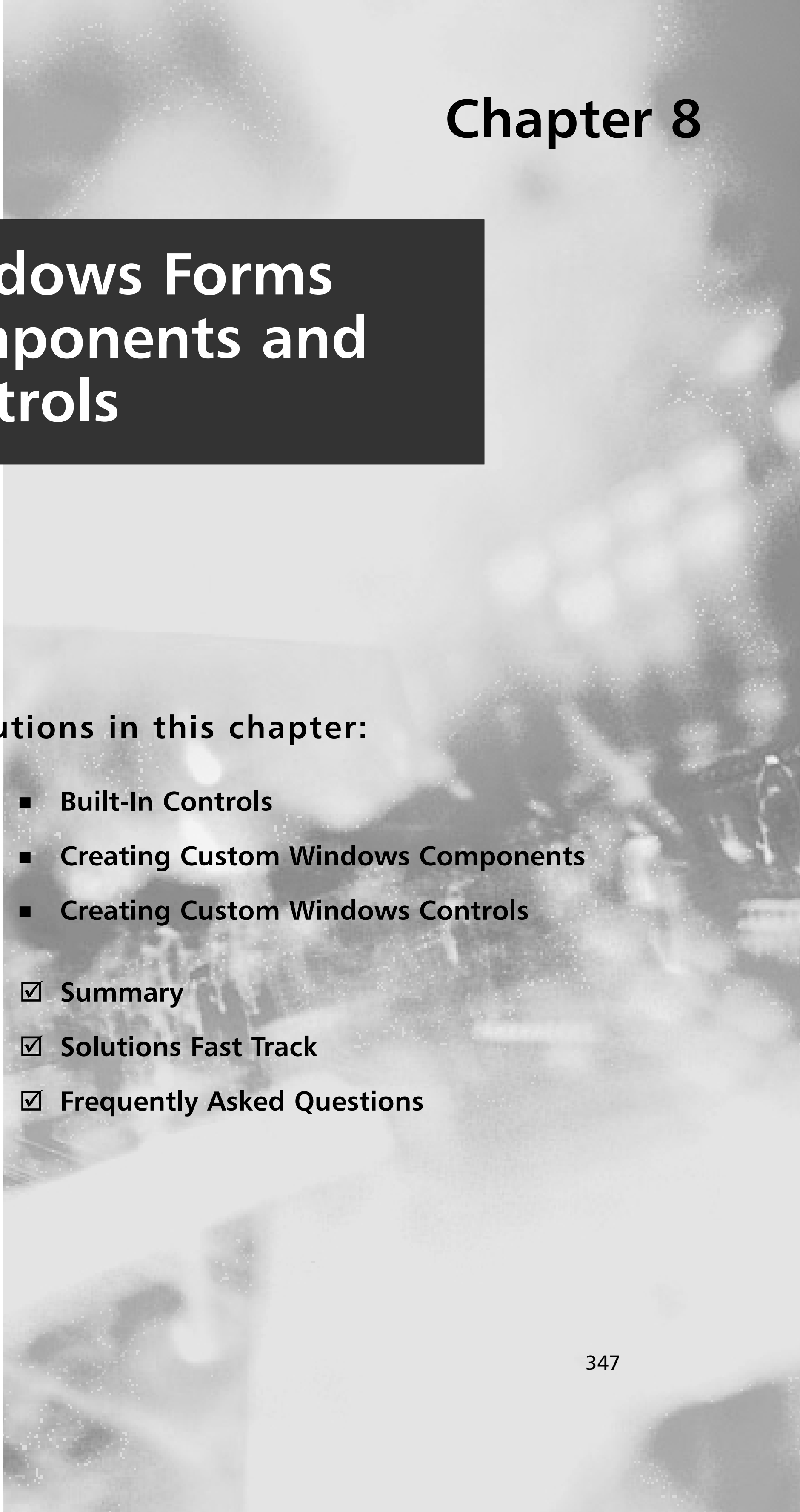
A: Technically, a dialog box is merely a form with a border style of Fixed Dialog and other visual cues.

Q: What is z-ordering?

A: Z-ordering is the visual layering of controls on a form along its depth, or z-axis. You can layer controls visually using their z-order. The control at the top of the z-order overlaps all other controls. All other controls overlap the control at the bottom of the z-order.

Q: What is the difference between simple and complex data binding?

A: In simple data binding, a single value within a data set is bound to a property of a component. In complex data binding, a whole dataset is bound to a component.



Chapter 8

Windows Forms Components and Controls

Solutions in this chapter:

- Built-In Controls
- Creating Custom Windows Components
- Creating Custom Windows Controls

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

Now that we have learned how to work with Windows Forms, we will take a look at Windows Controls. Visual Basic .NET comes with an extensive amount of built-in controls for use in your applications. This allows applications to be developed more rapidly. There are controls for displaying labels, allowing user input of text, and working with numbers. There are controls for users to make choices, and various ways of displaying information to the user. As in previous versions of Visual Basic, you can also use controls created by third-party vendors. This allows you great flexibility in utilizing the controls that best fit your needs. Controls are manipulated by their properties and methods. A control can be configured at design time with its design-time properties and further controlled at runtime with its runtime properties and methods.

So what do you do if you can't find a control that does exactly what you need? One option is to compromise and accept the control(s) closest to your needs. This is not always acceptable, however. In Visual Basic .NET, you will still have the ability to develop your own custom-made controls, which allows you to create the exact desired functionality. You can even develop your own control to become one of those third-part vendors that sell controls. You can create design-time properties as well as runtime properties and methods to provide the developer with access to manipulate your control. You even have the ability to bind your control to a data source.

You don't always need to have controls with a GUI. Sometimes you just want to create a component that only provides programmatic functionality. This allows you to create libraries or objects that greatly facilitate reuse of code. You also might need a middle-tier component to provide business logic for a multi-tier application. This is what Windows components are for. These are similar to Visual Basic 6.0 ActiveX DLLs. Whatever the needs of your application, you can use existing controls as necessary, or develop your own custom controls and components.

Built-In Controls

Visual Basic .NET offers numerous controls you can use to build an application. Each of these controls has a particular function. In some cases, you can choose more than one control to achieve the results you want. For example, if you want to display text that cannot be edited by the user, you can use a label, link label, read-only text box, or read-only rich text box. However, we will see in this chapter that not all of these allow the user to copy text to the Clipboard, and only one of these displays a bulleted list of items.

It is important to familiarize yourself with the built-in controls and their general functions so you can choose the right control for a particular application. Knowing the limitations of the built-in controls also allows you to identify applications for which no built-in control is appropriate. These applications call for custom Windows controls.

In this section, we will look closely at a number of controls built into Visual Basic .NET. Table 8.1 displays the controls discussed in this chapter according to their general function.

Table 8.1 Windows Forms Controls by Function

Control	Function
Informational Only	
Label	Displays text that cannot be edited by the user
LinkLabel	Displays text that is a link to another window or Web site
Text Edit	
TextBox	Displays text that can be edited by the user
RichTextBox	Displays text in Rich Text Format (RTF)
Selection from List	
ComboBox	Displays a drop-down list of items
DomainUpDown	Displays a list of text items through which the user can scroll with a spin button
NumericUpDown	Displays a list of numeric items through which the user can scroll with a spin button
ListBox	Displays a list of items
ListView	Displays text in a text-only, text-with-small-icons, text-with-large-icons, or report view
TreeView	Displays hierarchical information
Graphics Display	
PictureBox	Displays graphics

Continued

Table 8.1 Continued

Control	Function
Value Setting	
CheckBox	Presents options that are not mutually exclusive
CheckedListBox	Displays a list of items, each along with a check mark
RadioButton	Presents mutually exclusive options
TrackBar	Displays a scale on which the user can set a value
Date Setting	
DateTimePicker	Displays a graphical calendar from which the user can set a date
Command Controls	
Button	Starts, stops, or interrupts a process
Grouping Controls	
Panel	Holds a scrollable group of controls
GroupBox	Holds a captioned group of controls
Tab	Provides a tabbed page for organizing and efficiently accessing grouped objects

There are some properties that are shared by many controls, like the *Name* property, which indicates the name used in code to identify the object. Table 8.2 displays common properties of controls. As we look at each control in the following sections, we will discuss the properties unique to them.

Table 8.2 Common Properties of Controls

Property	Description
(Bindings)	This collection holds all the bindings of properties of this control to data sources
(Name)	Indicates the name used in code to identify the object
AccessibleDescription	The description that will be reported to accessibility clients
AccessibleName	The name that will be reported to accessibility clients

Continued

Table 8.2 Continued

Property	Description
AccessibleRole	The role that will be reported to accessibility clients
Anchor	The anchor of the control. Anchors define to which edges of the container a certain control is bound. When a control is anchored to an edge, the distance between the control's closest edge and the specified edge will remain constant
CausesValidation	Indicates whether this control causes and raises validation events
ContextMenu	The shortcut menu to display when the user right-clicks the control
Dock	The docking location of the control, indicating which borders are docked to the container
Enabled	Indicates whether the control is enabled
IMEMode	Determines the IME (Input Method Editor) status of the object when selected
Location	The position of the top-left corner of the control with respect to its container
Locked	Determines if the control can be moved or resized
Modifiers	Indicates the visibility level of the object
Size	The size of the control in pixels
Visible	Determines whether the control is visible or hidden

Label Control

The Windows Forms Label control allows you to display text that cannot be edited by the user. You can use labels to add descriptive captions to other controls to help identify their purpose. It is common practice to provide labels for controls that do not have a label themselves, such as text boxes, including a colon in the label's caption.

Labels cannot receive focus, but you can use labels to quickly move focus to other controls by creating access keys. When you use a label to create an access key, the user can press the **Alt** key, plus the character you designate, to move the focus to the control that follows the label in the tab order.

You can also use labels to display information about runtime events or processes in your application. For example, in an e-mail manager application, you can use a label to inform the user when your application is connecting to the mail server, checking for new messages, and disconnecting from the mail server. The *Text* property is the default property of a label. Table 8.3 shows other properties of the Label control.

Table 8.3 Label Properties

Property	Description
AllowDrop	Determines if the label will receive drag-drop notifications
AutoSize	Enables automatic resizing based on font size. Note that this is only valid for labels that don't wrap text
BackColor	The background color used to display text and graphics in the label
BorderStyle	Determines if the label has a visible border
Cursor	The cursor that appears when the mouse passes over the label
Font	The font used to display text in the label
ForeColor	The foreground color used to display text and graphics in the label
Image	The image that will be displayed on the face of the label
ImageAlign	The alignment of the image that will be displayed in the face of the label
ImageIndex	The index of the image in the image list to display in the face of the label
ImageList	The image list to get the image to display in the face of the label
RightToLeft	Indicates whether the label should draw right-to-left for RTL languages
TabIndex	Determines the index in the tab order that this label will occupy
Text	The text contained in the label
TextAlign	Determines the position of the text within the label
UseMnemonic	If True, the first character preceded by an ampersand (&) will be used as the label's mnemonic key

A foremost property of the Label control is the *Text* property, which contains the label's caption. In previous versions of Visual Basic, the *Caption* property was used to specify the text displayed in the label. As with most properties of the label, you can set the caption at design-time or runtime. To set a label's caption at design-time, use the Properties window to set the *Text* property to an appropriate string. To set a label's caption at runtime, set the *Text* property programmatically, as shown next:

```
'Set the label's caption
lblStatus.Text = "Finding Server..."
```

The *AutoSize* property helps you size a label to fit smaller or larger captions, which is useful if the caption changes run-time. Use the *AutoSize* property to fix the size of a label or make it dynamically resize to hold the value of the *Text* property. To make a label dynamically resize to hold its text, set the *AutoSize* property to True:

```
'Make the label dynamically resize to fit its contents
lblStatus.AutoSize = True
```

NOTE

If the *AutoSize* property is set to False, the text wraps to the next line, but the label does not grow.

The *TextAlign* property helps you change the horizontal alignment of text within a label. You can choose to align a label's caption with the left margin or right margin of the label, or to center the caption within a label's margins. For example, to center a label's caption within its margins, set the *TextAlign* property as shown in the following:

```
'Center text within the label's caption
lblStatus.TextAlign = HorizontalAlignment.Center
```

You can use labels to assign access keys to other controls. When you use a label to create an access key, the user can press the ALT key, plus the character you designate, to move the focus to the control that follows the label in the tab order. Since labels cannot receive focus, focus automatically moves to the control that follows it in the tab order.

LinkLabel Control

The Windows Forms LinkLabel control allows you to add Web-style links to your Windows Forms applications. The LinkLabel control retains all properties, methods, and events of the Label control, and you can use a link label for everything with which you can use a label. In addition, the LinkLabel control allows you to set part of its caption as a link to an object or Web page. For instance, you can use a link label in the About box of your application to provide a link to your company's Web page. Table 8.4 shows the properties of the LinkLabel control, in addition to the properties of the Label control.

Table 8.4 LinkLabel Properties

Property	**Description**
ActiveLinkColor	Determines the color of the hyperlink when the user is clicking the link
DisabledLinkColor	Determines the color of the hyperlink when disabled
LinkArea	Portion of the text in the label to render as a hyperlink
LinkBehavior	Determines the underline behavior of the hyperlink
LinkColor	Determines the color of the hyperlink
LinkVisited	Determines if the hyperlink should be rendered as visited
VisitedLinkColor	Determines the color of the hyperlink when the *LinkVisited* property is set to True

The *ActiveLinkColor*, *DisabledLinkColor*, *LinkColor*, and *VisitedLinkColor* properties determine the color of the link. For example, when the link is clicked, you can change its color to indicate it has been visited. The *LinkColor* property determines the color of the link when it is in its default state—when the user is not clicking the link, the link is not disabled, and the link is not rendered as visited. When the user is clicking the link, its color is determined by the *ActiveLinkColor* property. The *DisabledLinkColor* property determines the color of the link when it is disabled. When the link is rendered as visited, the *VisitedLinkColor* property determines the color of the link. To change the color of a link using defined color constants, set the *ActiveLinkColor*, *DisabledLinkColor*, *LinkColor*, and *VisitedLinkColor* properties as shown next:

```
'Set the link color using defined color constants
with lnkWebSite
    .ActiveLinkColor = color.Red
    .DisabledLinkColor = color.Blue
    .LinkColor = color.Blue
    .VisitedLinkColor = color.Purple
end with
```

You can also change the color of a link using decimal values for red, green, and blue. For instance, you can change the color of the same link using decimal values for red, green, and blue as shown in the following:

```
'Set the link color using decimal values for red, green, and blue
With lnkWebSite
    .ActiveLinkColor = Color.FromARGB(255, 0, 0)
    .DisabledLinkColor = Color.FromARGB(0, 0, 255)
    .LinkColor = Color.FromARGB(0, 0, 255)
    .VisitedLinkColor = Color.FromARGB(128, 0, 128)
End With
```

The *LinkArea* property holds the portion of the link label's caption that activates the link. The *LinkArea.X* property determines the start of the link area, and the *LinkArea.Y* property determines the length of the link area. In the following example, the link area starts before the first character of the link label's caption, and ends after the last character of the caption:

```
'Set the link area to all of the caption
With lnkWebSite
    .LinkArea.X = 0
    .LinkArea.Y = Len(linklabel1.Text)
End With
```

You can choose, however, to only use part of the link label to activate the link. For example, to have only the third character of the caption to comprise the link area, set the *LinkArea* property as follows:

```
With lnkWebSite
    .LinkArea.X = 3
    .LinkArea.Y = 1
End With
```

It is common to indicate that a link can be clicked by showing an underline with the link. The *LinkBehavior* property determines when the link shows an underline, be it always, never, when the mouse hovers over the link, or the system default setting. For example, to set the link to always appear with an underline, set the *LinkBehavior* property as follows:

```
'Set the link to always appear with an underline
lnkWebSite.LinkBehavior = LinkBehavior.AlwaysUnderline
```

Another useful property of the link label is the *LinkVisited* property, which determines when the link is to be marked as visited. You may, for instance, have a link label on a form with which you want to display another form. After the second form is displayed, you want to give the user an indication that the form was displayed. To mark the link as visited in the color specified by the *VisitedLinkColor* property, set the *LinkVisited* property to True:

```
Protected Sub lnkWebSite_LinkClicked(ByVal sender As Object,
ByVal e As System.EventArgs)
    Dim frmDetails As New Form()

    'Display another form
    frmDetails.Show()

    'Mark the link as visited
    lnkWebSite.LinkVisited = True
End Sub
```

The *Click* event of the link label is an important one, since the *Click* event determines what happens when the link is selected. Use the LinkClicked event handler to take appropriate action when the link is clicked. To start the default Web browser and link to a Web page:

1. In the LinkClicked event handler, use the *Process.Start* method to start the default browser with a URL. To use the *Process.Start* method, you need to add a reference to the *System.Diagnostics* namespace.
2. Set the *LinkVisited* property to True.

The following example shows how to start the default Web browser and link to the Microsoft Web site:

```
Private Sub lnkWebSite_LinkClicked (ByVal Sender As Object, _
ByVal e As EventArgs)
    'Open the default Web browser and link to the Microsoft Web site
    System.Diagnostics.Process.Start("http://www.microsoft.com")

    'Mark the link as visited
    lnkWebSite.LinkVisited = True
End Sub
```

TextBox Control

The Windows Forms TextBox control allows you to display text to the user and collect text from the user. You can also use text boxes to add basic formatting to your application such as password text boxes. A password text box is one that displays a placeholder character instead of each character entered. Text boxes can display multiple lines. By default, a text box holds up to 2,048 characters, but when displaying multiple lines, a text box holds up to 32K of text. Text boxes are most commonly used for editable text, but they can also be made read-only. Table 8.5 shows the properties of the TextBox control.

Table 8.5 TextBox Properties

Property	Description
AcceptsReturn	Indicates if return characters are accepted as input for multiline edit controls
AcceptsTab	Indicates if tab characters are accepted as input for multiline edit controls
AutoSize	Enables automatic resizing based on font size for single-line edit controls
BackColor	The background color used to display text and graphics in the control
BorderStyle	Indicates whether or not the edit control should have a border
CharacterCasing	Indicates if all characters should be left alone or converted to uppercase or lowercase
Cursor	The cursor that appears when the mouse passes over the control
Font	The font used to display text in the control

Continued

Table 8.5 Continued

Property	Description
ForeColor	The foreground color used to display text and graphics in the control
HideSelection	Indicates that the selection should be hidden when the edit control loses focus
Lines	The lines of text in a multiline edit, as in an array of string values
MaxLength	Specifies the maximum number of characters that can be entered into the edit control. Zero implies no maximum
MultiLine	Controls whether the text of the edit control can span more than one line
PasswordChar	Indicates the character to display for password input for single-line edit controls
ReadOnly	Controls whether the text in the edit control can be changed or not
RightToLeft	Indicates whether the control should draw right-to-left for RTL languages
ScrollBars	Indicates, for multi-line edit controls, which scroll bars will be shown for this control
TabIndex	Determines the index in the tab order that this control will occupy
TabStop	Indicates whether the user can use the **Tab** key to give focus to the control
Text	The text contained in the control
TextAlign	Indicates how the text should be aligned for edit controls
WordWrap	Indicates if lines are automatically word-wrapped for multi-line edit controls

NOTE

Use a Label control with the TextBox control to help the user identify the purpose of the TextBox control and to indicate when it is disabled.

You can display multiple lines in a TextBox control by using the *MultiLine*, *WordWrap*, and *ScrollBars* properties. To display multiple lines in a TextBox control:

1. Set the *MultiLine* property to True.
2. Set the *ScrollBars* property to None, Horizontal, or Both.
3. Set the *WordWrap* property to False or True.

Now let's see how this is done in code:

```
With txtTextBox
    'Set the MultiLine property to True
    .MultiLine = True

    'Set the ScrollBars property
    .ScrollBars = vbBoth

    'Set the WordWrap property
    .WordWrap = True
End With
```

You can create a read-only text box by using the *ReadOnly* property. If the *ReadOnly* property is set to True, the **Copy** command is available, but the **Cut** and **Paste** commands are unavailable.

NOTE

Create a read-only text box by using a TextBox control instead of a Label control when you want to allow the user to select the text. For example, if you use a TextBox control to create a read-only text box, the user can copy the text to the Clipboard.

You can use the *MaxLength* and *PasswordChar* properties to create a password text box that displays a placeholder character instead of each character entered. To create a password text box:

1. Set the *PasswordChar* property to a placeholder character.

2. Set the *MaxLength* property to the maximum number of characters allowed. If the user attempts to exceed this number, the system emits a beep and does not accept any more characters.

Now let's see how this is done in code:

```
With txtTextBox
    'Set PasswordChar property to the asterisk
    .PasswordChar = "*"

    'Set the MaxLength property to 10
    .MaxLength = 10
End With
```

There are several ways to programmatically place quotation marks in the text of a TextBox control. You can use an additional set of quotation marks, use the ASCII character (34), or define a constant for the quotation marks character, as shown in the following example:

```
With txtTextBox
    'Insert an additional set of quotation marks
    .Text = """Wow, five!"" the woman said."

    'Use the ASCII character (34)
    .Text = Chr(34) & "Wow, five!" & Chr(34) _
        & " the woman said."

    'Define a constant for the quotation marks character
    Const strQuotationMarks = """"

    .Text = strQuotationMarks & "Wow, five!" _
        strQuotationMarks & " the woman said."
End With
```

The default insertion point for a Windows Forms TextBox control is to the left of any text when it first receives the focus. After the focus moves away from, and then back to, the TextBox control, the insertion point is at the position where the user last placed it. You can use the *SelectionStart* and *SelectionLength* properties to control the insertion point to, for instance, select all existing text to speed data entry:

```
With txtTextBox
    'Set the SelectionStart property to the left of any
    'text
    .SelectionStart = 0

    'Set the SelectionLength property to select all text
    .SelectionLength = Len(txtTextBox)
End With
```

Button Control

The Windows Forms Button control performs an action when the button is clicked, making it look as if the button is being pushed in and released. When the user clicks the button, the Click event handler is invoked. To respond to a button click, write code in the button's Click event handler, as shown in the following example:

```
Private Sub btnCancel_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs)
   frmForm.Hide()
End Sub
```

NOTE

The button does not support a double-click event. If the user attempts to double-click a button, its Click event handler will be invoked twice if the button is still visible and available after the first click.

Like the other controls we discussed, the Button has a number of properties that are not among the properties common to all built-in controls. Table 8.6 shows the properties of the Button control.

Table 8.6 Button Properties

Property	Description
BackColor	The background color used to display text and graphics in the control
BackgroundImage	The background image used for the control
Cursor	The cursor that appears when the mouse passes over the control
DialogResult	The dialog result produced in a modal form by clicking the button
FlatStyle	Determines the display of the button when users move the mouse over the control and click
Font	The font used to display text in the control
ForeColor	The foreground color used to display text and graphics in the control
Image	The image that will be displayed on the face of the control
ImageAlign	The alignment of the image that will be displayed in the face of the control
ImageIndex	The index of the image in the image list to display in the face of the control
ImageList	The image list to get the image to display in the face of the button
RightToLeft	Indicates whether the button should draw right-to-left for RTL languages
TabIndex	Determines the index in the tab order that the button will occupy
TabStop	Indicates whether the user can use the **Tab** key to give focus to the button
Text	The text contained in the button
TextAlign	The alignment of the text that will be displayed in the face of the button

The user can click the Button control in several ways:

- The user uses a mouse to click the button.
- The user chooses the button by pressing the **Spacebar** or **Enter** key when the button has the focus.

- The user presses **Enter** when the button is the "accept" button of the form.
- The user presses **Esc** when the button is the "cancel" button of the form.
- The user presses the access key (**Alt + the underlined letter**) for the button.

In addition, you can click the Button control programmatically in the following ways:

- Invoke the button's *Click* event.
- Call the *PerformClick* method.

When you designate a Button control to be the accept button on the form, the button is clicked when the user presses **Enter**, even if another control has the focus—except when that other control is another button or a multiline text box. This basically makes it the default button on the form. When you designate a Button control to be the cancel button on the form, the button is clicked when the user presses **Esc,** even if another control has the focus. A button can be both the accept button *and* the cancel button on the form. The following code shows you how to do this:

```
With frmForm
    'Designate btnButton as the accept button and the
    'cancel button of frmForm

    'Set the AcceptButton property to btnButton
    .AcceptButton = btnButton

    'Set the CancelButton property to btnButton
    .CancelButton = btnButton
End With
```

NOTE

If the action represented by a Button control requires additional information, such as the folder in which to save a document, include an ellipsis (…) in the button's *Caption* property.

CheckBox Control

The Windows Forms CheckBox control indicates True/False or Yes/No options. The CheckBox control appears as a square box with an accompanying label. When a choice is set, a check mark appears in the box. When the choice is not set, the box is empty. Use the CheckBox control to present an independent or non-exclusive choice, a True/False or Yes/No selection to the user. You can group multiple check boxes using a GroupBox control to display multiple choices from which the user may select more than one.

The CheckBox control has two important properties: *Checked* and *CheckedState*. The *Checked* property returns either True or False, and indicates whether the choice is set. The *CheckedState* property returns CheckState.Checked when the choice is set, and CheckState.Unchecked when the choice is not set. If the *ThreeState* property is set to True, the *CheckedState* property also returns CheckState.Indeterminate, used when the choice is set for some but not all elements of the selection.

Since grouped controls can be moved around together on the form designer, group multiple boxes using the GroupBox control to enhance visual appearance and aid in GUI design Table 8.7 shows the properties of the CheckBox control.

Table 8.7 CheckBox Properties

Property	Description
AllowDrop	Determines if the check box will receive drag-drop notifications
Appearance	Controls the appearance of the check box
AutoCheck	Causes the check box to automatically change state when clicked
BackColor	The background color used to display text and graphics in the check box
BackgroundImage	The background image used for the check box
CheckAlign	Determines the location of the check box inside the control
Checked	Indicates whether the check box is checked or unchecked
Cursor	The cursor that appears when the mouse passes over the check box
FlatStyle	Determines the display of the check box when users move the mouse over the check box and click

Continued

Table 8.7 Continued

Property	Description
Font	The font used to display text in the check box
ForeColor	The foreground color used to display text and graphics in the check box
Image	The image that will be displayed on the face of the check box
ImageAlign	The alignment of the image that will be displayed in the face of the check box
ImageIndex	The index of the image in the image list to display in the face of the check box
ImageList	The image list to get the image to display in the face of the check box
RightToLeft	Indicates whether the check box should draw right-to-left for RTL languages
TabIndex	Determines the index in the tab order that the check box will occupy
TabStop	Indicates whether the user can use the **Tab** key to give focus to the check box
Text	The text contained in the check box
TextAlign	The alignment of the text that will be displayed in the face of the check box
ThreeState	Controls whether or not the user can select the indeterminate state of the check box

RadioButton Control

The Windows Forms RadioButton control is used to give the user a single choice within a set of two or more mutually exclusive choices. Radio buttons appear as a set of small circles. When an option button choice is set, a dot appears in the middle of the circle. When the item is not the current setting, the circle next to the current setting is empty. If the user chooses any radio button in a group, that value becomes the setting for the group; the dot appears in that button and all the other buttons in the group remain empty. Group radio buttons by adding them to a form. To add separate groups, you need to add them to a Panel control or a GroupBox control.

The radio button and the check box are used for different functions. Use a radio button when you want the user to choose only one option. When you want the user to choose all appropriate options, use a check box. Table 8.8 shows the properties of the RadioButton control.

Table 8.8 RadioButton Properties

Property	Description
AllowDrop	Determines if the radio button will receive drag-drop notifications
Appearance	Controls whether the radio button appears as normal or as a Windows push button
AutoCheck	Causes the radio button to automatically change state when clicked
BackColor	The background color used to display text and graphics in the radio button
BackgroundImage	The background image used for the radio button
CheckAlign	Determines the location of the check box inside the radio button
Checked	Indicates whether the radio button is checked or not
Cursor	The cursor that appears when the mouse passes over the radio button
FlatStyle	Determines the display of the radio button when users move the mouse over the radio button and click
Font	The font used to display text in the radio button
ForeColor	The foreground color used to display text and graphics in the radio button
Image	The image that will be displayed on the face of the radio button
ImageAlign	The alignment of the image that will be displayed in the face of the radio button
ImageIndex	The index of the image in the image list to display in the face of the radio button
ImageList	The image list to get the image to display in the face of the radio button
RightToLeft	Indicates whether the radio button should draw right-to-left for RTL languages
TabIndex	Determines the index in the tab order that the radio button will occupy

Continued

Table 8.8 Continued

Property	Description
TabStop	Indicates whether the user can use the **Tab** key to give focus to the radio button
Text	The text contained in the radio button
TextAlign	The alignment of the text that will be displayed in the face of the radio button

RichTextBox Control

Much like the TextBox control, the Windows Forms RichTextBox control allows you to display text to the user and collect text from the user. In addition to the many features it shares with the TextBox control, the RichTextBox control allows you to change the font, size, and color of text. You can use a rich text box to add advanced formatting, such as indents, hanging indents, and bulleted paragraphs, to your application. You can also use a rich text box to save text to a file, or load text from a file. Table 8.9 shows the properties of the RichTextBox control.

Table 8.9 RichTextBox Properties

Property	Description
AcceptsTab	Indicates if tab characters are accepted as input for the rich text box
AutoSize	Enables automatic resizing based on font size for a single-line rich text box
AutoWordSelection	Turns on/off automatic word selection
BackColor	The background color used to display text and graphics in the rich text box
BorderStyle	Indicates whether or not the rich text box should have a border
BulletIndent	Defines the indent for the bullets in the rich text box
Cursor	The cursor that appears when the mouse passes over the rich text box
Delimiter	Defines the delimiter characters (Asian version of OS only)
DetectURLs	Turns on/off automatic URL highlighting

Continued

Table 8.9 Continued

Property	Description
FollowPunctuation	Defines the non-leading punctuation (Asian version of OS only)
Font	The font used to display text in the rich text box
ForeColor	The foreground color used to display text and graphics in the rich text box
HideSelection	Indicates that the selection should be hidden when the rich text box loses focus
LeadPunctuation	Defines the leading punctuation (Asian version of OS only)
Lines	The lines of text in a multi-line rich text box, as in an array of string values
MaxLength	Specifies the maximum number of characters that can be entered into the rich text box. Zero implies no maximum
Multiline	Controls whether the text of the rich text box can span more than one line
OutlineMode	Turns on/off outline mode
ReadOnly	Controls whether the text in the rich text box can be changed or not
RightMargin	Defines the right margin dimensions
RightToLeft	Indicates whether the rich text box should draw right-to-left for RTL languages
ScrollBars	Defines the behavior of the scroll bars of the rich text box
SelectionMargin	Turns on/off the selection margin
TabIndex	Determines the index in the tab order that the rich text box will occupy
TabStop	Indicates whether the user can use the **Tab** key to give focus to the rich text box
Text	The text contained in the rich text box
WordBreak	Enables the word-break mode (Asian version of OS only)
WordPunctuation	Defines the type of punctuation table to be used for word operations (Asian version of OS only)
WordWrap	Indicates if lines are automatically word-wrapped for multi-line rich text boxes

You can change the font, size, and color of text in the RichTextBox control by using the *SelFont*, *SelFontSize*, and *SelColor* properties. Also, you can quickly open a Rich Text Format file using a rich text box. This is handy when you want your application to display a README file that changes with every version of your application. You make changes to the file, not your code, and you can simply open the file with a rich text box. Use an OpenFileDialog control to navigate to the file you want to open, as in the following example:

```
With OpenFileDialog1
     'Show only RTF files
     .Filter = "Rich Text Format|*.rtf"
     .ShowDialog()
End With

richtextbox2.LoadFile(openfiledialog1.FileName)
```

Often, when opening files, you only want to show those with a certain extension. This makes it easier for the user to locate a file. You can do this with the *Filter* property of the Open File dialog box:

```
.Filter = "Rich Text Format|*.rtf"
```

Use this line to show only RTF files in the Open File dialog box. Files with other extensions will not appear.

TreeView Control

The Windows Forms TreeView control is used to display hierarchical information, such as e-mail folders and messages, as well as folders and files on a computer. A tree view contains cascading branches of nodes, and each node consists of an image and a label. Node images are taken from an ImageList control. At the top-most level in a tree view are root nodes that can be expanded or collapsed if the nodes have child nodes (nodes that descend from other nodes). Table 8.10 shows the properties of the TreeView control.

Table 8.10 TreeView Properties

Property	Description
AllowDrop	Determines if the tree view will receive drag-drop notifications

Continued

Table 8.10 Continued

Property	Description
BackColor	The background color used to display text and graphics in the tree view
BorderStyle	The border style of the tree view
CheckBoxes	Indicates whether check boxes are displayed beside nodes
Cursor	The cursor that appears when the mouse passes over the tree view
Font	The font used to display text in the tree view
ForeColor	The foreground color used to display text and graphics in the tree view
FullRowSelect	Indicates whether the highlight spans the width of the tree view
HideSelection	Removes highlight from the selected node when the tree view loses focus
HotTracking	Indicates whether nodes give feedback when the mouse is moved over them
ImageIndex	The default image index for nodes
ImageList	The image list from which node images are taken
Indent	The indentation width of child nodes in pixels
ItemHeight	The height of every item in the tree view
LabelEdit	Indicates whether or not the user can edit the label text of nodes
Nodes	The root nodes in the tree view
PathSeparator	The string delimiter used for the path returned by a node's *FullPath* property
RightToLeft	Indicates whether the control should draw right-to-left for RTL languages
Scrollable	Indicates whether the tree view will display scroll bars if it contains more nodes than can fit in the visible area
SelectedImage	The default image index for selected nodes
ShowLines	Indicates whether lines are displayed between sibling nodes and between parent and child nodes
ShowPlusMinus	Indicates if plus/minus buttons are shown next to parent nodes

Continued

Table 8.10 Continued

Property	Description
ShowRootLines	Indicates whether lines are displayed between root nodes
Sorted	Indicates whether nodes are sorted
TabIndex	Determines the index in the tab order that this control will occupy
TabStop	Indicates whether the user can use the **Tab** key to give focus to the tree view

ListBox Control

A Windows Forms ListBox control displays a list of choices which the user can select from. List boxes are best used for displaying large number of choices. There are situations in which you can display choices with either a group of check boxes or a list box. In general, use a group of check boxes when the number of choices is small. For clarity, use a list box when the number of choices is large.

A vertical scroll bar accompanies a list box if the items displayed exceed the height of the box. A list box will also sport a horizontal scrollbar if the *MultiColumn* property is set to True. In that case, values are displayed in columns horizontally. Table 8.11 shows the properties of the ListBox control.

Table 8.11 ListBox Properties

Property	Description
AllowDrop	Determines if the list box will receive drag-drop notifications
BackColor	The background color used to display text and graphics in the list box
BorderStyle	Controls what type of border is drawn around the list box
ColumnWidth	Indicates how wide each column should be in a multi column list box
Cursor	The cursor that appears when the mouse passes over the list box
DataSource	Indicates the list that the list box will use to get its items

Continued

Table 8.11 Continued

Property	Description
DisplayMember	Indicates the property to display for the items in the list box
DrawMode	Indicates whether the system or the user paints items in the list box
Font	The font used to display text in the list box
ForeColor	The foreground color used to display text and graphics in the list box
HorizontalExtent	The width, in pixels, by which a list box can be scrolled horizontally. Only valid if HorizontalScrollbar is True
HorizontalScrollbar	Indicates whether the list box will display a horizontal scrollbar for items beyond the right edge of the list box
IntegralHeight	Indicates whether the list can contain only complete items
ItemHeight	The height, in pixels, of items in a fixed-height owner-drawn list box
Items	The items in the list box
MultiColumn	Indicates if values should be displayed in columns horizontally
RightToLeft	Indicates whether the list box should draw right-to-left for RTL languages
ScrollAlwaysVisible	Indicates if the list box should always have a scrollbar present, regardless of how many items are in it
SelectionMode	Indicates if the list box is to be single-select, multi-select, or unselectable
Sorted	Controls whether the list is sorted
TabIndex	Determines the index in the tab order that the list box will occupy
TabStop	Indicates whether the user can use the **Tab** key to give focus to the list box
UseTabStops	Indicates if tab characters should be expanded into full spacing
ValueMember	Indicates the property to use as the actual value for the items in the list box

The *SelectionMode* property determines how many items in the list can be selected at a time. If the *SelectionMode* property is set to SelectionMode.MultiSimple, the user can select more than one item by simply clicking the items in the list. If the *SelectionMode* property is set to SelectionMode.MultiExtended, the user can select more than one item by holding down the **Ctrl** key or **Shift** key and clicking items in the list.

As with other controls, the *SelectedIndex* property holds the index of the selected item in the list box. If more than one item is selected, the *SelectedIndex* property contains the index of the first selected item in the box. Keep in mind that the *SelectedIndex* property is zero-based like other index properties in VB .NET. For example, the *SelectedIndex* property equals 0 when the first item in a list box is selected.

You can use the Add or Insert method to add items to a list box. The Add method adds new items at the end of an unsorted list box. The Insert method allows you to specify where to insert the item you are adding. In the following example, the name "John Doe" is added to a list box first. Then the name "Jane Doe" is inserted at the first position in the list, rather than at the last position:

```
'Add the name "John Doe" to the Employees list box
lstEmployees.Items.Add("John Doe")


'Insert the name "Jane Doe" at the first position in the list box
lstEmployees.Items.Insert(0, "Jane Doe")
```

You can use the Remove method to remove an item from a list box. Continuing our example, the following statement removes the first name from the list box:

```
'Remove the first name from the Employees list box
lstEmployees.Items.Remove(0)
```

You can also quickly remove all items from a list box. To remove all items from a list box, simply use the Clear method of the Items collection.

```
'Remove all names from the Employees list box
lstEmployees.Items.Clear()
```

CheckedListBox Control

The Windows Forms CheckedListBox control, an extension of the ListBox control, gives you all the capability of a list box and also allows you to display a check mark next to the items in the list box. Use the checked list box instead of the list box to display additional information about the items you display. For example, a checked list box is a good choice to display steps of an installation, with the check marks indicating which steps have been completed. Table 8.12 shows the properties of the CheckedListBox control.

Table 8.12 CheckedListBox Properties

Property	Description
AllowDrop	Determines if the checked list box will receive drag-drop notifications
BackColor	The background color used to display text and graphics in the checked list box
BorderStyle	Controls what type of border is drawn around the checked list box
CheckOnClick	Indicates if the check box should be toggled with the first click of an item
ColumnWidth	Indicates how wide each column should be in a multi column checked list box
Cursor	The cursor that appears when the mouse passes over the checked list box
DataSource	Indicates the list that the checked list box will use to get its items
DisplayMember	Indicates the property to display for the items in the checked list box
Font	The font used to display text in the checked list box
ForeColor	The foreground color used to display text and graphics in the checked list box
HorizontalExtent	The width, in pixels, by which a list box can be scrolled horizontally (Only valid if HorizontalScrollBar is True)
HorizontalScrollbar	Indicates whether the checked list box will display a horizontal scrollbar for items beyond the right edge of the checked list box
IntegralHeight	Indicates whether the list can contain only complete items

Continued

Table 8.12 Continued

Property	Description
Items	The items in the checked list box
MultiColumn	Indicates if values should be displayed in columns horizontally
RightToLeft	Indicates whether the control should draw right-to-left for RTL languages
ScrollAlwaysVisible	Indicates if the checked list box should always have a scrollbar present, regardless of how many items are in it
SelectionMode	Indicates if the checked list box is to be single-select, multiselect, or unselectable
Sorted	Controls whether the checked list box is sorted
TabIndex	Determines the index in the tab order that the checked list box will occupy
TabStop	Indicates whether the user can use the **ab** key to give focus to the control
ThreeDCheckBoxes	Indicates whether the check values should be shown as flat or 3D check marks
UseTabStops	Indicates if tab characters should be expanded into full spacing
ValueMember	Indicates the property to use as the actual value for the items in the checked list box

You can change the appearance of the check boxes that appear next to the items by using the *ThreeDCheckBoxes* property. The check boxes can appear as flat or 3D check marks. The *CheckOnClick* property determines if the items are toggled with the first click of an item. To toggle items with the first click, set the *CheckOnClick* property to True.

You can quickly add multiple items to a checked list box. Adding multiple items to a checked list box is a multistep process. To add items to a checked list box:

1. Create an array of type *System.Object*.
2. Set each member of the array to a string—the string becomes the text displayed in the list.
3. Set the Items collection's *All* property to the array.

For example, suppose you want to display the days of the week in a checked list box. Follow the steps just outlined in the manner shown next:

```
'Create an array of type System.Object
Dim objDaysOfTheWeek(7) As System.Object

'Set each member of the array to a string
objDaysOfTheWeek(0) = "Monday"
objDaysOfTheWeek(1) = "Tuesday"
objDaysOfTheWeek(2) = "Wednesday"
objDaysOfTheWeek(3) = "Thursday"
objDaysOfTheWeek(4) = "Friday"
objDaysOfTheWeek(5) = "Saturday"
objDaysOfTheWeek(6) = "Sunday"

'Set the Items collection's All property to the array
clbDaysOfTheWeek.Items.All = objDaysOfTheWeek
```

ListView Control

The ListView control can display text in four different views: text-only, text-with-small-icons, text-with-large-icons, or report. A list view is similar to the right pane that displays the contents of the selected folder in Windows Explorer. Just as you can use the View menu in Windows Explorer to change how the icons in the right pane appear, you can change how items in a list view appear.

The list view comes in handy when you want to display items that contain multiple pieces of information, because it can show more than one column. For example, you can use a list view on an About box to display version information of key files used by your application. Your list view would have three columns with the headings “File,” “Version,” and “Full Path.” All the information you need to support your user would be readily available. We will look at code snippets that allow you to create a version information box. Table 8.13 shows the properties of the ListView control.

Table 8.13 ListView Properties

Property	Description
Activation	Indicates the type of action required by the user to activate an item, and the feedback given
Alignment	Indicates how items are aligned within the list view
AllowColumnReorder	Indicates whether the user can reorder columns in the Report view
AllowDrop	Determines if the list view will receive drag-drop notifications
AutoArrange	Indicates whether items are kept arranged automatically
BackColor	The background color used to display text and graphics in the list view
BorderStyle	The border style of the list view
CheckBoxes	Indicates whether check boxes are displayed beside items
Columns	The columns shown in Report view
Cursor	The cursor that appears when the mouse passes over the list view
Font	The font used to display text in the list view
ForeColor	The foreground color used to display text and graphics in the list view
FullRowSelect	Indicates whether all subitems are highlighted along with the item when selected
GridLines	Displays grid lines around items and subitems
HeaderStyle	The style of the column headers in Report view
HideSelection	Removes highlighting from the selected item when the list view loses focus
HoverSelection	Allows items to be selected by hovering over them with the mouse
LabelEdit	Allows item labels to be edited in place by the user
LabelWrap	Determines whether label text can wrap to a new line
LargeImageList	The image list used by the list view for images in Large Icon view
ListItems	The items in the list view
MultiSelect	Allows multiple items to be selected

Continued

Table 8.13 Continued

Property	Description
RightToLeft	Indicates whether the list view should draw right-to-left for RTL languages
Scrollable	Indicates whether the list view will display scrollbars if it contains more items than can fit in the client area
SmallImageList	The image list used by the list view for images in all views except for the Large Icon view
Sorting	Indicates the manner in which items are to be sorted
StateImageList	The image list used by the list view for custom states
TabIndex	Determines the index in the tab order that the list view will occupy
TabStop	Indicates whether the user can use the **Tab** key to give focus to the list view
View	Selects one of four different views in which items can be shown

The *View* property is an important property of the list view, because it determines one of four different views in which items can be shown: text-only, text-with-small-icons, text-with-large-icons, or report. The self-explanatory text-only view renders items in much the same manner as a list box. The text-with-small-icons, text-with-large-icons, and report views are similar to the Small Icons, Large Icons, and Detail views in Windows Explorer.

The *ListItems* property holds the items in a list view. If a ListView control has multiple columns, the items have subitems that hold information in the columns beyond the first. For example, a list view with one row and three columns has one item (to hold the information in the first column) and two subitems (to hold the information in the second and third columns).

Another useful property of the list view is the *HeaderStyle* property. The *HeaderStyle* property determines the style of the column headers in Report view. The property can be set to show no headers, clickable headers, and non-clickable headers.

Before we delve into our example, you should be familiar with the *Sorting* property of the list view. The *Sorting* property indicates the manner in which items are to be sorted. Items can remain unsorted, or you can sort them by ascending or descending order.

Let's create our version information box. Our box will have three columns to show a file's name, its version, and its full path. Declare three variables to hold these values:

```
Dim strFile As String
Dim strVersion As String
Dim strFullPath As String
```

Declare an array of strings to hold subitems in the list view. The array has size two to show two columns in addition to the first in the list view.

```
Dim strSubItems(2) As String
```

Our ListView control will show information about the common control DLL comctl32.dll. The following statement assigns the appropriate values to the variables we declared:

```
'Set the file's name, version, and full path
strFile = "comctl32.dll"
strVersion = "5.81"
strFullPath = "C:\WINNT\system32\COMCTL32.DLL"
```

We now need to fill our subitem array with the file's version and full path. These will be displayed in the second and third columns of the list view:

```
'Fill the subitem array
strSubItems(0) = strVersion
strSubItems(1) = strFullPath
```

We are now ready to set properties of the list view. We want to display the items in Report view, have clickable headers, and display items in ascending order:

```
With lvwVersionInformation
   .View = View.report
   .HeaderStyle = ColumnHeaderStyle.Clickable
   .Sorting = SortOrder.Ascending
```

The following statements add three columns to the list view. The column headings are "File," "Version," and "Full Path," and the column widths are 100, 100, and 200. The text in all columns aligns with the left margin:

```
   .Columns.Add("File", 100, HorizontalAlignment.Left)
   .Columns.Add("Version", 100, HorizontalAlignment.Left)
   .Columns.Add("Full Path", 200, HorizontalAlignment.Left)
```

To add the item to the list view, use the *Add* method of the *Columns* property. The first parameter is the name of the file, the second is the index of an appropriate image, and the third is the array of subitems:

```
   .ListItems.Add(strFile, 0, strSubItems)
End With
```

The list view now shows the version information of the common control DLL. The following is the full code listing for the example:

```
Dim strFile As String
Dim strVersion As String
Dim strFullPath As String
Dim strSubItems(2) As String

'Set the file's name, version, and full path
strFile = "comctl32.dll"
strVersion = "5.81"
strFullPath = "C:\WINNT\system32\COMCTL32.DLL"

'Fill the subitem array
strSubItems(0) = strVersion
strSubItems(1) = strFullPath

With lvwVersionInformation
    'Set the view to Report view
    .View = View.report

    'Set the header style to clickable
    .HeaderStyle = ColumnHeaderStyle.Clickable

    'Display items in ascending order
    .Sorting = SortOrder.Ascending
    'Add File, Version, and Full Path columns
    .Columns.Add("File", 100, HorizontalAlignment.Left)
    .Columns.Add("Version", 100, HorizontalAlignment.Left)
    .Columns.Add("Full Path", 200, HorizontalAlignment.Left)

    'Add item
    .ListItems.Add(strFile, 0, strSubItems)
End With
```

ComboBox Control

The Windows Forms ComboBox control displays a list from which the user can select one or more choices. The ComboBox control appears as a text box and an associated list box. As text is typed into the text box, the list scrolls to the nearest match. In addition, when the user selects an item in the list box, it automatically uses that entry to replace the content of the text box and selects the text.

Because a combo box is similar to a list box, you may wonder when to use one or the other, but there are differences. Unlike a list box, a combo box allows the user to type an item that does not appear in the list. In general, use a combo box to present to the user a list of merely suggested choices, and use a list box to strictly limit the user's input to only the choices you present. In addition, as a combo box generally consumes less space on a form than a list box, a combo box may be a better choice when such space is at a premium.

The combo box has three different styles: simple, drop down, and drop-down list. In the simple style, the combo box has an edit box along with a list box. In the drop down style, the combo box looks like an edit box, but you can click it to see a drop down containing its items. The drop-down list style is similar to the drop down style. However, in the drop down list style, the user can only choose an item in the list. No item can be entered that does not appear in the list. Table 8.14 shows the properties of the ComboBox control.

Table 8.14 ComboBox Properties

Property	Description
AllowDrop	Determines if the combo box will receive drag-drop notifications
BackColor	The background color used to display text and graphics in the combo box

Continued

Table 8.14 Continued

Property	Description
Cursor	The cursor that appears when the mouse passes over the combo box
DataSource	Indicates the list that the combo box will use to get its items
DisplayMember	Indicates the property to display for the items in the combo box
Font	The font used to display text in the combo box
ForeColor	The foreground color used to display text and graphics in the combo box
IntegralHeight	Indicates whether the list portion can contain only complete items
ItemHeight	The height, in pixels, of items in an owner-drawn combo box
Items	The items in the combo box
MaxDropDownItems	The maximum number of entries to display in the drop-down list
MaxLength	Specifies the maximum number of characters that can be entered into the combo box
RightToLeft	Indicates whether the combo box should draw right-to-left for RTL languages
Sorted	Controls whether items in the list portion are sorted
Style	Controls the appearance and functionality of the combo box
TabInde☑	Determines the index in the tab order that the combo box will occupy
TabStop	Indicates whether the user can use the **Tab** key to give focus to the combo box
Text	The text contained in the combo box
ValueMember	Indicates the property to use as the actual value for the items in the combo box

You can add items to the combo box at design-time. To add items to a combo box at design-time:

1. Select the **ComboBox** control on the form.
2. If necessary, use the **View** menu to open the **Properties** window.
3. In the **Properties** window, click the **Items** property, then click the ellipsis.
4. In **String Collection Editor**, type the first item, then press **Enter**.
5. Type the next items, pressing **Enter** after each item.
6. Click **OK**.

There are a variety of ways to programmatically add items to a combo box. You can simply add an item to the list, letting the combo box control the position of insertion based on whether or not it is sorted. Alternatively, you can make explicit the point at which to insert an item. To simply add an item to a combo box, use the Add method of the Items collection:

```
'Add an item
cboUser.Items.Add("(New User)")
```

To add an item to a combo box, specifying the point of insertion, use the Insert method of the Items collection. This is useful when you want to specifically insert an item at a particular spot. This does not work when the *Sorted* property is set to True. As with other index properties, the point of insertion is zero-based; in the example that follows, the item is added at the first position in the list:

```
'Add an item at the first position
cboUser.Items.Insert(0, "(New User)")
```

Now, let's look at removing an item. You can remove an item by its index or by its value. For instance, to remove the first item in a list (by specifying its zero-based position), use the Remove method of the Items collection as follows:

```
'Remove the first item
cboUser.Items.Remove(0)
```

You can also remove an item by its value. Here is a snippet that specifies the value of the item to be removed:

```
'Remove the item "(New User)"
```

```
cboUser.Items.Remove("(New User)")
```

Sometimes you want to remove the selected item. You can use the *SelectedItem* and *Remove* methods for this. Let's look at how this is done in code:

```
'Remove the selected item
cboUser.Items.Remove(cboUser.SelectedItem)
```

DomainUpDown Control

The DomainUpDown control displays a list from which the user can select only one choice. You can use a combo box for everything you can use a domain up-down control for. However, a domain up-down control is generally used when the items have an inherent order, like days of the week, or months of the year.

The Domain UpDown control consists of an edit box and up-down buttons. The edit box displays the currently selected item. The user can select the next or previous item in the list by clicking the up-down buttons or pressing the **Up Arrow** and **Down Arrow** keys. Table 8.15 shows the properties of the DomainUpDown control.

Table 8.15 DomainUpDown Properties

Property	Description
AllowDrop	Determines if the up-down control will receive drag-drop notifications
BackColor	The background color used to display text and graphics in the up-down control
BorderStyle	Indicates the border style of the up-down control
Cursor	The cursor that appears when the mouse passes over the up-down control
Font	The font used to display text in the up-down control
ForeColor	The foreground color used to display text and graphics in the up-down control
InterceptArrowKeys	Indicates whether the up-down control will increment and decrement the value when the **Up Arrow** and **Down Arrow** keys are pressed
Items	The allowable values of the up-down control
ReadOnly	Indicates whether or not the up-down control is read-only

Continued

Table 8.15 Continued

Property	**Description**
RightToLeft	Indicates whether the up-down control should draw right-to-left for RTL languages
Sorted	Controls whether items in the domain list are sorted
TabIndex	Determines the index in the tab order that this control will occupy
TabStop	Indicates whether the user can use the **Tab**key to give focus to the up-down control
Text	The text contained in the up-down control
TextAlign	Indicates how the text should be aligned in the edit box
UpDownAlign	Indicates how the up-down control will position the up-down buttons relative to its edit box
Wrap	Indicates whether or not values wrap around at either end of the item list

The *Text* property holds the text contained in the up-down control. Another important property is the Items collection, which holds the items in the up-down control. You can use the *Items* property to programmatically add items to, and remove items from, the up-down control. We have added items to and removed items from many of the controls we discussed previously. However, you will often want to add items to an up-down control at design-time. To add items to an up-down control on a form at design-time:

1. Click the **DomainUpDown** control on the form.
2. In the **Properties** window, click the **Items** property, then click the ellipsis.
3. In the **String Collection Editor** window, type the first item, then press **Enter**.
4. Type the next items, pressing **Enter** after each item.
5. Click **OK**.

When the *ReadOnly* property is set to True, the *Items* property holds the only allowable values of the up-down control. In other words, the *Items* property is the domain of a read-only up-down control, which is where the control gets its name. The user cannot type a new value in a read-only up-down control. Instead, a keystroke selects the item in the control that starts with the letter pressed.

You can change the horizontal alignment of the up-down buttons. The default alignment of the up-down buttons is with the right margin of the control. To align the up-down buttons to the left of the edit box, set the *UpDownAlign* property as follows:

```
'Align the up-down buttons to the left of the edit box
dudDaysOfTheWeek.UpDownAlign = LeftRightAlignment.Left
```

The DomainUpDown control is similar to another control, the NumericUpDown control. In the next section, we will take a closer look at the numeric up-down control.

NumericUpDown Control

The NumericUpDown control allows the user to quickly change a numeric value by a chosen increment. The numeric up-down control shares many properties with the domain up-down control, but it is used to display numbers instead of text. The numeric up-down control is a great choice when collecting input such as a number of employees and a number of days from the user. Table 8.16 shows the properties of the NumericUpDown control.

Table 8.16 NumericUpDown Properties

Property	Description
AllowDrop	Determines if the up-down control will receive drag-drop notifications
BackColor	The background color used to display text and graphics in the up-down control
BorderStyle	Indicates the border style of the up-down control
Cursor	The cursor that appears when the mouse passes over the up-down control
DecimalPlaces	Indicates the number of decimal places to display
Font	The font used to display text in the up-down control
ForeColor	The foreground color used to display text and graphics in the up-down control
Hexadecimal	Indicates whether the up-down control should display its value in hexadecimal

Continued

Table 8.16 Continued

Property	Description
Increment	Indicates the amount to increment/decrement on each button click
InterceptArrowKeys	Indicates whether the up-down control will increment and decrement the value when the **Up Arrow** and **Down Arrow** keys are pressed
Maximum	Indicates the maximum value for the up-down control
Minimum	Indicates the minimum value for the up-down control
ReadOnly	Indicates whether or not the edit box is read-only
RightToLeft	Indicates whether the up-down control should draw right-to-left for RTL languages
TabIndex	Determines the index in the tab order that the up-down control will occupy
TabStop	Indicates whether the user can use the **Tab** key to give focus to the up-down control
TextAlign	Indicates how the text should be aligned in the edit box
ThousandsSeparator	Indicates whether thousands separators will be inserted between every three digits
UpDownAlign	Indicates how the up-down control will position the up-down buttons relative to its edit box
Value	The current value of the up-down control

The numeric up-down control has several unique properties. The *Value* property holds the current value of the numeric up-down control. The user can use the up-down buttons to increment or decrement the current value by the value of the *Increment* property.

The default value of the *Increment* property is 1. You can set the *Increment* property to a smaller value and display decimal places by increasing the value of the *DecimalPlaces* property. Along with displaying decimal places, you can also display thousands separators between every three digits.

The *Minimum* and *Maximum* properties indicate the minimum and maximum values of the up-down control. The value of the up-down control cannot be decremented past the value of the *Minimum* property. Also, the current value cannot be incremented past the value of the *Maximum* property.

PictureBox Control

The Windows Forms PictureBox control is used to display images in bitmap, GIF, icon, or JPEG formats. For example, you can use a picture box to display the logo of your company in the About box of your application. You can also change the image displayed in a picture box at runtime. Table 8.17 contains the properties of the PictureBox control.

Table 8.17 PictureBox Properties

Property	Description
BackColor	The background color used to display text and graphics in the picture box
BackgroundImage	The background image used for the control
BorderStyle	Controls what type of border the picture box should have
Cursor	The cursor that appears when the mouse passes over the picture box
Image	The image displayed in the picture box
SizeMode	Controls how the picture box will handle image placement and control sizing
TabIndex	Determines the index in the tab order that the picture box will occupy

You can programmatically change the image displayed in a picture box, which is particularly useful when you use a single form to display different pieces of information. For instance, you may choose to create one dialog box to display errors or status at different times during the run, and display different images for different messages. To set a picture at runtime, use the *FromFile* method of the *Image* class as follows:

```
'Display the company logo
Dim strImagePath As String = "Company Logo.jpg"
picCompanyLogo.Image = Image.FromFile(strImagePath)
```

You can also set a picture at design-time. To set a picture at design-time:

1. Select the **picture box** on the form.
2. If necessary, use the **View** menu to open the **Properties** window.
3. Select the **Image** property. Choose the **picture** in the **Open File** dialog box.

If you use a picture box to display images of different sizes, use the *SizeMode* property to size the picture box to prevent image cropping. You can set the *SizeMode* property to the following values:

- *PictureBoxSizeMode.AutoSize* to automatically resize the picture box
- *PictureBoxSizeMode.CenterImage* to center the image in the picture box
- *PictureBoxSizeMode.Normal* for not resizing either the image or the picture box
- *PictureBoxSizeMode.StretchImage* to stretch the image to the size of the fixed picture box

You can allow the picture box to automatically resize as follows:

```
'Allow the picture box to automatically resize
picCompanyLogo.SizeMode = PictureBoxSizeMode.AutoSize
```

At times you may want to clear the picture box and remove the image. To clear a picture box, simply set its *Image* property to Nothing. Here is what it would like in code:

```
'Clear the picture box
picCompanyLogo.Image = Nothing
```

TrackBar Control

The Windows Forms TrackBar control is a control similar to the ScrollBar that allows the user to scroll through a range of values. The track bar is great for adjusting volume, contrast, and brightness levels. It consists of a slider and tick marks. The user can adjust the value of the track bar by dragging the slider, using the arrow keys, or using the **Page Up** or **Page Down** keys. Table 8.18 shows the properties of the TrackBar control.

Table 8.18 TrackBar Properties

Property	Description
AllowDrop	Determines if the track bar will receive drag-drop notifications
AutoSize	Indicates whether the track bar will resize itself automatically based on a computation of the default scrollbar dimensions
BackColor	The background color used to display text and graphics in the track bar
Cursor	The cursor that appears when the mouse passes over the track bar
LargeChange	The number of positions the slider moves in response to mouse clicks or the **Page Up** and **Page Down** keys
Maximum	The maximum value for the position of the slider on the track bar
Minimum	The minimum value for the position of the slider on the track bar
Orientation	The orientation of the track bar
RightToLeft	Indicates whether the track bar should draw right-to-left for RTL languages
SmallChange	The number of positions the slider moves in response to keyboard input (arrow keys)
TabIndex	Determines the index in the tab order that the track bar will occupy
TabStop	Indicates whether the user can use the **Tab** key to give focus to the track bar
TickFrequency	The number of positions between tick marks
TickStyle	Indicates where the ticks appear on the track bar
Value	The position of the slider

The track bar shares most of its properties with other controls. You should be familiar with the *Value* property, which indicates the position of the slider. You should also know about the *SmallChange* and *LargeChange* properties. The *SmallChange* property determines the number of positions the slider moves to the right when the user uses the **Right Arrow** and **Down Arrow** keys, and to the left when the user uses the **Left Arrow** and **Up Arrow** keys. The *LargeChange* property determines the number of positions the slider moves in response to mouse clicks or when the **Page Up** and **Page Down** keys are pressed. The *LargeChange* property also determines the number of positions the slider moves when the user clicks to the left or right of the slider.

The *Maximum* and *Minimum* properties determine the maximum and minimum values for the position of the slider on the track bar. The *TickFrequency* property holds the number of positions between tick marks.

As for appearance, the *TickStyle* property indicates where the ticks appear on the track bar, which can be at the bottom right or top left of the slider, at both sides, or not at all. For example, to make the tick marks appear on both sides of the slider, use the *TickStyle* property as follows:

```
'Show tick marks at both sides of the slider
trkVolume.TickStyle = TickStyle.Both
```

Next, we will look at a powerful control: the DateTimePicker control.

DateTimePicker Control

The Windows Forms DateTimePicker control allows you to display and collect dates and times for the user. The date-time picker is a great replacement for a masked edit control with a date-time mask, because it allows you to display calendar information as you collect input from the user. For example, the date-time picker allows the user to view the days of the week around the day selected, or to view different months, as if flipping through a calendar. Another advantage of the date-time picker is that it disallows invalid input. While you can quickly set up a masked edit control to disallow alpha characters in a numeric field, it requires coding to disallow other invalid input such as 13 in a month field.

The date-time picker consists of a text box with an accompanying calendar drop down. The user can input a date in several ways. First, the user can enter a date by simply typing it into the text box. As discussed before, the date-time picker validates the entry and disallows it if it is invalid. Second, the user can use the drop down to navigate to a date. Third, the user can use the drop down and quickly click on Today's Date to enter the current date, regardless of the month displayed in the drop down.

The date-time picker can also show only time. The date-time picker shows only time when the *Format* property is set to Time. In that case, the date-time picker does not show a drop-down calendar, but you can still use it to collect only times that are valid.

The most important property of the date-time picker is the *Value* property, which holds the selected date and time. The *Value* property is set to the current date by default. To change the date before displaying the control, use the *Value* property as follows:

```
'Change the date to the following day
dtpEffectiveDate.Value = _
    DateAdd(Microsoft.VisualBasic.DateInterval.Day, 1, Date.Today)
```

Table 8.19 shows the other properties of the DateTimePicker control.

Table 8.19 DateTimePicker Properties

Property	Description
AllowDrop	Determines if the date-time picker will receive drag-drop notifications
CalendarFont	The font used to display the calendar
CalendarForeColor	The color used to display text within a month
CalendarMonthBackground	The background color displayed within the month
CalendarTitleBackColor	The background color displayed in the calendar's title
CalendarTitleForeColor	The color used to display text within the calendar's title
CalendarTrailingForeColor	The color used to display header day and trailing day text. Header and trailing days are the days from the previous and following months that appear on the current month calendar
Cursor	The cursor that appears when the mouse passes over the control
CustomFormat	The custom format string used to format the date or time displayed in the date-time picker
DropDownAlign	Controls whether the month drop down is aligned to the left or right of the date-time picker
Font	The font used to display text in the date-time picker
Format	Determines whether dates and times are dis played using standard or custom formatting
MaxDate	The maximum date selectable
MinDate	The minimum date selectable

Continued

Table 8.19 Continued

Property	Description
ReadOnly	Determines whether the user can free-form edit the date field. If this is set to False, then *onUserString* events will be fired
RightToLeft	Indicates whether the control should draw right-to-left for RTL languages
ShowCheckBox	Determines whether a check box is displayed in the date-time picker. When the check box is unchecked, no value is selected
ShowUpDown	Controls whether an up-down button is used to modify dates instead of a drop-down calendar
TabIndex	Determines the index in the tab order that this control will occupy
TabStop	Indicates whether the user can use the **Tab** key to give focus to the date-time picker
Value	The current date/time value for the date-time picker
ValueSet	Determines if the None check box is checked, indicating the user has selected a value

The *CheckBox* property is another key property of the date-time picker. Ordinarily, the *CheckBox* property is set to False and the date-time picker always holds a value. There may be times, however, when you want to allow the user not to specify a value—for example, if you are using a date-time picker to collect an employee's date of termination, but the employee is still active. In such a case, set the *CheckBox* property to True as follows:

```
dtpDateOfTermination.CheckBox = True
```

When the *CheckBox* property is set to True, the edit portion of the date-time picker displays a check box. Your user can now uncheck the box to indicate that there has not been a date of termination. When an employee is terminated, the user can check the box, and select the appropriate date. It is important to note that when the check box is unchecked, the date-time picker *Value* property returns Null.

You should be familiar with another property of the date-time picker, the *Format* property. The *Format* property allows you to use standard formatting strings or custom formats to display the date. To display a custom format, use the *Format* property along with the *CustomFormat* property as follows:

```
'Display the day of week
dtpDayOfWeek.Format = _
    System.WinForms.DateTimePickerFormats.Custom
dtpDayOfWeek.CustomFormat = "dddd"
```

You should also be familiar with the *MinDate* and *MaxDate* properties. The *MinDate* property is the minimum date selectable and the MaxDate is the maximum date selectable. The user cannot choose a date before the minimum date. This is useful when you need to compare dates. For example, you would not want the user to enter a date of termination that occurs before the date of hire.

Panel Control

The Windows Forms Panel control is used to group other controls. A panel allows you to give the user a logical visual cue of controls that belong together. For example, on a dialog box that displays properties of a file, you could place all check boxes describing the attributes of the file—*Archive*, *Normal*, *System*, *Hidden*, and *ReadOnly*—on one panel to help the user identify them as one group. In addition, a panel is also useful at design-time, as you can move all controls on a panel simultaneously by moving only the panel. Table 8.20 shows the properties of the Panel control.

Table 8.20 Panel Properties

Property	Description
AllowDrop	Determines if the panel will receive drag-drop notifications
AutoScroll	Determines whether scroll bars will automatically appear if controls are placed outside the form's client area
AutoScrollMargin	The margin around controls during autoscrolls
AutoScrollMinSize	The minimum logical size for the autoscroll region
BackColor	The background color used to display text and graphics in the panel
BackgroundImage	The background image used for the panel
BorderStyle	Indicates whether or not the panel should have a border
Cursor	The cursor that appears when the mouse passes over the panel

Continued

Table 8.20 Continued

Property	Description
DockPadding	Determines the size of the border for docked controls
DrawGrid	Indicates whether or not to draw the positioning grid
Font	The font used to display text in the panel
ForeColor	The foreground color used to display text and graphics in the panel
GridSize	Determines the size of the positioning grid
RightToLeft	Indicates whether the panel should draw right-to-left for RTL languages
SnapToGrid	Determines if controls should snap to the positioning grid
TabIndex	Determines the index in the order that the panel will occupy
TabStop	Indicates whether the user can use the **Tab** key to give focus to the panel

Let's look at how you can add check boxes to a panel in code. The following snippet adds check boxes describing file attributes to a panel:

```
'Add file attribute checkboxes to a panel
With pnlAttributes.Controls
    .Add(chkArchive)
    .Add(chkNormal)
    .Add(chkSystem)
    .Add(chkHidden)
    .Add(chkReadOnly)
End With
```

The *BorderStyle* property indicates whether a panel should have a border. You can add a 3D border or a flat border to a panel. Your panel can also have no border. Let's look at the code for these options:

```
pnlAttributes.BorderStyle = WinForms.BorderStyle.Fixed3D
pnlAttributes.BorderStyle = WinForms.BorderStyle.FixedSingle
pnlAttributes.BorderStyle = WinForms.BorderStyle.None
```

You can change the background color used to display graphics and text in a panel. When you change a panel's background color, the background color of all the controls it contains also changes to the color you set.

```
'Change background color to gray
panel1.BackColor = Color.Gray
```

Your panel can also display a background image, which appears behind the controls contained within the panel. To display a background image, set the *BackgroundImage* property as follows:

```
pnlAttributes.BackgroundImage = Image.FromFile("Background.jpg")
```

GroupBox Control

Like the Panel Control, the GroupBox control is used to group other controls. In contrast to the Panel control, the GroupBox cannot have scrollbars and only the GroupBox allows you to display a caption with a group of controls. This is similar to the Frame control in previous versions of Visual Basic. Nonetheless, you can use the *Add* method we discussed in the previous section to create a group of controls. The group box also allows you to set the background programmatically using the *BackColor* and *BackgroundImage* properties discussed in the previous section. Table 8.21 shows the other properties of the GroupBox control.

Table 8.21 GroupBox Properties

Property	Description
BackColor	The background color used to display text and graphics in the group box
BackgroundImage	The background image used for the control
DrawGrid	Indicates whether or not to draw the positioning grid
Font	The font used to display text in the group box
ForeColor	The foreground color used to display text and graphics in the group box
GridSize	Determines the size of the positioning grid
RightToLeft	Indicates whether the group box should draw right-to-left for RTL languages
SnapToGrid	Determines if controls should snap to the positioning grid

Continued

Table 8.21 Continued

Property	Description
TabIndex	Determines the index in the tab order that this control will occupy
Text	The text contained in the group box

TabControl Control

The Windows Forms TabControl control is used to hold controls separated by tabs. The tab control is handy when you need to display different groups of information on limited real estate. You can put each group of information on a separate tab of the tab control. For example, in a payroll application you can put an employee's general information, such as her name and address, on one tab and information about her dependents on another tab.

The *TabPages* property is an important property of the tab control, because it controls the collection of tab pages. Often, you will want to add tab pages to a tab control at design-time so you can add controls to each page. To add a tab page to a tab control:

1. Select the **tab control** on the form.
2. If necessary, use the **View** menu to open the **Properties** window.
3. Click the **TabPages** property, then click the ellipsis.
4. In **Tab Page Collection Editor**, click **Add**.
5. In the **TabPage1 Properties** box, change the **Text property** to an appropriate caption for the tab page.
6. Click **OK**.

The tab control has other unique properties. Table 8.22 lists the properties of the TabControl control.

Table 8.22 TabControl Properties

Property	Description
Alignment	Determines whether the tabs appear on the top, bottom, left, or right side of the tab control (left or right are implicitly multilined)

Continued

Table 8.22 Continued

Property	Description
AllowDrop	Determines if the tab control will receive drag-drop notifications
Appearance	Indicates whether the tabs are painted as buttons or regular tabs
Cursor	The cursor that appears when the mouse passes over the tab control
DrawGrid	Indicates whether or not to draw the positioning grid
DrawMode	Indicates whether the user or the system paints the captions
Font	The font used to display text in the tab control
GridSize	Determines the size of the positioning grid
HotTrack	Indicates whether the tabs visually change when the mouse passes over them
ImageList	The image list from which the tab control takes its images
ItemSize	Determines the width of fixed-width or owner-draw tabs and the height of all tabs
Locked	Determines if the user can move or resize the control
MultiLine	Indicates if more than one row of tabs is allowed
Padding	Indicates how much extra space should be added around the text/image in the tab, if the *DrawMode* property value is Fixed
RightToLeft	Indicates whether the tab control should draw right-to-left for RTL languages
ShowToolTips	Indicates whether tooltips should be shown for tables that have their tooltips set
SizeMode	Indicates how tabs are sized
SnapToGrid	Determines if controls should snap to the positioning grid
TabInde☑	Determines the index in the tab order that the tab control will occupy
TabPages	The tab pages in the tab control
TabStop	Indicates whether the user can use the **Tab** key to give focus to the tab control

Another property you should be familiar with is the *MultiLine* property. The *MultiLine* property determines if more than one row of tabs is allowed. This property is useful when you have a large number of tabs or long captions. You can ensure the captions are visible by setting the *MultiLine* property to True.

The *HotTrack* property is an eye-catching, commonly used property of the tab control. When the *HotTrack* property is set to True, the caption of the tabs change colors when the mouse passes over them. The tabs change to their regular color when the mouse is not over them.

We have now discussed the powerful controls built into Visual Basic .NET. However, at times these controls will not quite provide the functionality you want. When that happens, you will want to create your own controls. Next, we will look at creating custom Windows components and controls.

Creating Custom Windows Components

The Windows Forms framework offers numerous components that you can use to build applications. In advanced applications these components may not provide the functionality you want. Should this be the case, you can create your own components to provide exactly what you need. But what exactly are components?

A component is a class with emphasis on cleanup and containment. Components provide reusable code in the form of objects. You can think of a component as a control without user interface capabilities. Therefore, it makes sense to discuss how to author a component before we look at authoring controls. The two are very similar, and you can use everything you know about authoring a component when you author a control.

A good way to understand how to author a component is by walking through the process. In the following exercise you will create a component.

Exercise 8.1: Creating a Custom Windows Component

In this exercise, you will create a class library project for a component CFirst, add constructors and destructors, add a property, and test the component. It is useful to have the Toolbox, Solution Explorer, Properties, and Task List windows open when you start creating your component:

1. Use the **View** menu to open the **Toolbox**, **Solution Explorer**, and the **Properties** window.

2. From the **View** menu, point to **Other Windows** and then select **Task List** to open the Task List.

Creating a Class Library Project

To create a component, you have to create a class library. Visual Basic .NET provides you with a class library project template to help you create the class library. To create the *CFirstLib* class library and the CFirst component:

1. On the **File** menu, select **New** and then **Project** to open the **New Project** dialog box. From the list of **Visual Basic Projects**, select the **Class Library** project template, and enter **CFirstLib** in the Name box.

NOTE

When you create a new project, always specify its name to set the root namespace, assembly name, and project name. Doing so also ensures that the default component will be in the correct namespace.

2. In **Solution Explorer**, right-click **Class1.vb** and select **View Code** from the shortcut menu.
3. In the **Code** window, locate the **Class** statement, **Public Class Class1**, and change the name of the component from **Class1** to **CFirst**.

NOTE

As shown by the Inherits statement immediately below the Class statement, a component inherits from the *Component* class by default. The *Component* class provides the ability to use designers with your component.

4. In **Solution Explorer**, click **Class1.vb**, and in the **Properties** window change the filename to **CFirst.vb**.
5. On the **File** menu, select **Save CFirst.vb** to save the project.

Adding Constructors and Destructors

We now add constructors and destructors to control the way the CFirst component is initialized and torn down. The only function of our component will be to maintain a running count of how many *CFirst* objects are in memory at any given time. When a *CFirst* object is initialized, the running count will be incremented, and when it a *CFirst* object is torn down, the running count will be decremented. To add code for the constructor and destructor of the *CFirst* class:

1. In the **Code Editor** window (either before or after the declaration of the components container) add member variables to keep a running total of instances of the *CFirst* class, and an ID number for each instance:

   ```
   Public ReadOnly intInstanceID As Integer
   Private Shared intNextInstanceID As Integer = 0
   Private Shared intClassInstanceCount As Integer = 0
   ```

   Shared member variables such as *intClassInstanceCount* and *intNextInstanceID* are initialized the first time the *CFirst* class is referred to in code and exist at the class level only. All instances of CFirst that access these members will use the same memory locations. Read-only members such as intInstanceID can be set only in the constructor:

2. Locate ***Public Sub New***, the default constructor for the *CFirst* class. After the call to **InitializeComponent**, add the following code to set the instance ID number and to increment the instance count when a new CFirst object is created:

   ```
   intInstanceID = intNextInstanceID
   intNextInstanceID += 1
   intClassInstanceCount += 1
   ```

3. Add the following method after the end of the constructor to decrement the instance count just before the *CFirst* object is removed from memory:

   ```
   Protected Overrides Sub Finalize()
       intClassInstanceCount -= 1
   End Sub
   ```

Adding a Property to the Class

Now we will add a single property to our *CFirst* class. The shared property *intClassInstanceCount* holds the number of CFirst objects in memory.

To create a property for the *CFirst* class, add the following property declaration to the CFirst class:

```
Public Shared ReadOnly Property InstanceCount() As Integer
    Get
        Return intClassInstanceCount
    End Get
End Property
```

Testing the Component

To test our CFirst component, we need to create a client project that uses the component. In order to do this, we need to set the client project as the startup project to ensure it will be the first to run when our solution is started. We will call the client project CFirstTest. To add the CFirstTest client project as the startup project for the solution:

1. On the **File** menu, select **Add Project** and then **New Project** to open the **Add New Project** dialog box.
2. From the list of **Visual Basic Projects**, select the **Windows Application** project template. In the **Name** box, type **CFirstTest**, then click **OK**.
3. In **Solution Explorer**, right-click **CFirstTest** and click **Set As Startup Project** on the shortcut menu.

In order to use the fully qualified name of the CFirst component (CFirstLib.CFirst), we need to add a reference to the class library project. To add a reference to the class library project:

1. In **Solution Explorer**, right-click the **References** node immediately beneath CFirstTest, and select **Add Reference** from the shortcut menu.
2. In the **Add Reference** dialog box, select the **Projects** tab.
3. Double-click the **CFirstLib** class library project to add it to the Selected Components list. **CFirstLib** will appear under the References node for the *CFirstTest* project. Click **OK**.

4. In **Solution Explorer**, right-click **Form1.vb** and select **View Code** from the shortcut menu.

Adding an Imports statement allows us to refer to the component type as CFirst, omitting the library name. This makes it easier to use the component.

To add an Imports statement, add the following Imports statement to the list of **Imports** statements at the top of the **Code Editor** window for **Form1**:

```
Imports CFirstLib
```

Using the Component

You can now use the component's properties in code. We need to add a button to our form to create new instances of CFirst. To use the CFirst component programmatically:

1. From the **Toolbox** window, select the **Win Forms** tab, and double-click the **Button** control.
2. On **Form1**, double-click the **Button** control. In the **Click** event handler for **Button1**, add the following code:

```
Dim objCFirst As New CFirst()


MessageBox.Show("There are " & objCFirst.InstanceCount _
    & " CFirst objects in memory.")
```

As you type the code in the Code window, the Complete Word box will appear displaying the *InstanceCount* property. You can expose methods similarly.

Creating Custom Windows Controls

As we discussed, controls are components with user interface capabilities. There is a lot of overlap between component creation and control creation. Instead of discussing the similarities, let's look at a common problem. You have a certain built-in control available to you and you are happy with it, except for this one nagging feature. The control does not do exactly what you want, or it lacks one feature you need. You will find that often you do not need a whole new control; you just need to be able to change an existing one.

This can be done by inheriting from a control. For a simple example, let's look at a Button control. You can create your own control to be placed in the

Toolbox window. To keep it simple, let's say you want to create a custom control that inherits from the Button control but has a different background color. This is a small change, but once we discuss how to make this small change, you can change other controls to do more elegant things.

Exercise 8.2: Creating a Custom Windows Control

In this exercise, you will create a custom control that inherits from the Windows Forms Button. Our custom control will resemble the Windows Forms Button control, but our custom control will have a white background. It is useful to have the **Toolbox**, **Solution Explorer**, **Properties**, and **Task List** windows open when you start creating your component:

1. Use the **View** menu to open the **Toolbox**, **Solution Explorer**, and the **Properties** window.
2. From the **View** menu, point to **Other Windows** and then select **Task List** to open the Task List.

Creating a Control Library Project

As with components, we need to create a control library. Visual Basic .NET provides you with a control library project template to help you create the control library. To create the *CMyButtonLib* class library and the CMyButton component:

1. On the **File** menu, select **New** and then **Project** to open the **New Project** dialog box. From the list of **Visual Basic Projects**, select the **Windows Control Library** project template, and enter **CMyButtonLib** in the **Name** box.
2. In **Solution Explorer**, right-click **Control1.vb** and select **View Code** from the shortcut menu.
3. In the **Code** window, locate the Class statement, **Public Class Control1**, and change the name of the component from Control1 to **CMyButton**.
4. In **Solution Explorer**, click **Control1.vb** and in the **Properties** window change the filename to **CMyButton.vb**.
5. On the **File** menu, select **Save CMyButton.vb** to save the project.

Adding Constructors and Destructors

We now add constructors and destructors to control the way the CMyButton control is initialized and torn down. Our component will resemble a button control, but our component's background color will be white.

To add code for the constructor and destructor of the CMyButton control, locate **Public Sub New**, the default constructor for the *CMyButton* class. After the call to **InitializeComponent**, add the following code to set the background color to white:

```
Me.BackColor = System.Drawing.Color.White
```

Testing the Component

To test our CMyButton control, we need to create a client project that uses the control:

1. On the **File** menu, select **Add Project** and then **New Project** to open the **Add New Project** dialog box.
2. From the list of **Visual Basic Projects**, select the **Windows Application** project template. In the **Name** box, type **CmyButtonTest**, then click **OK**.
3. In **Solution Explorer**, right-click **CMyButtonTest** and click **Set As Startup Project** on the shortcut menu.

In order to use the fully qualified name of the CButton control (CMyButtonLib.CMyButton), we need to add a reference to the control library project. To add a reference to the control library project:

1. In **Solution Explorer**, right-click the **References** node immediately beneath CMyButtonTest, and select **Add Reference** from the shortcut menu.
2. In the **Add Reference** dialog box, select the **Projects** tab.
3. Double-click the **CMyButton** class library project to add it to the **Selected Components** list. **CMyButton** will appear under the References node for the *CMyButtonTest* project. Click **OK**.
4. In **Solution Explorer**, right-click **Form1.vb** and select **View Code** from the shortcut menu.

Adding an Imports statement allows us to refer to the component type as CFirst, omitting the library name. This makes it easier to use the component.

To add an Imports statement, add the following Imports statement to the list of **Imports** statements at the top of the **Code Editor** window for **Form1**:

```
Imports CMyButtonLib
```

Using the Component

You can now use the component's properties in code. We need to add a button to our form to create new instances of CFirst.

To use the CFirst component programmatically,, double-click **Form1**. In the **Click** event handler, add the following code:

```
Dim btnMyButton As New CMyButton()
btnMyButton.Show()
```

As you type the code in the Code window, the Complete Word box will appear displaying the properties and methods of the Button control.

Summary

We have seen that the Window Forms framework offers many controls that you can use. These controls have properties, methods, and events to help you accomplish your goals. As we have discussed, many properties of controls can be changed at design-time and at runtime. We have seen many examples of just how to change those properties at runtime.

Sometimes you need a reusable object for a certain functionality that is not built into the Windows Forms framework. For these, applications components are just the ticket. Not only do components provide a great way to reuse code, but they also allow you to get the exact functionality you need.

Controls are components with user interface capabilities. You can create properties for controls the same way you saw properties for components created in this chapter. More often than not, you do not need a whole new control, but simply a change to one of the many powerful controls built into the Windows Forms framework. You have seen how to make such a change and get a control to do exactly what you want.

Solutions Fast Track

Built-In Controls

- ☑ The Windows Forms framework offers many controls that you can use to build applications.
- ☑ The built-in controls have many properties, methods, and events in common.
- ☑ You can change many of the properties of built-in controls both at design-time and runtime.

Creating Custom Windows Components

- ☑ At times, you need a custom component to provide exactly the functionality you need.
- ☑ Components provide reusable code in the form of objects.
- ☑ Creating a custom component is a delicate multistep process.

Creating Custom Windows Controls

☑ Sometimes built-in controls do not provide the functionality you need.

☑ A control is a component with user interface capabilities.

☑ Often, you can extend an existing control instead of creating a new one.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: Does Visual Basic .NET support component and control creation?

A: Yes, Visual Basic .NET allows you to create your own components and controls. Components can be used to provide reusable code in the form of objects.

Q: Can I choose more than one control to collect a piece of information from the user?

A: Yes, many controls allow you to collect input from the user. However, some controls are better than others for collecting and validating specific types of information, like dates and times.

Q: Can the properties of controls be changed at runtime?

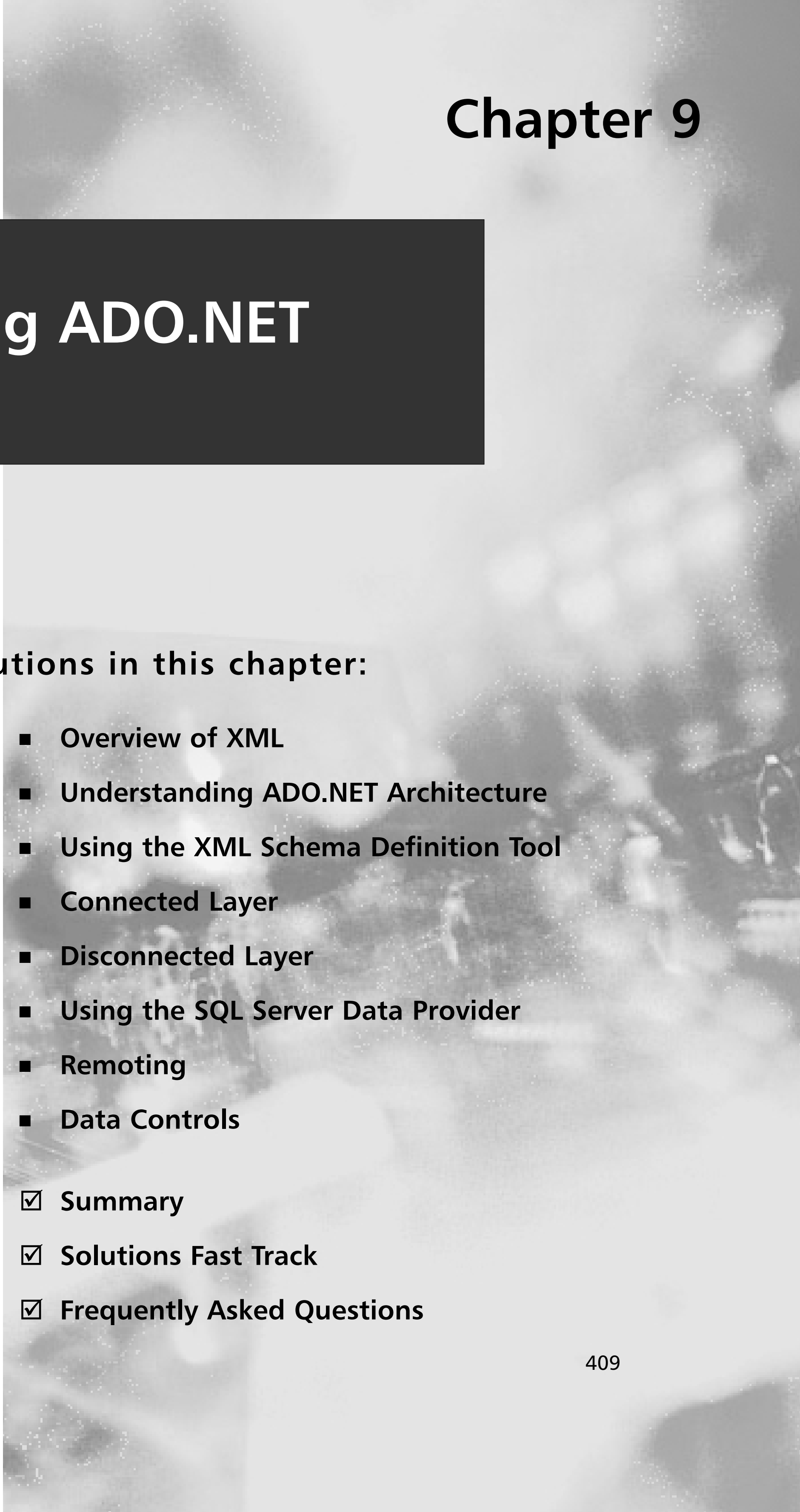
A: Many properties of controls can be changed at runtime. However, other properties are read-only at runtime.

Q: How do I programmatically add items to a combo box?

A: You can programmatically add items to a combo box and other controls by using the *Add* method of the Items collection.

Q: Should I create my own controls?

A: You only need to create your own controls when the built-in controls do not provide the functionality you need, or if they cannot be extended to provide that functionality.



Chapter 9

Using ADO.NET

Solutions in this chapter:

- Overview of XML
- Understanding ADO.NET Architecture
- Using the XML Schema Definition Tool
- Connected Layer
- Disconnected Layer
- Using the SQL Server Data Provider
- Remoting
- Data Controls

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

The .NET Framework leverages XML heavily. XML is steadily growing and has gone beyond being the latest buzzword. It allows for interoperability of passing data between disparate systems on different platforms and Microsoft has created a number of enterprise servers that support XML, such as their Biztalk server and even SQL Server 2000 has XML functionality built in. As industries accept standards based on XML, its use will spread to all phases of enterprise solutions. XML is a self-describing data format and many tools are already available to automatically create the schemas and documents. In this chapter, we discuss one of these tools, the XML Schema Definition tool.

The ADO.NET architecture uses XML as its native data format. ADO.NET is different from ADO 2.x. ADO.NET is filled with XML functionality, including XML Document objects. Here we discuss working with data when connected to the data source and when disconnected. You should already be familiar with working with disconnected data from previous versions of ADO. This functionality is extended using XML and will allow for better use of data in Web pages without a constant connection to a remote database server. ADO.NET has the concept of a *data* provider, which is similar to an OLE DB provider in ADO. ADO.NET currently has a built-in data provider for SQL Server. You gain significant performance benefits from a built-in provider.

We also discuss *remoting* in .NET. Remoting allows objects or components to communicate across networks or the Internet and takes care of many of the complexities of communicating across networks. This allows business objects to reside on any computer across your network or even the Internet. It also allows the integration of third-party business objects into your applications by calling their methods across the Internet. Imagine that you are creating a Web application that will accept credit cards online. With this technology, you can just make calls to their objects across the Internet and not have to concern yourself with supporting and configuring their product. Finally, in this chapter, we discuss the most common data controls: the DataGrid, DataList, and Repeater. These controls bind directly to a data source, display larger amounts of data in an easy-to-read format, and are easy to configure and control. This is an important chapter. XML and ADO.NET will be the core of most applications, and it is necessary to understand how to handle data in the .NET Framework.

Overview of XML

XML stands for *Extensible Markup Language*. XML is a standard for formatting data that is extensible, self-describing, and solves many of the problems inherent with data manipulation and transportation. XML uses a document paradigm whereby a document contains a few basic elements that are easy to understand and grasp. Moving data using XML is similar to moving data in ASCII format; however, we have more options. For example, we can represent relationships in XML that were very difficult to do in flat ASCII files. We do not have to deal with fixed width, delimiting characters, or even column order.

XML is a standard with broad support in many industries and has grown to encompass almost any type of data transportation. Since it is a specification based on industry standards, we can count on wide-scale use and acceptance as well as a broad range of tools to get the most out of our data. Keep in mind that XML documents are not to be read directly, but are to be transported or transformed into another format such as HTML.

XML Documents

XML documents are the heart of the XML standard. An XML document has at least one element that is delimited with one start tag and one end tag. XML documents are similar to HTML, except that the tags are made up by the author. They may have attributes designated by name value pairs to further allow for data description:

```
<?xml version="1.0" standalone="yes"?>
<NewXMLDocument>
  <MyTable>
    <Column1>Data1</Column1>
    <Column2>Data2</Column2>
  </MyData>
</NewXMLDocument>
```

XSL

XSL stands for *Extensible Stylesheet Language*. XSL documents are XML documents that, when combined with another XML document, transform data into something more useful, such as highlighting specific data. This allows a developer to separate the logic of data presentation and the actual data into two independent objects. XSL documents use special format objects to control the transformation of data from one format (XML) to another (HTML).

XDR

XDR stands for *XML Data Reduced*, an XML-based schema format that is more powerful than Data Type Definition, or DTD. For our purposes, XDR is synonymous with XML Schema Definition (XSD). These formats allow verification of data types and much more during parsing and will allow XML data to be applied in many forms that DTD could not address. This allows our XML documents to describe their data, and allows us to constrain them to particular schemas or structure.

ADO.NET uses XML schema files to implement a Typed DataSet. A Typed DataSet is a wizard-generated set of controls that make use of a schema definition to enforce DataType information and column names; and enforce these at compile time.

XPath

XPath describes location paths of data between XML documents, and can be used to query other XML documents. Using a syntax that is similar to navigating a file system, we can select parts of a document. This may be useful for developers familiar with creating XPath queries. XML Query promises to combine queries to relational databases and XML documents with similar syntax and dialog. This will give us a powerful tool that is independent of the data source, as long as the data source can serve up XML.

We can use XPath queries in DataXmlNavigator and XMLNavigator objects to return nodes for reading or updating. It is a tool useful for eliminating the scrolling and manual search implementation for addressing data in XML documents. We can take advantage of XPath, but it is not necessary to understand this language to make the most of XML in ADO.NET.

Understanding ADO.NET Architecture

ADO.NET is the latest extension of the Universal Data Access technology. Its architecture is similar to ADO in some respects, but a great departure in others. ADO.NET is much simpler, less dependant on the data source, more flexible, and the format of data is textual instead of binary. Textual formatted data is more verbose than binary formatted data, which makes it comparably larger. What we get for this is ease of transportation through disconnected networks, flexibility, and speed. Because ADO.NET is based on XML, it only requires that a data provider serve up the data in XML. Once you write your data access code, you only need to change a few parameters to connect to a different data source.

ADO.NET is based on a connectionless principle that is designed to ease the connection limitations that we traditionally had to deal with when creating distributed solutions. We no longer have to maintain a connection, or even worry about many of the connection options that plagued us in the past. Since the ADO.NET classes inherit from the same core of data access classes, it will make switching data sources much easier and less troublesome. The core ADO.NET namespaces are described in Table 9.1.

Table 9.1 Core ADO.NET Namespaces

Namespace	Description
System.Data	Makes up the core objects such as DataTable, DataColumn, DataView, and Constraints, to name a few. This namespace forms the basis for the others.
System.Data.Common	Defines generic objects shared by different data providers, such as DataAdapter, DataColumnMapping, and DataTableMapping. This namespace is used by data providers and contains collections useful for accessing data sources. For the most part, we do not use this namespace unless we are creating our own data provider.
System.Data.OleDb	Defines objects that we use to connect to and modify data in various data sources. It is written as the generic data provider and the implementation provided by the .NET Framework in Beta2 contained drivers for Microsoft SQL Server, the Microsoft OLE DB Provider for Oracle, and Microsoft Provider for Jet 4.0. This class is useful if your project connects to many different data sources, but you want more performance than the ODBC provider.
System.Data.SqlClient	A data provider namespace created specifically for Microsoft SQL Server version 7.0 and later. When using Microsoft SQL Server, this namespace is written to take advantage of the Microsoft SQL Server API directly and provides better performance than the more generic System.Data.OleDb namespace.

Continued

Table 9.1 Continued

Namespace	Description
System.Data.SQLTypes	Provides classes for data types specific to Microsoft SQL Server. These classes are designed specifically for SQL Server and provide better performance. If we do not use these specifically, the SQLClient objects will do it for us, but may result in loss of precision or type conversion errors.
System.Data.ODBC	This namespace is intended to work with all compliant ODBC drivers, and is available as a separate download from Microsoft.

The Command, Connection, DataReader, and DataAdapter are the core objects in ADO.NET. They form the basis for all operations regarding data in .NET. These objects are created from the *System.Data.OleDb*, *System.Data.SqlClient*, and the *System.Data*.ODBC namespaces.

Differences between ADO and ADO.NET

ADO is based on a binary format for data, with a database connection paradigm. We were allowed to disconnect record sets in later versions of ADO, but this required marshaling the binary representation of the data from process to process and then reconstructing it into valid data formats. This limited us to tightly coupled implementations with connection issues if we needed to operate through firewalls, proxies, and so forth. Since ADO.NET is based on a textual standard that does not require type conversion, marshaling, and special RPC ports for transporting the data, Microsoft has removed many of the obstacles encountered when creating distributed applications .

ADO uses a Recordset object that represents a table of data; however, this view was not ideal for representing the often-relational data of today's applications. In addition to this limitation, it has several options that confused many developers learning the technology, such as cursor types, cursor location, and lock types. These options generated a lot of confusion, especially if the user was not familiar with the database from which the data was coming.

XML Support

XML is the native data format for ADO.NET. XML documents replace the binary elements that were an integral part of ADO. XML is a standard with broad support and is ideal for the type of disconnected, distributed, Internet applications being developed today. By leveraging XML, it is easier for developers of data providers to integrate with ADO.NET. A data provider only has to create XML documents to work with ADO.NET, which makes it a viable alternative for those seeking the maximum flexibility for cross database development.

ADO.NET Configuration

Configuration for ADO.NET is handled during the .NET Framework installation. We don't need to set any registry settings or path statements to alter. It all happens on the fly, and all we have to do is understand it and use it. With ADO we had to contend with MDAC updates until Windows 2000 came along and incorporated this into the architecture. In some ways, we are back to the MDAC days, except for .NET's strategy to do away with dll. This also allows us to run multiple versions of data access technologies without having to worry about ramifications or breaking old code. Yes, even ADO.NET gets its roots from the Common Language Runtime, along with all the benefits.

Remoting in ADO.NET

XML is key to *remoting* in ADO.NET. XML and HTML together can be used to create services based on the Simple Object Access Protocol, or SOAP. This allows us to create applications that operate similar to the Browser paradigm that has made the Internet so diverse. By offering a service—for example, credit card authorization—we can create a listener that understands a standard SOAP request and after verifying a valid request can return a proper response. All this is possible with XML. XML is at the center of ADO.NET, and therefore very important to any .NET application that will involve data of any type.

Maintaining State

Since ADO.NET uses a connectionless, XML document methodology to handle data access, *maintaining state* is not optional when using the DataSet object. When we populate the DataSet, we are retrieving a populated XML document that is a copy of the data stored in the original data source. This mandates that we have "state," and comes in handy when we have lengthy operations that may tie up a connection. It also allows us to easily enhance our applications with a data cache of often queried but seldom modified data. We can create a DataSet at application startup and populate it with DataTables that we can use to populate drop-down boxes and list boxes in our application without having to access the database each time the list needs populating. Add some code to refresh the cache after an update, and we have an in-memory database that we can use to ease the burden on the database server. This is just one example of a use for maintaining state in an application. ADO.NET makes this type of development exceptionally easy.

Using the XML Schema Definition Tool

We can use the *XSD Schema Definition tool* to create schema files used for XML data validation. In the Beta2 version, it is command line only. Entering **XSD.EXE /?** at the command prompt brings up a long list of options for creating XML schemas. To get a simple schema definition, consider the following MyData.xml example (see CD file Chapter 09/Chapter9 Beta2/Samples/XML/MyData.xml):

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
  <Shippers>
    <ShipperID>1</ShipperID>
    <CompanyName>Speedy Express</CompanyName>
    <Phone>(503) 555-9831</Phone>
  </Shippers>
  <Shippers>
    <ShipperID>2</ShipperID>
    <CompanyName>United Package</CompanyName>
    <Phone>(503) 555-3199</Phone>
  </Shippers>
  <Shippers>
    <ShipperID>3</ShipperID>
    <CompanyName>Federal Shipping</CompanyName>
    <Phone>(503) 555-9931</Phone>
  </Shippers>
</NewDataSet>
```

Running the file through XSD.EXE results in this (see CD file Chapter 09/Chapter9 Beta2/Samples/XML/MyData.xsd):

```
C:>xsd c:\MyData.xml


MyData.xsd
```

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema id="NewDataSet" targetNamespace="" xmlns=""
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xsd:element name="NewDataSet" msdata:IsDataSet="true">
    <xsd:complexType>
      <xsd:choice maxOccurs="unbounded">
        <xsd:element name="Shippers">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="ShipperID" type="xsd:string" _
    minOccurs="0" />
              <xsd:element name="CompanyName" type="xsd:string" _
    minOccurs="0" />
              <xsd:element name="Phone" type="xsd:string"
    minOccurs="0" />
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

To specify our new XSD as the schema of choice for our fresh new XML Document, we simply add this attribute to our original root tag:

```
<NewDataSet xmlns="x-schema:MyData.xsd">
```

ADO.NET uses the xsd file to implement the Strongly Typed DataSet later in the chapter. Visual Studio.NET uses the xsd.exe to create this file and then adds it to the assembly.

Connected Layer

ADO.NET does not support connected data operations. You can, however, use the older ADO and OLE DB libraries by converting them to .NET. This process involves creating a .NET wrapper around the ADO libraries. Using this wrapper and the COM Interop module, we can call any COM+ component as if it were a native .NET library. This module is a proxy that allows us to make use of the existing code base without having to modify any of the COM components. Therefore, we can easily make use of the ADO functionality we have grown to love (and sometimes hate).

To make this happen, we will use the TLBIMP, a utility that comes with the .NET Framework. Using the Type library from the COM component to be imported, TLBIMP creates a .NET assembly containing the necessary .NET metadata. To import the ADO libraries into .NET, all we need to do is execute the following command at the command prompt:

```
tlbimp.exe <pathto>\msado15.dll
```

This creates a file ADODB.dll in the current directory, unless you specify a different path or name using the /out: switch. Usually, you don't have to do any of this if you are using Visual Studio.NET. Set a reference to the ADO library in the registered COM Components list to set this up in Visual Studio.NET.

To summarize, ADO.NET does not support connected database operations. If you need this functionality, you are going to have to revert to the ADO and OLE DB libraries using the COM Interop module.

Data Providers

Data providers are written specifically for a database. *System.Data.SqlClient* is an example of a data provider written solely for access to Microsoft SQL Server. This provider takes advantage of SQL Server in ways that are unique to Microsoft SQL Server. It is written to use the SQL Server API directly and is more efficient than using the more generic OLE DB, or ODBC providers. You should not use this provider to connect to Oracle, Access, or any other database. It is, however, inheriting from the same base classes that System.Data.OleDb is. Therefore, if we write code for the SQL data provider, we can just change our connection from SqlConnection to an OleDbConnection, change the connection string, and the rest of the code stays the same. This is one benefit of loose coupling; for example, we can now easily change the data provider from Access to SQL Server.

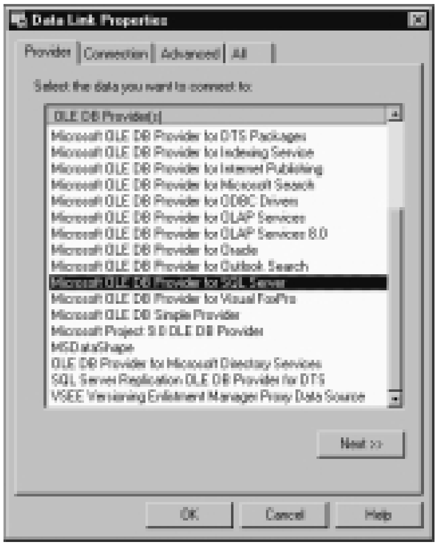
Connection Strings

Connections are simpler than ever. An easy way to get a connection string is to create an empty text file. Rename it using "udl" (Universal Data Link) for the extension.

Exercise 9.1 Creating a Connection String

1. Create a file with the name **MyConnection.txt** and then rename it to **MyConnection.udl**. After opening MyConnection.udl from Explorer, the Data Link Properties dialog box shown in Figure 9.1 appears.

Figure 9.1 Data Link Properties Dialog Box



Select the data you want to connect to:

OLE DB Provider(s)
Microsoft OLE DB Provider for DTS Packages
Microsoft OLE DB Provider for Indexing Service
Microsoft OLE DB Provider for Internet Publishing
Microsoft OLE DB Provider for Microsoft Search
Microsoft OLE DB Provider for ODBC Drivers
Microsoft OLE DB Provider for OLAP Services
Microsoft OLE DB Provider for OLAP Services 8.0
Microsoft OLE DB Provider for Oracle
Microsoft OLE DB Provider for Outlook Search
Microsoft OLE DB Provider for SQL Server
Microsoft OLE DB Provider for Visual FoxPro
Microsoft OLE DB Simple Provider
Microsoft Project 9.0 OLE DB Provider
MSDataShape
OLE DB Provider for Microsoft Directory Services
SQL Server Replication OLE DB Provider for DTS
VSEE Versioning Enlistment Manager Proxy Data Source

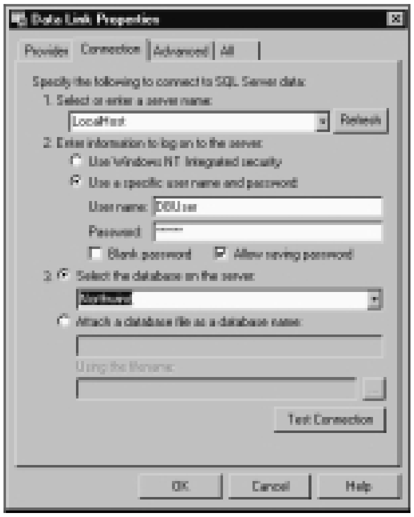
2. Select the **Provider** tab and then pick your OLE DB provider. Figure 9.2 is an example of the provider-specific dialog that will prompt you for information for building a SQL Server connection string.

   The resulting text will look something like this:

```
[oledb]
; Everything after this line is an OLE DB initstring
Provider=SQLOLEDB.1;Password=;Persist Security Info=True; _
User ID=DBUser;Initial Catalog=Northwind;Data Source=LocalHost
```

   You can paste this into your code without worrying about whether the parameters are set up correctly.

Figure 9.2 Connection Options for a SQL Server OLE DB Provider



Data Link Properties

Provider | Connection | Advanced | All

Specify the following to connect to SQL Server data:

1. Select or enter a server name: LocalHost

2. Enter information to log on to the server:
   - ( ) Use Windows NT Integrated security
   - (•) Use a specific user name and password:
     - User name: DBUser
     - Password: ******
     - [ ] Blank password [✓] Allow saving password

3. (•) Select the database on the server: Northwind
   - ( ) Attach a database file as a database name:
   - Using the filename:

Developing & Deploying…

Connection Pooling

Connection pooling for the SqlClient connections is handled in Windows 2000 Component services. Each connection pool is differentiated using a unique connection string using an exact matching algorithm. In other words, all we have to do to take advantage of connection pooling is use the same connection string. In addition, since the SqlConnection is being hosted in Windows 2000 Component services, we can take advantage of the resource management that Component services provides.

We do have some options that we can include in the connection string to modify the default behavior of connection pooling for the SqlConnection object. These options are well documented in the Framework SDK.

Connection pooling for the OleDbConnection object is handled using OLE DB session pooling, which is handled by the OLE DB provider, not ADO.NET. As with SqlConnection pooling, we use identical connection strings to differentiate the pools, and we can modify the behavior of the pool using connection string arguments. These arguments are not documented in the Framework SDK; they are specific to OLE DB, and are not the same as the SqlConnection options. Therefore, the connections are not portable across namespaces if they modify the connection pools.

Command Objects

The Command objects OleDbCommand and SqlCommand allow us to execute statements directly against the database. They provide for simple and direct route to our data, regardless of where the data resides. They can have a collection of parameters that are used to pass variables in and get variables out. If you need to get the return value of a stored procedure, the Command object is the object to use. Command objects are particularly useful for executing INSERT, UPDATE, and DELETE statements, but can also generate DataReader and XMLDataReader objects for returning data:

```
Dim strSql As String = "SELECT * FROM Orders"
Dim sConn As String = "Provider=SQLOLEDB.1;" & _
                  "Password=password;" & _
                  "Persist Security Info=True;" & _
                  "User ID=sa;" & _
                  "Initial Catalog=Northwind;" & _
                  "Data Source=localhost"
Dim myConnection = New OleDbConnection(sConn)
Dim myCmd As OleDbCommand = New OleDbCommand(strSql,

    myOleDbConnection)
```

These are useful for database operations that do not require records to be returned. They also give us an object to use for parameterized stored procedures and process output values and return values when necessary.

The Command objects are particularly suited for calling stored procedures, which is the preferred method for data access. Stored procedures allow some Relational Database Management Systems to precompile and take advantage of statistics that it has gathered on the source tables. For example:

```
CREATE PROCEDURE getShippers AS
Select *
From shippers
Order By CompanyName
```

This stored procedure just returns an ordered list of records from the Shippers table in the Northwind database. To call this procedure, we can use a couple of different syntaxes. We can just specify the name of the stored procedure instead of a SQL statement, or we can create a Command object explicitly. Here is an example of replacing a select statement with the name of a stored procedure:

```
'strSql = "SELECT * FROM Shippers"
strSql = "getShippers"

objOleDbCommand = New OleDbCommand(strSql, myOleDbConnection)
```

Here we commented out the line with the select statement in it and inserted the stored procedure name. For a better example, let's add an input parameter. By adding a parameter to the stored procedure, we can now limit the rows that our application uses and make it more efficient. For example, say we add a parameter to the stored procedure that is used to find a shipper with a particular ShipperID. To call it, we just add the parameter in the order required by the stored procedure. In our case, with one parameter, it would look like this:

```
strSql = "getShippersByID 2"
```

This method is fine when you are only trying to get some records back from a stored procedure, but not very useful if you are trying to get an output value or a return value. Here is where the Parameter objects come into play. To implement our example with a parameter:

```
Dim strSP As String
Dim objOleDbCmd As OleDbCommand
Dim objParam As OleDbParameter
Dim objConnection As OleDbConnection
Dim objAdapter As OleDbDataAdapter
Dim myDataSet As DataSet

Try
strSP = "getShippersByID"
```

Get the new connection to the database. If we have a connection that is available, we could use it instead of creating a new one—the fewer connections, the better:

```
objConnection = New OleDbConnection(sConn)
objConnection.Open()
```

Instantiate a new Command object and specify the new connection we just created. Set the type of command to stored procedure:

```
objOleDbCmd = New OleDbCommand(strSP, objConnection)
objOleDbCmd.CommandType = CommandType.StoredProcedure
```

The following code does several things. First, starting from the inner parentheses, we are creating a new OleDbParameter with a DataType of unsigned integer and a size of 4. Then it adds this new parameter to the parameters collection of the Command object that we just created. Finally, we put a reference to this newly created Parameter object in the variable *objParam*:

```
objParam = objOleDbCmd.Parameters.Add(New OleDbParameter("@ID", _
    OleDbType.UnsignedInt, 4))
```

Here we are setting the direction of the parameter and its value. The value is easy enough to explain, but the direction is a little more complicated. Refer to Table 9.2 for an explanation of the different options we have for parameter direction:

```
objParam.Direction = ParameterDirection.Input
objParam.value = intShipperID
```

This line of code sets the SelectCommand of our DataAdapter to the newly created CommandObject objOleDbCmd. We have the option of specifying **SelectCommand**, **InsertCommand**, **DeleteCommand**, and **UpdateCommand**:

```
objAdapter.SelectCommand = objOleDbCmd
```

Here we fill our dataset using the *Fill* method of the Adapter object:

```
objAdapter.Fill(myDataSet)
```

Now, all that is left is to set the data source of our DataGrid and complete the Error handler:

```
DGorders.DataSource = myDataSet
Catch e As Exception
     MsgBox(e.ToString)
Finally
     objConnection.Close()
End Try
```

Table 9.2 Parameter Directions

Member	Name Description
Input	The parameter is an input parameter. This allows for data to be passed into the command, but not out. We may have more than one.
Output	The parameter is an output parameter. It is used to return variables, but cannot be used to pass data into a command. The command must be written specifically to populate this variable as part of its routine. We may have more than one.
InputOutput	The parameter is capable of both input and output. It is used when we need to pass data in to and out of a command in one object. It does exactly as its name implies, it performs both the input and the output operations. We may have more than one.
ReturnValue	The parameter represents a return value. This is similar to the output parameter, except that we can only have one.

This example demonstrated the use of an OleDbCommand object to populate a DataSet. We passed a reference of the OleDbCommand object we created to the SelectCommand property of the DataAdapter. When we called the **Fill** method, it used our OleDbCommand object to execute a DataReader and populate our DataSet.

We had to create a Parameter object, set its direction to Input, and then its value. It is interesting to note that in ADO we made up our own names for the Parameter objects that we created. In ADO.NET, we must ensure that our parameters are named the same as they are in the definition of the stored procedure. ADO.NET uses them to implement named parameters and will throw an exception if it doesn't find a match. Of course, DataTypes must also match.

To get an output parameter, we can modify our stored procedure to return the current day of the server just as a demonstration of the output parameter. You can easily turn this into an example of returning the ID of a newly created record:

```
objParam = objOleDbCmd.Parameters.Add(New
    OleDbParameter("@CurrentDay",_
    OleDbType.Date, 8))
objParam.Direction = ParameterDirection.Output
```

Accessing this value after the *OleDbCommand.ExecuteNonQuery* method had been called is simple:

```
dtServerDate = objSQLCmd.Parameters("@CurrentDay").Value
```

As we can see here, using the stored procedure in the SQL statement is simpler, but not as flexible. We can also access the return value using a similar technique. The only difference in using the return value is that we must declare a parameter with the name "RETURN VALUE," and a direction of type return value. After that, we access it just like any other output value. The return value from a SQL Server stored procedure can only be a DataType of Integer. If the preceding example were something like the number of days since an order date, we could use these lines of code to get it. The stored procedure might look something like this:

```
CREATE PROCEDRUE GetDaysSinceLastOrder(@CustID nChar(5))
    AS DECLARE @iDays INT

Select @iDays = DATEDIFF(dd, Max(OrderDate), GETDATE())
From Orders
Where CustomerID = @CustID
Return @iDays
```

The VB code to create the parameter and get the return value should look something like this:

```
objParam = objOleDbCmd.Parameters.Add(New OleDbParameter("RETURN
    VALUE"_ , OleDbType.Char, 5))
objParam.Direction = ParameterDirection.ReturnValue
```

Play around with this object; it is probably going to be one of the most used in your toolbox. Understanding how to use the output values and returning data from them will be essential to your high-performance development.

DataReader

The *DataReader* is a read-only, forward-scrolling Data object that allows us to gain access to rows in a streaming fashion. It is typically used when we need read-only access to data because it is much faster than using a DataSet. A DataSet is populated behind the scenes using a DataReader, so if we don't need the features of a DataSet, we should not create one. A DataReader is created either from the OLE DB libraries or from the SqlClient libraries. Here is a simple example of creating an OleDbDataReader from a Command object:

```
Dim myReader As OleDbDataReader = myCmd.ExecuteReader()
```

We now have a populated DataReader object that we can use like this:

```
While myReader.Read
        '// do some row level data manipulation here
End While
```

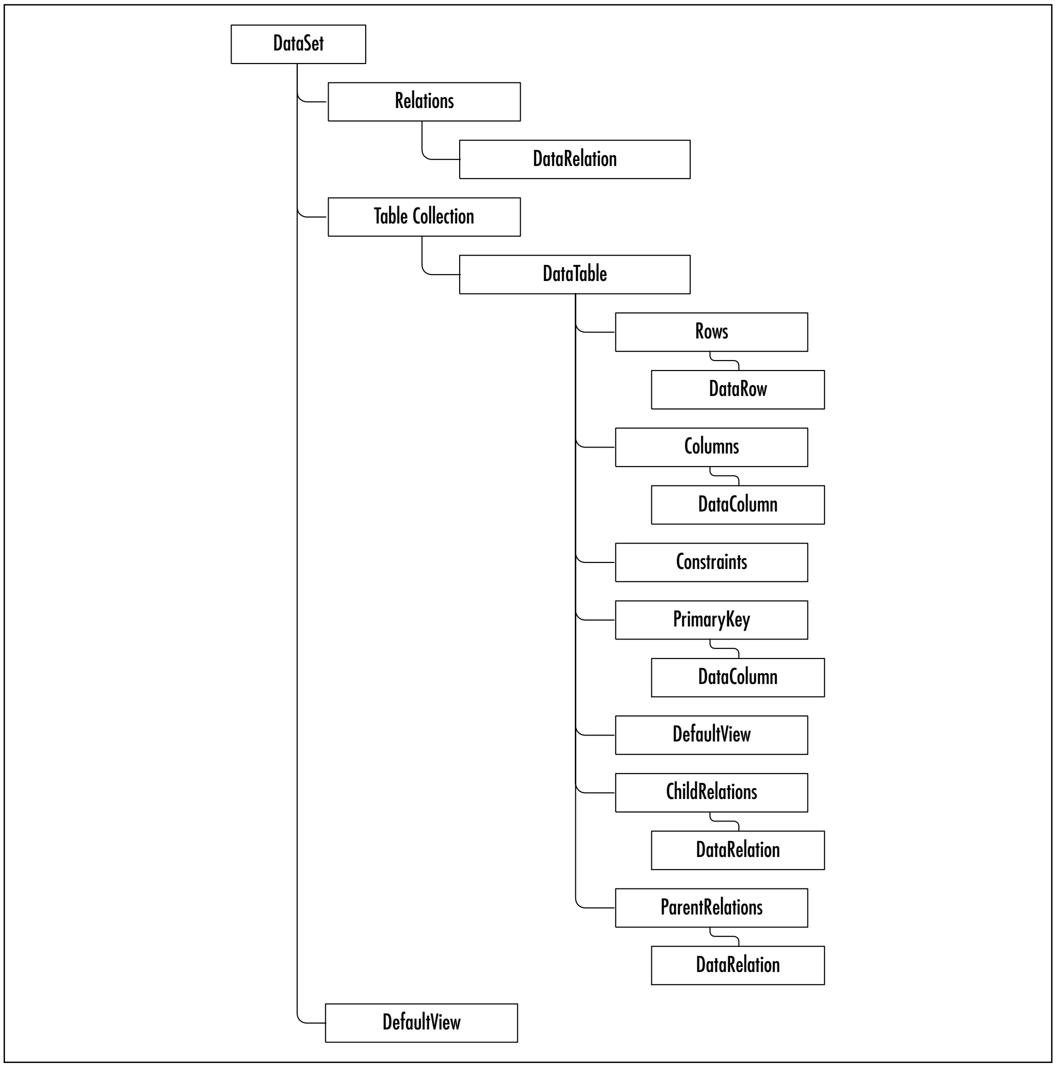
The DataReader object allows for much greater speed, especially if you need to access a large amount of data. It does not allow us to update information, nor does it allow us to store information as the DataSet object does.

DataSet

A *DataSet* is an in-memory copy of a portion of the database in which we are interested. This may be one table, or many tables. Imagine a small relational database residing in a variable. This is a complete copy of the requested data. It is completely disconnected from the original data source and doesn't know anything about where the data came from. By that, I mean that we could populate the data from XML from our Microsoft BizTalk Server, save it to Microsoft SQL Server, and then write it out to an XML file.

When we are finished with our operations, the entire DataSet is submitted to the data source for processing. It takes care of standard data processing such as updating, deleting, and inserting records. The DataSet object is a key player in the ADO.NET object model. Examine the object model in Figure 9.3 for the DataSet object and the collections it can contain. Due to the architecture of ADO.NET, it is possible for several combinations of collections. Take the Columns Collection as an example. As you can see, the DataTable object has a Columns collection made up of DataColumn objects. The PrimaryKey property of the DataTable contains collection of DataColumns as well. This is the same DataColumn object in the DataTables.Columns collection, but two different instances of it.

Figure 9.3 DataSet Object Model



Disconnected Layer

By its very nature, ADO.NET is disconnected, which means that we do not maintain a connection to the data source. We open a connection, retrieve the data, and close the connection. This allows our data source to free up resources, and respond to the next user.

This may place a higher premium on bandwidth, so we need to be as specific as possible when requesting data. We are going to have to refrain from *Select* *

From MyTable statements, and instead, retrieve only the columns we are actually using. In addition, make sure to use well thought-out Where clauses to limit the number of rows being returned. If all you need are names beginning with "A," then don't select "A–Z." Most of this is common sense; however, when you are battling a particular piece of functionality, it easy to let the rules slip. Just be aware that we are bringing that entire set of data through our connection.

Using DataSet

A DataSet holds a copy of all the data we request. It may contain more than one table, and we can set up relationships within the DataSet that may not be present in the database. For example, we can fill the DataSet with Orders and OrderDetails tables. We can then create a relationship between them and step through each OrderDetail in each order.

DataSets are populated from a DataReader implicitly. ADO.NET creates a DataReader and populates the dataset for us. We don't see this happening; we just call the *Fill* method and let ADO.NET do the work for us. The DataSet will temporarily hold data until we pass it along in the form of a DataTable or DataView. In addition to holding our data, the DataSet offers some features that were reserved for relational databases in the past; for example, primary keys, relationships, and constraints, to name a few. If you have worked with relational databases, these terms should sound familiar, and their implementations in ADO.NET are true to their database counterparts.

Relational Schema

With the DataRelation object, we have some new functionality that we never had before. A DataRelation object allows us to specify simple join criteria between DataTables in our Tables collection. This allows us to simplify code operations for situations that call for a hierarchy, or otherwise shaped data. Shaped data is one of the primary reasons why XML is so useful in our data operations. XML lends itself to building data in a parent-child relationship, instead of a flat file format. If you have been working with databases for any length of time, you can see the similarities. A parent-child relationship is typically thought of as a one-to-many relationship. One table contains the parent, or header records such as the order date, customer information, OrderID, and so forth, and the other can contain the line items for the order. In this scenario, we can have one order with many items. There are entire books written about the theories behind these concepts; for our purposes, we will restrict ourselves to a simple one-to-many parent-child relationship. In the past, to get Orders and OrderDetail records in one set of data, for every row in the OrderDetails record, we had all the Order record information as well. This required some ugly code to get the parent-child relationship in front of the end user.

To set up a DataRelation, we need two DataTable objects in our DataSet: a Primary key and a Foreign key. These will establish our Parent objects and our Child objects. For example:

```
Dim myds As DataSet = New DataSet("Orders")
Dim MyCol() As DataColumn
```

Add a Primary key to the DataTable to define the parent in the relationship:

```
MyCol(0) = myds.Tables("Orders").Columns("OrderID")
myds.Tables("Orders").PrimaryKey = MyCol
```

Add the relation using the Primary key we just created. Note: Do not use the Primary key of the Child object; this will essentially restrict us to one item per order and break our business rule:

```
myds.Relations.Add("Orders",
    myds.Tables("Orders").Columns("OrderID"), _
    myds.Tables("OrderDetail").Columns("OrderID"))
```

We can also place constraints on a DataTable object to give us even more relational power. A ForeignKeyConstraint object requires a matching record in the Parent DataTable. For example, we could not add a OrderDetail record without first have a matching Order record. For example:

```
Private myDataSet As DataSet
Private Sub CreateConstraint()
```

Declare parent column and child column variables:

```
Dim dColOrder As DataColumn
Dim dColOrderDet As DataColumn
Dim myFK As ForeignKeyConstraint
```

Set parent and child column variables:

```
dColOrder = MyDataSet.Tables("Orders").Columns("OrderID")
dColOrderDet =
       MyDataSet.Tables("OrderDetails").Columns("OrderID")
myFK = New ForeignKeyConstraint("OrderFK", dColOrder, dColOrderDet)
```

Set rules to indicate what action is to be performed when a Foreign key is enforced:

```
myFK.DeleteRule = Rule.Cascade
myFK.UpdateRule = Rule.Cascade
```

In this instance, if we were to delete the parent record, the DeleteRule would cascade the delete to the OrderDetails table and delete the child records as well. With the UpdateRule, if the Primary key is updated, the Foreign key is updated as well. This ensures that our records stay related, even if we change the Primary key value. Changing Primary keys is taboo in database design, but this allows us to break the rules occasionally.

The AcceptRejectRule.Cascade rule will cascade the changes at the data source during the *AcceptChanges* method of the DataSet object. It is important to realize that we have created the relationship outside of the database. The database need not have these relationships defined:

```
myFK.AcceptRejectRule = AcceptRejectRule.Cascade
```

Add the constraint to the parent and set EnforceConstraints to true. This is the final step to set up the relationships in our DataSet object:

```
myDataSet.Tables("Orders").Constraints.Add myFKC
myDataSet.EnforceConstraints = True
      End Sub
```

Collection of Tables

A DataSet contains a collection of tables. These tables are tabbed representations of our data and are the key to the DataSet's versatility. Essentially identical to the tables in our database, or other data source, they are added to our DataSet in the same way we add objects to other collections. Once in our DataSet, we can define properties such as the DataRelations, Primary keys, and so forth. DataTables can be created programmatically, or retrieved from a database through a SqlDataAdapter/OleDbDataAdapter object using the *Fill* method.

After we populate our DataSet with DataTable objects, we can access these tables using an index or the name we gave the table when we added it to the DataSet. The collection uses a zero-based index, so the first DataTable is at index 0:

```
ds.Tables(0)
```

The preceding code is more efficient, but harder to read; the following code is easier to read, but a little less efficient. How inefficient is yet to be determined, but, generally speaking, your users won't be able to tell. Therefore, unless you have a compelling reason to use the index, the following code will be easier to maintain:

```
ds.Tables("Orders")
```

The Tables collection is the basis for DataSet operations. From the collection we can pull tables into separate *DataTable* variables and DataView objects, we can bind them to DataGrids and DataLists, or act on them in the collection as in the previous examples.

Data States

Data has many states. When we change data, ADO.NET performs some amazing trickery on our behalf. ADO.NET maintains the various states, and versions of our data as we manipulate it. This allows us to perform all sorts of validation based on previous values since the last AcceptChanges call. As data is added, updated, or deleted in our application, we can access the versions of the data, display it to the user, and give them some pretty powerful undo functionality. It is important to note that this is available until we accept the changes and send them back to the data source.

A table is comprised of DataRow objects. The DataRow objects allow access to the entire row of data, but we also have versioning capability built into the row. The DataRow maintains the versions of data listed in Table 9.3.

Table 9.3 Possible DataRowVersion Values for a DataRow Object

Member Name	Description
Default	Default values for a row that was added.
Original	Values in the DataTable when the DataTable was added to the DataSet.
Proposed	Data that has been added or updated, but not committed.
Current	Current data in the row as of the last time the row was committed.

For example, suppose our business rule will not allow us to save a change to a record if the OrderAmount has not been updated. We first check to see if the row has a version; if it does, we compare the proposed value and the current one. If they match, the OrderAmount was not changed, and we reject the changes:

```
If MyRow.HasVersion(DataRowVersion.Proposed) Then
    If MyRow("OrderAmount"), DataRowVersion.Proposed) = _
        MyRow("OrderAmount"), DataRowVersion.Current) Then
          Ds.RejectChanges()
   End IF
End IF
```

In addition to row versions, we also have a RowState property that can give us more information about the condition of a row. Table 9.4 lists the possible states a row can be in.

Table 9.4 Possible RowState Values for a DataRow Object

Member Name	Description
Unchanged	No changes since last AcceptChanges call.
New	The row was added to the table, but AcceptChanges has not been called.
Modified	A change has been made, but AcceptChanges has not been called.
Deleted	The *Delete* method was used to delete the row from the table.
Detached	The row has either been deleted, but AcceptChanges has not been called, or the row has been created, but not added to the table.

With this type of versioning and state information we can really leverage ADO.NET to enforce some pretty bizarre and wonderful business rules. This is especially true where you are sending and receiving DataSets through a Web service, or remoting to and from dissimilar systems.

Populating with the DataSet Command

The DataAdapter object allows us to populate DataTables in a DataSet. We call the *Fill* method after constructing the object. Pass in the DataSet, and a name for our DataTable object, and ADO.NET takes care of the rest. For example, to use an OleDbDataAdapter to populate a DataTable:

```
Dim myDS As DataSet = New DataSet("myDataSet")
Dim myAdapter As OleDbDataAdapter = New OleDbDataAdapter( _
"SELECT * FROM Orders", sConn)
myAdapter.Fill(myDS, "Orders")
```

That is pretty easy. To use the SQL data provider, you just have to substitute SqlDataAdapter for OleDbDataAdapter. The *Fill* method is constructed the same way, but results in tighter integration with SQL Server and allows for better performance.

Populating with XML

We can populate a DataSet directly from a well-formed XML file. By *well formed*, I mean an XML file that adheres to W3C guidelines for properly formatted XML tags. Using the *ReadXML* method of a DataSet object, we can populate a DataSet with XML. This allows us to access and manipulate XML data as easily as we manipulate any other type of data. Once we are finished manipulating it, we can use the *WriteXML* method to save back to the original file, or create a new file. A simple example to read a file in, manipulate some data, and write a file out looks like this:

```
dsXML = New DataSet()
       dsXML.ReadXml("Orders.xml")
DataGrid1.DataSource = dsXML
```

All we did was write a few lines of code, and the DataSet took care of parsing the XML file, generating the DataTable and DataColumn objects, and then importing the data from the XML file. When we run our project we see the contents of the XML file in our data grid. To write the file out, we use the following syntax:

```
dsXML.WriteXML("Orders.xml")
```

Notice that in both examples, I am not specifying a path to these files. Our application expects to find them in the same folder in which the executable resides. An easy way to create an XML file for the example is to connect to a database, populate a DataSet, and use the *WriteXML* method to extract the data into an XML file. This is also useful if you need to quickly generate an XML file to transmit data to a distant location, or for archival purposes. ADO.NET—specifically, the DataSet—allows us to quickly and easily work with XML data, from the data aspect. We can make changes to the schema of the XML by making changes to the DataColumn and DataTable objects in our newly created DataSet. These changes will be reflected when we write the data out.

It may not be obvious here, but by using the *ReadXML* method, we are not connecting to the XML file as we would a database; we are importing the data. This effectively means that we are copying the data, so if you need to persist your changes, you also need to write out the XML file when you are finished with the data.

Populating Programmatically

We can create a DataTable object and populate it manually. That is, we can construct an empty DataTable, and then, using the *DataTable* methods to add columns and rows, we can create and populate a DataTable without having to connect to a database. For example, I want to create a table to hold connection strings, but it will be a local file that my application uses. The application will load this file on startup, and I can cycle through the connections to find an available database server—sort of like a poor man's failover. Here is a portion of that code; refer to the chapter samples for a more entertaining example.

Create an empty DataTable:

```
Dim myTbl As New DataTable("tblConn")
Dim myDS As New DataSet("ConnString")
```

Create the columns to hold our connection information:

```
Dim colProvider As New DataColumn("Provider")
```

Create the DataRow:

```
Dim rowConn As DataRow
```

Add the new table to the DataSet:

```
myDS.Tables.Add(mTbl)
```

Now, add the columns to the DataTable:

```
colProvider.DataType = System.Type.GetType("System.String")
myDS.Tables("tblConn").Columns.Add(colProvider)
```

First, create the row:

```
rowConn = mDS.Tables("tblConn").NewRow
```

Second, add the new row into the DataTable:

```
myDS.Tables("tblConn").Rows.Add(rowConn)
```

Populate the columns with data:

```
rowConn("Provider") = "SQLOLEDB.1"
```

From here on we can use the DataTable just as if it were created from a multi-terabyte server. Get the idea behind the portability of our code? Same code, just change the source and we open up new avenues for our application to expand into.

Using the SQL Server Data Provider

The SQL Server data provider is written explicitly to provide data access to Microsoft SQL Server version 7 and later. This set of classes takes advantage of the SQL Server API in a way that makes it more efficient for data access than going through the OLE DB libraries. Think of it as native access to Microsoft SQL Server for ADO.NET.

Some of the ADO.NET objects are a little different for the SQL libraries. They implement the same interfaces; the only difference is really the name as far as we are concerned. Of course, the underlying implementation is different as well.

The SQL Server data provider's core objects are SqlConnection, SqlDataAdapter, SqlCommand, and SqlDataReader. There are other objects and events, but these are the main ones we will focus on day to day. The examples in the chapter can be executed against these objects by changing the namespace from System.Data.OleDb to System.Data.SqlClient. Then, change the object types from OleDbxx to Sqlxx. For example:

```
Dim myDS As DataSet = New DataSet("myDataSet")
Dim myAdapter As OleDbDataAdapter = New OleDbDataAdapter( _
    "SELECT * FROM Orders", sConn)
myAdapter.Fill(myDS, "Orders")
```

becomes:

```
Dim myDS As DataSet = New DataSet("myDataSet")
Dim myAdapter As SqlDataAdapter = New SqlDataAdapter( _
    "SELECT * FROM Orders", sConn)
myAdapter.Fill(myDS, "Orders")
```

This makes it easy to change from, say, Oracle to SQL Server, at least from this standpoint. The devil is in the details, so take this type of project cautiously. Switching an RDBMS is not trivial, but ADO.NET goes a long way to making life easier for the VB.NET programmer. However, we still have to work out the structural differences in the databases, for instance.

If, on the other hand, the RDBMS your project is to reside on is not chosen, you could safely use the OLE DB or ODBC provider and continue developing. You won't get the tight integration to any database product, but that is not always the most important aspect of a project. Many times, the driving force behind a project is time to market, and it may not be wise to let deadlines slip waiting on the often-political process of choosing a relational database. Think of the OLE DB managed provider as the great RDBMS equalizer, at least from the data access point of view.

NOTE

While generating SQL command text, use the *FormatDateTime* method to convert DataTime information into localized strings. This will return a string that will work with SQL Server. SQL Server currently does not recognize the ISO 8601 date format and will throw an error:

```
'// Will result in "1999-05-16T12:40:30"
MyDate.ToString()
Imports Microsoft.VisualBasic
'// would result in "5/16/1999"
FormatDateTime(MyDate, DateFormat.ShortDate)
```

TDS

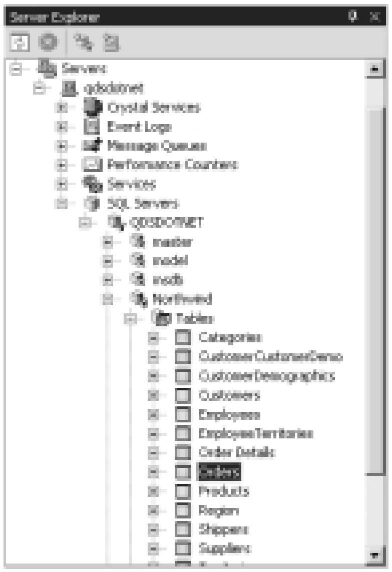
TDS stands for *TypedDataSet*, also known as a *Strongly TypedDataSet*. A TypedDataSet is bit of wizardry that the Visual Studio IDE does for us. What we are talking about here is basically early binding to our tables and data columns. They are created by using a class derived from the *DataSet* class and are combined with a Schema Definition file to provide the impression of *early binding*. What I mean by early binding is that the VB compiler will enforce the column naming and datatyping during compile time. See CD file Chapter 09/ Chapter9 Beta2/Samples/XML/MyData.xsd.

Exercise 9.2 Using TypedDataSet

1. Open Visual Studio and create a new Windows Application.
2. From the Server Explorer, expand the **Server tab | SQL Server Databases | Databases | Northwind | Tables**, and drag the **Orders Table** onto the form. See Figure 9.4 for a view of the Server Explorer.

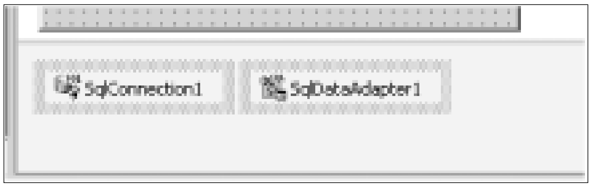
   My Server name in this graphic is **QDSDOTNET**, so expect your server name to appear here.

   Figure 9.4 Server Explorer

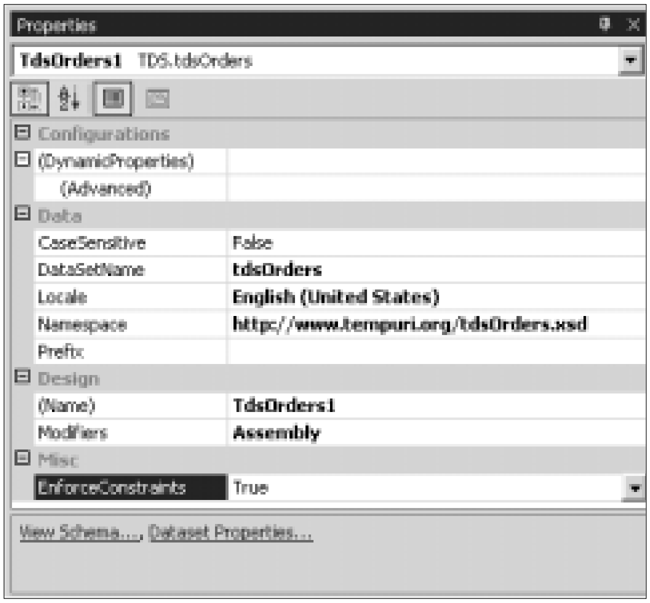
3. Visual Studio will create two Data objects, a SqlConnection object and a SqlDataAdapter object. These do not show up on the form; instead, they show up on the Component Tray in Visual Studio. Figure 9.5 shows an example.

   Figure 9.5 Data Objects on the Component Tray

4. Notice that in the menu bar at the top of the screen is an additional menu called **Data**. Pull this down and select **Generate DataSet**. We can also right-click on the form and select **Generate DataSet** from the available options. Note that this functionality is not available until we add a SqlConnection and SqlDataAdapter objects to the form. When we select the **Generate DataSet** command, a dialog box is presented. Enter **tdsOrders** in the **New** text box, and check the box to add this DataSet to the designer. Make sure that the Orders table is checked in the list of available DataAdapters. The IDE executes the XSD.EXE and creates a schema file to add to our assembly. It also creates a file called **tdsOrders.vb** with the code that implements the TypedDataSet. This file actually contains many classes that inherit from various parts of the *System.Data* classes. The result is a custom class that contains all the standard *DataSet* methods and properties, in addition to some added functionality to enforce datatyping and schema enforcement for the Orders table. Set the focus to the tdsOrders1 DataSet object on the Component tray and examine the Properties dialog box in Figure 9.6.

Figure 9.6 Properties Dialog Box


Properties

TdsOrders1 TDS.tdsOrders

Configurations	
(DynamicProperties)	
(Advanced)	
Data	
CaseSensitive	False
DataSetName	tdsOrders
Locale	English (United States)
Namespace	http://www.tempuri.org/tdsOrders.xsd
Prefix	
Design	
(Name)	TdsOrders1
Modifiers	Assembly
Misc	
EnforceConstraints	True

View Schema..., Dataset Properties...

Notice the hyperlinks at the bottom of Figure 9.6. The View Schema link presents you with a graphical representation of your DataSet schema; the other link presents a dialog box with more detailed properties of the DataSet object. All of this comes at a price. A Strongly TypedDataSet is slower than its weaker typed sibling. The advantage is in speed of development, and the compile time enforcement of DataTypes. The wizard creates a DataSet, so if we only need read-only access to the data, a SQLDataReader would provide much more speed; not just with the same ease of development. In addition to the enforcement of column names and DataTypes during development and compilation, the TDS lets us access our DataTables and DataColumns using Intellisense.

Data access using a regular DataSet:

```
strOrderID = dsOrders.Tables("Orders")(0).Columns("Orderid")
```

Data access using a TDS:

```
strOrderID = dsOrders.Orders(0).Orderid
```

In my testing, data access took nearly twice as long using a TDS when compared to a more traditional approach. We are only talking milliseconds with DataSets this small, so most users will probably not notice the difference. At this point, the argument gets academic, and you will have to weigh the benefits. What is more important to your project: time to market or trying to optimize CPU cycles?

Remember that to set this up, all we did was drag and drop. After that, we could reference a column just like it was a native property—no misspelling the column names, and DataTypes are enforced at compile time. This speeds up development in several ways: no more having to remember the column names, and the compiler will keep us from making mistakes with DataTypes.

Remoting

The act of *remoting* is not new. A Web request is a form of remoting, and a useful example of what we are trying to accomplish with .NET remoting. It is a message-based communication methodology applied using standard technologies such as TCP, XML, HTML, and SOAP. We can leverage this to create truly distributed applications that take advantage of industry standards, and allow different platforms to unite into very useful applications.

Remote objects are defined as objects that need to communicate across application domains, and are derived from MarshalByRefObject. From then on, when a client activates a Remote object, it creates an instance of a Proxy object that redirects calls to and from the Remote object. This redirection does have performance penalties that you need to consider when developing a distributed architecture.

This communication takes place over three different channels. Basically, it comes down to the environment in which we are going to be operating. If clients are going to be accessing our service over the Web, then we will be using the HTTP remoting channels; namely, SOAP. SOAP is the combination of HTTP and XML combined per the SOAP specification to allow the production of Web services.

If the services are on a local network, with routers that we can control, then we can use TCP for faster, leaner, and more robust operation. TCP processes data and data requests through sockets that are specified in the configuration of the service.

Data Controls

Data controls allow us to control data and come in many forms. The data controls in VB.NET are similar to the data controls in VB6. They provide for flexible viewing and editing of data either in the form of bound data or unbound data. Which method you will employ depends largely on your particular project constraints. Generally speaking, bound controls allow for speed of development, but have traditionally limited the developer to basic functionality. For more control, we had to revert to unbound methods that were more involved, but definitely more flexible.

DataGrid

DataGrid is a spreadsheet-like representation of data. The catch here is that the DataGrid is optimized for access to relational data. This means that we can start with our parent records and drill down into the child records. The DataGrid comes in two implementations. The first and most obvious is the Windows Form version. The Windows Form implementation of the DataGrid differs quite a bit from its Web Form counterpart, so we will look at them both. The WinForm DataGrid can bind to many sources of data. The following list includes the out-of-the-box objects that we can use for our data source:

- DataTable
- DataView
- DataSet
- DataSetView
- Single dimension arrays

In addition to the preceding objects, any components that implement the following interfaces can also be used to populate a DataGrid:

- The IListSource interface
- The IList interface

First, the Windows Form version is a scrollable grid that when bound to a DataSet with relations defined will give us a view of our data that we can use to drill down into the detail records. Follow Exercise 9.3 to create a simple example of using a TDS with two tables and a DataRelation to create the drill-down functionality described previously. The Beta2 documentation contains a list of properties, methods, and events that would take an entire book to cover; suffice to say that much effort has been put into this object from a functionality standpoint. It is, however, very simple to implement, so let's do an exercise that builds on the TDS solution that we created earlier.

Exercise 9.3 Using TypedDataSet and DataRelation

1. Open Visual Studio and open the project we started during the Typed Data discussion.
2. Drag the **Order Details** table onto our form as we did in Figure 9.4.
3. Select **Generate DataSet** from the **Data** menu, and select the existing **TypedDataSet**, making sure that both tables are selected in the available Adapters.
4. Drag a **DataGrid** and a **Button object** from the toolbox onto the form.
5. Enter these few lines of code into the **Buttons Click Event Handler** (**Button1_Click**, for example). This uses the SqlDataAdapters that were wizard generated to populate our TypedDataSet:

   ```
   SqlDataAdapter1.Fill(TdsOrders1.Orders)
   SqlDataAdapter2.Fill(TdsOrders1.Order_Details)
   ```

   This line sets up a parent-child relationship between the Orders and the Order_Details DataTables:

   ```
   tdsOrders1.Relations.Add("relOrders",
       tdsOrders1.Orders.OrderIDColumn, _
       tdsOrders1.Order_Details.OrderIDColumn)
   ```

This line binds the TypedDataSet to the DataGrid:

```
DataGrid1.DataSource = tdsOrders1
```

6. Set the **Anchor property** of the DataGrid to **ALL** and build the solution. The finished form should look like Figures 9.7 and 9.8. After you press **F5**, click the button, and expand the default node.

Figure 9.7 Finished Form

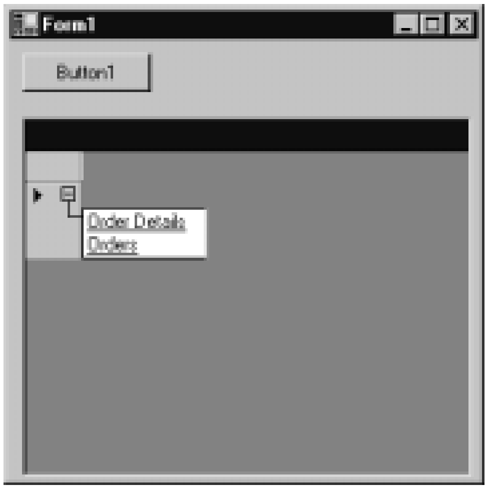
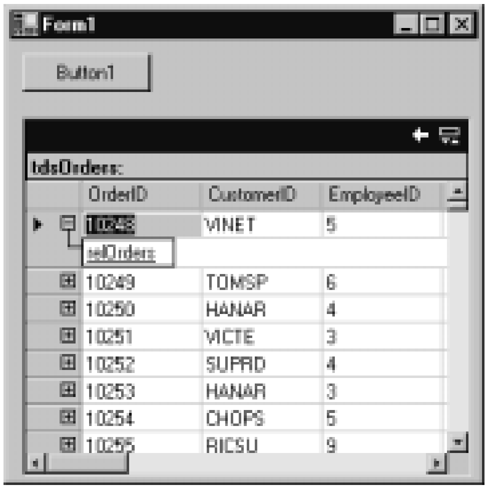


Figure 9.8 Orders Table



tdsOrders:

OrderID	CustomerID	EmployeeID
10248	VINET	5
10249	TOMSP	6
10250	HANAR	4
10251	VICTE	3
10252	SUPRD	4
10253	HANAR	3
10254	CHOPS	5
10255	RICSU	9

Click the **plus sign** to get a list of tables in our DataSet. Select the **Orders table** from the drop-down to get the view shown in Figure 9.8.

In our example, we start out with a list of orders, and we can expand them by clicking the plus sign at the record selector in the far left of the grid. This provides us with a list of relations. In our example, we only have the one, so select it, and the grid will change to show the detail records for that order. When you consider that we started with a project that consisted of dragging and dropping some objects from the Server Explorer and the toolbox, entering half a dozen lines of code, this is pretty powerful stuff!

The WebForm DataGrid object in ASP.NET is quite a bit different as far getting one to operate. A few lines of code are all it takes to put one of these to work with some basic functionality. The following example creates a basic HTML table of our data with just a few lines of code. This is the line of code that will actually place the control on the form for us. Place it where you would like the grid to show up on the page (see CD file Chapter 09/Chapter9 Beta2/Samples/wwwroot/Chapter9/DataGridSample.aspx):

```
<asp:DataGrid id="DGOrders" runat="server"/>
```

All that is left to do is to populate the grid with a data source:

```
Private sConn As String = "<MyConnectionstring>"
Private dsShippers As DataSet

Private Sub BindData()
     Dim strSQLQuery As String
     Dim objConn As SqlClient.SqlConnection
     Dim objAdapter As SqlClient.SqlDataAdapter

     strSQLQuery = "SELECT * FROM Shippers"
```

Create a connection to the SQL Server:

```
objConn = New SqlClient.SqlConnection(sConn)
```

Create a SqlDataAdapter using the connection and the SQL statement as parameters for the constructor:

```
objAdapter = New SqlClient.SqlDataAdapter(strSQLQuery, objConn)
```

Create the DataSet and fill it with the SqlDataAdapter object we created earlier:

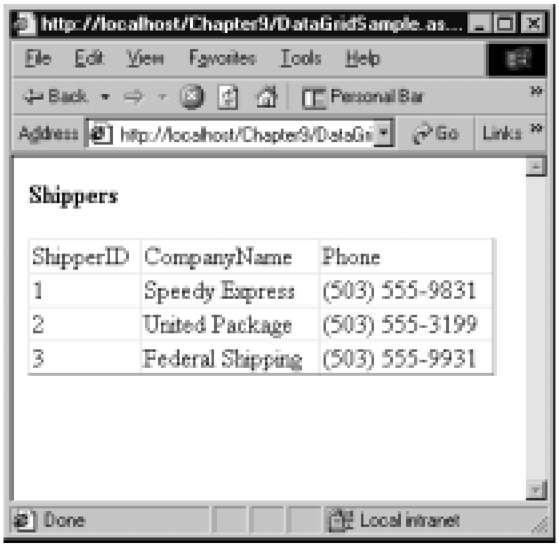
```
dsShippers = New DataSet()
objAdapter.Fill(dsShippers, "Shippers")
```

Set the DataSource of our data grid to the newly created DataSet, and call the *DataBind* method to finish the grid:

```
DGShippers.DataSource =
dsShippers.Tables("Shippers").DefaultView
DGShippers.DataBind()
    End Sub
```

Call the BindData procedure from the *Page_Load* event. The finished product is shown in Figure 9.9.

Figure 9.9 DataGrid Sample


http://localhost/Chapter9/DataGridSample.as...
File Edit View Favorites Tools Help
Back Personal Bar
Address http://localhost/Chapter9/DataGri Go Links
Done Local intranet

Shippers

ShipperID	CompanyName	Phone
1	Speedy Express	(503) 555-9831
2	United Package	(503) 555-3199
3	Federal Shipping	(503) 555-9931

The WebForm DataGrid can be further enhanced with the Columns collection that may contain any or all of the objects in Table 9.5. These give us the power to create some very appealing functionality with only a few lines of code.

Table 9.5 Column Collection Objects

Column Name	Description
BoundColumn	Default column, it is bound to data and allows us to control the ordering and rendering of the column.
HyperLinkColumn	Presents the bound data in HyperLink controls. The text and URL of the hyperlink can be static or bound to data.
ButtonColumn	Represents a column with a set of push button controls. Bubbles a user command from within a row to an Event Handler on the grid. Not bound to data. The developer must write a handler to perform the action specified.
TemplateColumn	Defines a template used to format controls within a column.
EditCommandColumn	Displays Edit, Update, and Cancel links to allow the user to edit line items. The DataGrid will present the user with appropriate links and text boxes for the line being edited, while displaying the other rows in read-only text.

The EditCommandColumn creates the DataGrid with some options for the HTML output. It is also important to note that the OnEditCommand, OnCancelCommand, and OnUpdateCommand are mapped to server-side subprocedures that we have to write. Also, turn off the AutoGenerateColumns, which defaults to true:

```
<asp:DataGrid id="DGOrders" runat="server"
        BorderColor="black"
        BorderWidth="1"
        CellPadding="3"
        Font-Size="8pt"
        HeaderStyle-BackColor="#aaaadd"
        OnEditCommand="DGShippers_Edit"
        OnCancelCommand="DGShippers_Cancel"
        OnUpdateCommand="DGShippers_Update"
        AutoGenerateColumns="false"
        >
```

Here we define our columns. The first column is the EditCommandColumn. The options are self-explanatory. The EditText, CancelText, and UpdateText are the text that will show up in our hyperlinks to perform the desired action. The other columns are standard BoundColumns that display the data from the Shippers table:

```
    <Columns>
        <asp:EditCommandColumn
            EditText="Edit"
            CancelText="Cancel"
            UpdateText="Update"
            ItemStyle-Wrap="false"
            HeaderText="Edit Command Column"
            HeaderStyle-Wrap="false"
            />
        <asp:BoundColumn HeaderText="Shipper ID"
ReadOnly="true" _
        DataField="ShipperID"/>
        <asp:BoundColumn HeaderText="Company Name" _
              DataField="CompanyName"/>
        <asp:BoundColumn HeaderText="Phone"
DataField="Phone"/>
    </Columns>
</asp:DataGrid>
```

It is important to note that while both of these controls are called DataGrid, they inherit from different base classes. Microsoft has named many of the methods and properties for these controls the same, but do not take this for granted. The name may be the same, but the implementation is quite different.

DataList

A *DataList* is very similar to the WebForm DataGrid, with a few exceptions. It uses template tags, is a WebForm control, and inherits from the WebForm namespace. After adding a new WebForm to a Web project, place the following code between the opening and closing form tags. This block of code defines our DataList and binds the DataList_Click procedure to the *OnItemComand* event for the list (see CD file Chapter 09/Chapter9 Beta2/Samples/wwwroot/Chapter9/DataListSmaple.aspx):

```
<asp:DataList id="DLShippers"
       runat="server"
       Width="100%"
       OnItemCommand="DataList_Click"
       RepeatColumns="1" >
```

This block of code defines the default formatting of each row in our list:

```
       <ItemTemplate>
              <asp:Table Runat="server" Width="100%" ID=Table1>
                  <asp:TableRow>
                            <asp:TableCell>
                                   <asp:LinkButton id="Select" _
                                   runat="server" Text="Select "
                                    CommandName="Select"/>

                                   <asp:Label id="lblShipper" _
                                   runat="server">
              <% #DataBinder.Eval(Container.DataItem,
              "CompanyName") %>
                                   </asp:Label>
                            </asp:TableCell>
                     </asp:TableRow>
              </asp:Table>
       </ItemTemplate>
```

This block of code defines the formatting of the row we selected in the browser:

```
       <SelectedItemTemplate>
              <asp:Table Runat="server" Width="100%" ID=Table2>
                     <asp:TableRow BackColor="#EEEEEE">
                                   <asp:TableCell>
                                   <asp:Label id="lblPhone"_
                                   runat="server">
              <% #DataBinder.Eval(Container.DataItem, "Phone") %>
                                   </asp:Label>
```

```
                        </asp:TableCell>
                    </asp:TableRow>
                </asp:Table>
            </SelectedItemTemplate >
</asp:DataList>
```

The following code goes into the CodeBehind file created for your WebForm:

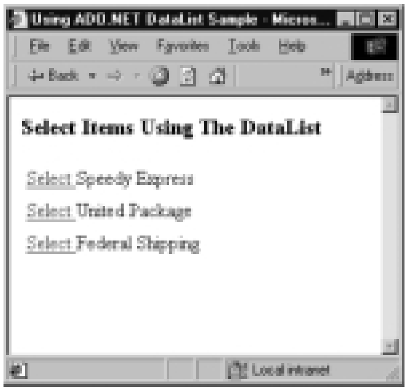
```
Private sConn As String = "<MyConnectionString>"
Private dsShippers As DataSet

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
        If Not Page.IsPostBack Then
            BindData()
        End If
    End Sub

    Private Sub BindData()
        Dim strSQLQuery As String
        Dim objConn As SqlClient.SqlConnection
        Dim objAdapter As SqlClient.SqlDataAdapter

        strSQLQuery = "SELECT * FROM Shippers"

        objConn = New SqlClient.SqlConnection(sConn)

        objAdapter = New SqlClient.SqlDataAdapter(strSQLQuery,
            objConn)

        dsShippers = New DataSet()
        objAdapter.Fill(dsShippers, "Shippers")
        DLShippers.DataSource =
            sShippers.Tables("Shippers").DefaultView
        DLShippers.DataBind()
    End Sub
```

This procedure handles the event bubbled up from the browser:

```
Public Sub DataList_Click(ByVal Source As Object, _
                      ByVal E As DataListCommandEventArgs)
    DLShippers.SelectedIndex = E.Item.ItemIndex
    Call BindData()
End Sub
```

Figure 9.10 shows the result.

Figure 9.10 DataList Output


Using ADO.NET DataList Sample - Micros...
File Edit View Favorites Tools Help
Back Address
Local intranet

Select Items Using The DataList

Select Speedy Express

Select United Package

Select Federal Shipping

This page shows the usage of the template tags and some of the options available. Template tags give us some options for formatting the different rows. They really make the data controls very flexible and remove many of the reasons we have for not using them. The available templates vary depending on which DataControl you are using, Table 9.6 lists the available templates for the DataList control.

Table 9.6 DataList Template Items

Template Name	Description
ItemTemplate	Required template that provides the style for items in the DataList.
AlternatingItemTemplate	Provides the style for alternating items in the DataList.
SeparatorTemplate	Provides the style for the separator between items in the DataList.
SelectedItemTemplate	Provides the style for the currently selected item in the DataList.
EditItemTemplate	Provides the style for the item currently being edited in the DataList.
HeaderTemplate	Provides the style for the header of the DataList.
FooterTemplate	Provides the style for the footer of the DataList.

Repeater

The *Repeater* object is a Web form control that allows us to create templates out of HTML, and then bind them to data—sort of like making our own grid control, except that we can use data binding. The tricky part of setting up the repeater is the use of template tags. An example is worth a thousand words, so here is a repeater bound to a DataSet that contains the Shippers table from the Northwind database (see CD file Chapter 09/Chapter9 Beta2/Samples/wwwroot/Chapter9/RepeaterSample.aspx):

```
<html>
<head>
<title>Using ADO.NET Repeater Sample</title>
</head>
<body>
<h3><font face=Verdana>Repeater Example</FONT></H3>

<form id=Form1 runat="server">
<b>Shippers:</b>
<p><asp:repeater id=RPTRShippers runat="server">
         <HeaderTemplate>
             <table border=1>
```

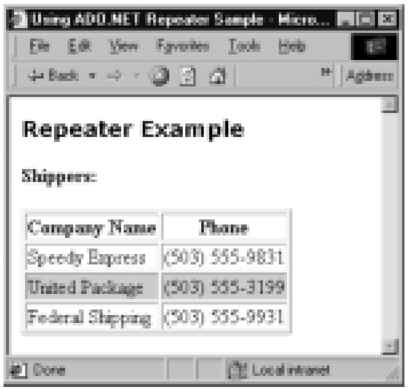
```
                <tr>
                  <th>Company Name</th>
                  <th>Phone</th>
                </tr>
          </HeaderTemplate>

     <ItemTemplate>
       <tr>
     <td> <%# DataBinder.Eval(Container.DataItem, "CompanyName") %> </td>
     <td> <%# DataBinder.Eval(Container.DataItem, "Phone") %> </td>
                </tr>
       </template>

          <AlternatingItemTemplate>
              <tr bgcolor="#cccccc">
                <td > <%# DataBinder.Eval(Container.DataItem,
                "CompanyName") %> </td>
                <td > <%# DataBinder.Eval(Container.DataItem, "Phone") %>
                </td>
              </tr>
          </AlternatingItemTemplate>

          <FooterTemplate>
              </table>
          </FooterTemplate>

     </asp:Repeater>
     </FORM>
   </P>
 </body>
 </html>
```

Notice the use of the <template> tag. In between the opening and closing template tags, we have plain vanilla HTML except for the DataBinding code. The DataBinding is handled during the *Page_Load* event, and is very similar to the DataBinding code in the DataList example, the only difference being the name of the Repeater object:

```
Private sConn As String = "<MyConnectionString>"
Private dsShippers As DataSet

    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
        If Not Page.IsPostBack Then
            BindData()
        End If
    End Sub

    Private Sub BindData()
        Dim strSQLQuery As String
        Dim objConn As SqlClient.SqlConnection
        Dim objAdapter As SqlClient.SqlDataAdapter

        strSQLQuery = "SELECT * FROM Shippers"

        objConn = New SqlClient.SqlConnection(sConn)

        objAdapter = New SqlClient.SqlDataAdapter(strSQLQuery,
            objConn)

        dsShippers = New DataSet()
        objAdapter.Fill(dsShippers, "Shippers")

        RPTRShippers.DataSource = _
          dsShippers.Tables("Shippers").DefaultView
        RPTRShippers.DataBind()
    End Sub
```

Here we are creating a DataSet, populating it with some data, and then binding it. To bind data to a container, it must implement the Ilist interface. This can be done by using the DataTable object. Figure 9.11 shows the finished page.

Figure 9.11 Repeater DataControl Sample


Using ADO.NET Repeater Sample - Micro...
File Edit View Favorites Tools Help
Back
Address
Done
Local intranet

Repeater Example

Shippers:

Company Name	Phone
Speedy Express	(503) 555-9831
United Package	(503) 555-3199
Federal Shipping	(503) 555-9931

Summary

In this chapter, we discussed the base functionality in ADO.NET. We discussed XML, and how it relates to ADO.NET. By embracing XML, ADO.NET is breaking new ground and will allow us to develop ever more creative applications, while simplifying the complexities of distributed development. We looked at XML schemas and how ADO.NET uses them to produce Strongly Typed DataSets to speed up development while enforcing the schema definition at compile time.

We emphasized the connectionless architecture on which ADO.NET is built, and provided a solution if we need to maintain a connection to our data source by wrapping the COM+ ADO libraries in .NET and using them as native .NET libraries.

We reviewed the data providers and discussed the benefits of the SQL Server data provider and its close integration with Microsoft SQL Server. We talked about connecting to other data sources using the ADO data provider, and the implications of using the provider to give our applications more options when it comes to connecting to heterogeneous data.

We briefly discussed remoting. This topic is broad enough for a separate book; however, we covered some of the highlights to get you started if that is a direction in which your application needs to go. Remoting allows our application to communicate across application boundaries. Cross-boundary communication can take place between two separate applications on the same machine, or applications on separate servers thousands of miles apart.

Finally, we discussed the data controls and provided some examples to demonstrate the ease of developing using these controls and some of the new functionality that is possible with them.

Solutions Fast Track

Overview of XML

☑ XML documents are the heart of the XML specification.

☑ XSL is a language for applying styles and otherwise formatting XML data.

☑ XDR is a specification for describing the data in XML documents.

☑ XSD is similar to XDR, but provides far more flexibility and is itself an XML document.

☑ XPath is a language for querying XML documents, or describing paths between them.

Understanding ADO.NET Architecture

☑ ADO.NET Architecture is the latest extension of the Universal Data Access technology from Microsoft that uses textual formatting instead of binary formatting.

☑ In ADO.NET Architecture data is accessed and manipulated in a connectionless manner by using data providers.

☑ The Command, Connection, DataReader, and DataAdapter are the core objects in ADO.NET. They form the basis for all operations regarding data in .NET.

Using the XML Schema Definition Tool

☑ XML Schema Definition Tool is used to generate XSD, or schema definition files.

☑ XML Schema Definition Tool is used by Visual Studio when creating TypedDataSets.

Connected Layer

☑ The term connected layer implies that a connection to a data source is open while data is being analyzed or manipulated.

☑ ADO.NET does not support connected data operations.

☑ Persistent connections must be made using the COM Interop modules with the older ADO, and OLE DB Libraries.

☑ Use tlbimp.exe to create .NET wrappers.

Disconnected Layer

☑ By its very nature, ADO.NET is disconnected, which means a connection to the data source is not maintained.

- ☑ A disconnected layer allows our data source to free up resources and respond to the next user.

Using the SQL Server Data Provider

- ☑ The SQL Server data provider is written explicitly to provide data access to Microsoft SQL Server version 7 and later. This set of classes takes advantage of the SQL Server API in a way that makes it more efficient for data access than going through the OLE DB libraries.
- ☑ The SQL Server data provider's core objects are SqlConnection, SqlDataAdapter, SqlCommand, and SqlDataReader.

Remoting

- ☑ *Remoting* allows objects or components to communicate across networks or the Internet and takes care of many of the complexities of communicating across networks.
- ☑ XML is key to remoting in ADO.NET.

Data Controls

- ☑ The data controls in VB.NET are similar to the data controls in VB6. They provide for flexible viewing and editing of data either in the form of bound data or unbound data.
- ☑ Data controls can be manipulated at runtime for flexibility and speed.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: Which object allows for faster reading of data, the DataReader or the DataSet?

A: As always, testing is the final determination, but generally, the DataReader is faster than the DataSet. The DataReader is intended to provide a scrollable source of data that provides access to data one row at a time. If you are returning a great number of rows, then the DataReader may be a better idea than the DataSet. Your testing will determine if the DataSet is better for smaller amounts of data.

Q: Should I use the OLE DB data provider or the SQL data provider?

A: If your project is using SQL Server in production, then by all means use the SQL data provider. The SQL data provider is more efficient and faster than the OLE DB libraries, which is the only advantage that I see. Both objects have the same options and methods; the differences lie in the implementation. The OLE DB data provider will allow you to change the DataSource easily without having to change much code.

Q: Should I use SQL statements or stored procedures for data access?

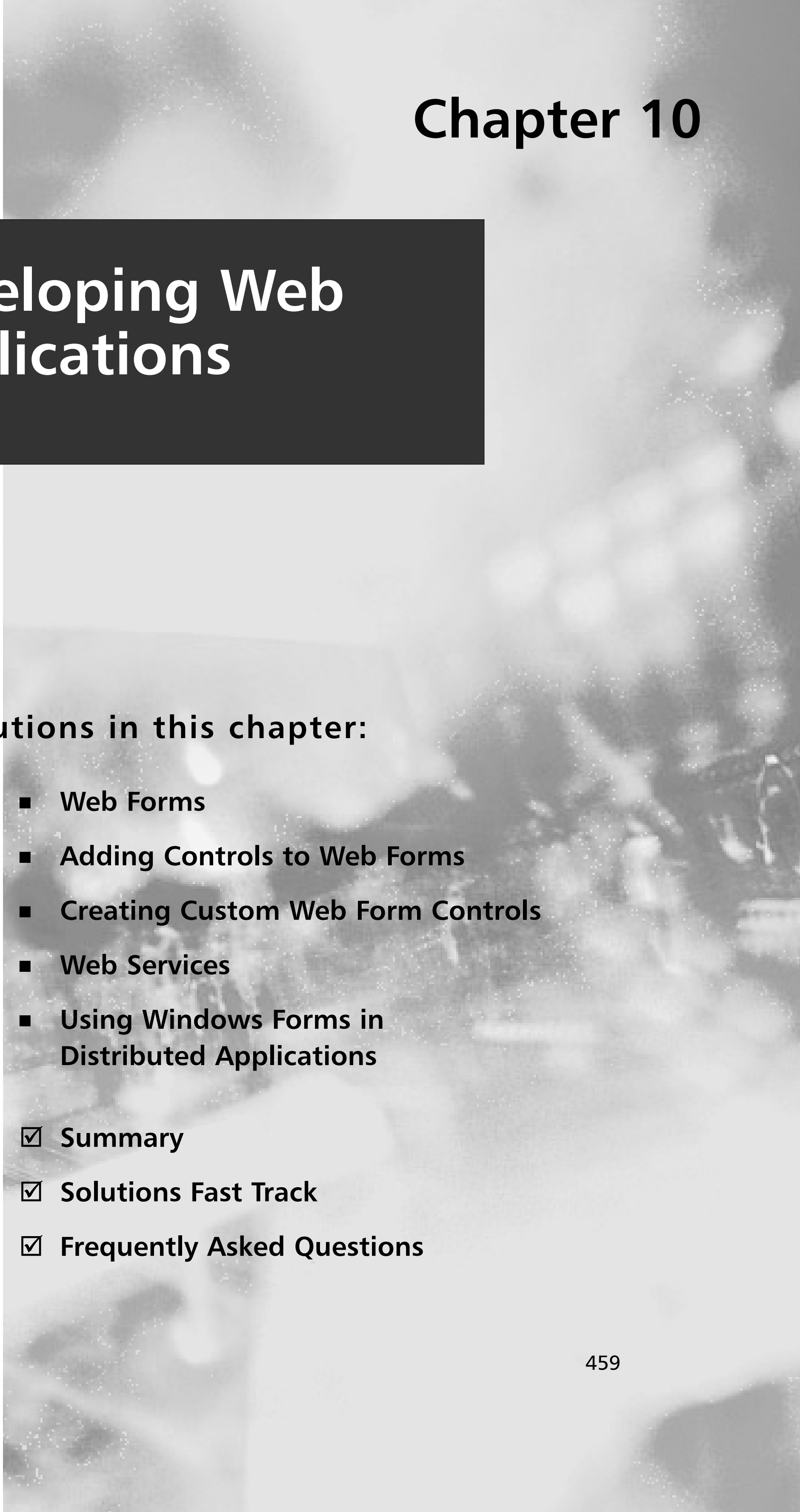
A: Stored procedures are the preferred method of data access, as they allow for another layer of granularity to your application. Most relational databases also precompile, and take the opportunity to optimize the query plan of the stored procedure based on index statistics. They do, however, require other specialized skills that may not be available on your team. In general, I would use SQL statements as a last resort, or in special instances.

Q: When should I use output parameters?

A: Output parameters have less overhead than returning data from a stored procedure. If you are returning a couple of pieces of data, or even an entire row of data, it is more efficient to use the output parameters. It is, however, much more work for both the DBA and the developers. It may come down to your project deadlines, but in general, they are variables in memory that are more efficient than an XML data stream.

Q: When should I use return values from stored procedures?

A: Return values are limited to integer data. For the most part, this is the limiting factor for using return values. As far as ADO.NET is concerned, it is just another output parameter. Return values are useful for returning an error number, informational number, or the identities of new records.



Chapter 10

Developing Web Applications

Solutions in this chapter:

Introduction

In Chapter 8 we learned how to use Windows forms and controls; now it is time to learn about Web forms and controls. This chapter is not meant to teach you how to develop complete Web applications. It is intended to show you how to use Visual Basic .NET as a development tool for Web applications.

Web forms are a part of ASP.NET that allows you to create programmable Web pages. They integrate HTML with server-side and client-side programming logic. For those of you who have developed ASP Web pages, you know the headaches that arise when you try to ensure that your page works correctly on both Microsoft Internet Explorer and Netscape. Web forms automatically determine the client's browser type and create the correct client-side code for that browser. Web forms also give you a richer set of controls for a better user interface.

Web form controls are server-side controls that provide the user interface as well as programmatic functionality. Web form controls are different from Windows controls in that they are designed to work with ASP.NET. A common and tedious task in Web pages is data validation. Web form controls have built-in data validation capabilities that streamline this task. You will find that Web application development is greatly enhanced, with many more features and capabilities than were previously available.

In order to move to the next phase of the Internet, Web applications need to be distributed across the Internet with different functionalities. *Web services* allow this to happen. Web services are object-based components that use XML and HTTP. By allowing communication across the HTTP protocol, you don't have to change your existing network architectures or firewall configurations. Web services are based on open Internet standards that can communicate with components on different platforms and written in different programming languages. This flexibility allows you to communicate with objects through a URL.

Another feature of .NET that further enhances Web applications is the ability to use a Windows form as a client-side user interface in a distributed application. This capability is useful when you require a richer user interface. An example is creating Web forms for most of your users but a thick client for your administrators who would require greater functionality.

We end the chapter with developing a Web application utilizing the features covered in this chapter. This discussion shows how the pieces fit together. To completely cover Web application development with ASP.NET would require a book unto itself.

Web Forms

Web forms extend the Rapid Application Development (RAD) capabilities of Visual Basic to Web applications, allowing developers to create rich, form-based Web pages. Web forms allow users to design Web applications the same way they use Windows forms for Windows applications. Web forms provide a rich user interface, complex layout, and user interaction with no burden on the developer. They also separate the code from the content on a page, eliminating spaghetti code. This feature helps different groups concentrate more on their respective pieces of code. For example, the design group can work on the site using their design tools without having to worry about the code, while the programmers can concentrate on functionality without worrying about the screen design.

Similar to the Windows controls, Web forms have their own controls, called *Web controls*. However, Web controls have more limited capabilities than their counterparts, due to the limitations of the Web model. For example, some of the events found in Windows form controls, such as mouse events, are not practical in the Web model because they require an expensive round trip to the server. Furthermore, these controls aren't ActiveX controls; they exist only on the server and render themselves as standard HTML to the client.

The Integrated Development Environment (IDE) for Web forms is similar to that for Windows forms. You will notice few differences in this environment compared with Windows applications. The design area does not have a form window. It contains a blank Web page, white in the background with two tabs (Design and HTML) at the bottom. The HTML tab allows you to view and edit the HTML code for the page. On the Design tab you have two options: absolute positioning (grid layout) and flow layout, which is the default layout. You can change to the grid layout by using the pageLayout property in the Properties dialog box. The difference between flow layout and grid layout is that in flow layout, the control is dropped where the cursor is currently positioned, whereas in grid layout, the control is placed in the exact X-Y position, similar to Windows forms. It is not a good idea to use grid layout, because your users can use any platform and any screen resolution. In this chapter we use flow layout only.

In this section, we design a simple Web form. We then see how Web forms are different from Windows forms and why they are better than the existing Active Server Pages (ASP). The next section covers Web form controls—the types of controls available and their event model—and finally, we create our own control.

A Simple Web Form

Let's begin our work with a tiny program, the classic "Hello World." In Exercise 10.1, we create a simple Web form. This Web form will have no Web controls. (We discuss Web controls in the next section.) In order to create a Web form, we must first create a Web application.

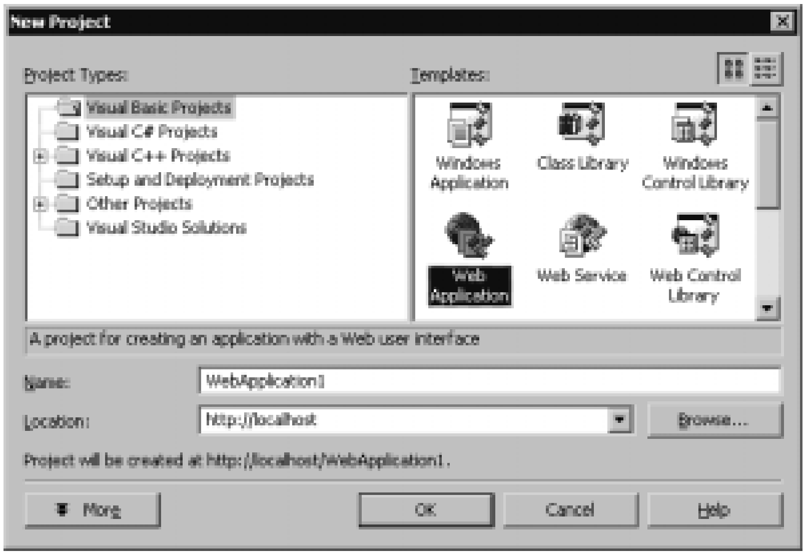
CD Exec. 10.1

Exercise 10.1 Creating a Simple Web Form

In this exercise, we create the standard Hello World example.

1. Begin a new Visual Basic Web project by selecting **File | New | Project** and then selecting **ASP.NET Web Application** under Visual Basic Projects. The New Project dialog box appears, as shown in Figure 10.1.

Figure 10.1 The New Project Dialog Box



A project for creating an application with a Web user interface

Name: WebApplication1

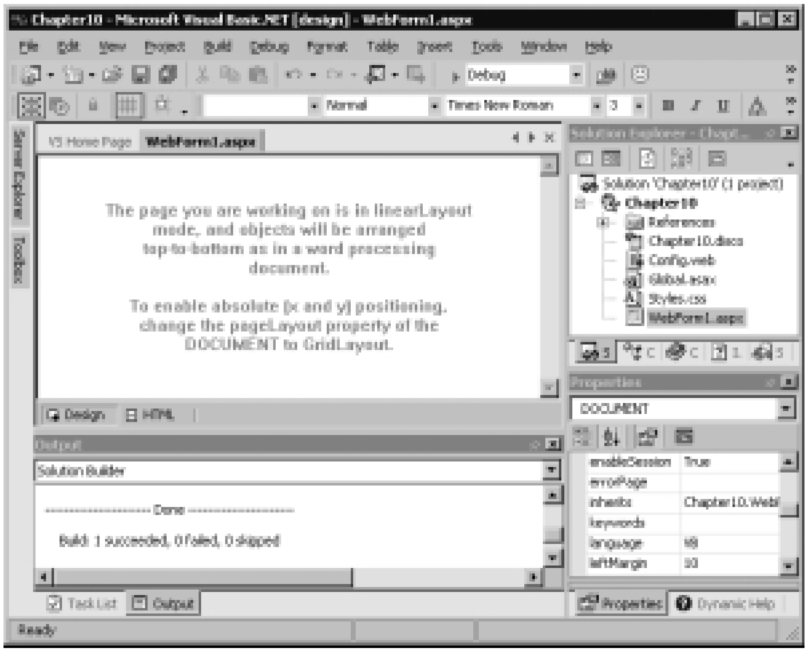
Location: http://localhost

Project will be created at http://localhost/WebApplication1.

2. Change the name of the application to **Chapter10**. If you are using your local machine as the Web server, leave the Location box set to **localhost**; otherwise, enter the name of your Web server in the Location field. Click **OK** and Visual Studio.NET will create the Web project for you, as shown in Figure 10.2.

   If you look at the Solution Explorer of the IDE, you will see that it has more files than a Windows application. All of these files are created under the root directory of your Web site, typically C:\Inetpub\wwwroot\Chapter10 (*Chapter10* is the name of our project). Among these files, WebForm1.aspx (note the new file extension) contains your Web form. Global.asax is similar to Global.asa in the classic ASP, which contains the Application and Session events. Styles.css contains the styles to render HTML. AssemblyInfo.vb contains the assembly information for this project, such as versioning and assembly name. Web.config contains configuration details, and Chapter10.vsdisco contains the discovery information for the Web services. We discuss these two files later in this chapter.

Figure 10.2 Visual Studio.NET IDE for a Web Application



The page you are working on is in linearLayout mode, and objects will be arranged top-to-bottom as in a word processing document.

To enable absolute (x and y) positioning, change the pageLayout property of the DOCUMENT to GridLayout.

```
---------------------- Done ----------------------

    Build: 1 succeeded, 0 failed, 0 skipped
```

enableSession	True
errorPage	
inherits	Chapter10.WebF
keywords	
language	vb
leftMargin	10

3. Using the Properties dialog box, set the pageLayout property to **FlowLayout**.
4. Now click the design surface of the form and type **Hello World**.
5. We have created our first Web form. Before you run the program, you must set the start page for this project. In order to do this, right-click the Web form **WebForm1.aspx** in the Solution Explorer and then click the **Set As Start Page** menu option in the popup menu. Now let's run

the program by pressing **F5** or clicking the **Start** icon in the toolbar. VS.NET builds the application and invokes Internet Explorer. You will see the "Hello World" text in the browser.

Wasn't that easy! You can also run the program by compiling it using **Build | Build menu** (**Ctrl+Shift+B**) once the form is compiled. You can manually open the browser and type the URL. Here the URL of our Web form is http://localhost/chapter10/webform1.aspx. In this exercise, we created a simple Web form. Now let's see the difference between Web forms and Windows forms.

How Web Forms Differ from Windows Forms

You might have observed that Web forms and Windows forms look similar; both can provide a rich user interface and complex application support to fulfill business requirements. The two types of forms might look similar from a design and development point of view, but they differ a great deal in terms of implementation. For example, for an e-commerce application that will be accessed over the Internet on different platforms and browsers, Web forms may be used. When creating a highly responsive system with high-volume transactions for an office application, Windows forms are the best choice. In other words, a Web form is a *thin* client and a Windows form is a *thick* client. In some cases, the choice between Web forms and Windows forms might not be immediately clear. In this section we see how Web forms and Windows forms differ in various situations:

- **User interface** Windows forms can take advantage of .NET WinForms and graphics classes to create rich user interfaces, whereas in Web forms, user interaction requires an expensive round trip to the server, creating a slower response.
- **Security** Windows forms have complete access to local computer resources, whereas browser security limits Web forms.
- **Client platform** All clients in Windows forms require the .NET framework, whereas Web forms require only a browser, so they can target any client platform. The only requirement here is that the Web server should be running .NET Framework.
- **Client application** Windows forms are thick clients, so they rely on the client processor. Web forms usually are thin clients and hence their clients don't have to use high-performance machines.

- **Throughput** Windows forms run on the client side, so they can provide the quickest response and high throughput. Web forms rely on the network traffic over HTTP and hence can't give high throughput and might not be suitable for applications requiring high throughput with high-volume transactions.
- **Deployment** Since Windows forms run on the client machine, they need to be installed on all user desktops. As the users grow in number and with each new release, the deployment becomes tedious. Web forms have no client deployment at all. The client requires only a browser to view them.

You can see that the difference between Web forms and Windows forms is basically the difference between client/server applications and Web applications. Because Web forms use a browser, a universal client, they are excellent for targeting a wide range of clients and for intranet applications.

Why Web Forms Are Better Than Classic ASP

ASP.NET makes it much easier to build enterprise Web applications. ASP.NET is largely syntax compatible with ASP, but under the hood it is completely changed and rewritten to take the advantage of .NET Framework. ASP.NET pages are compiled to CLR. For this reason, you can use any .NET-compatible language: Visual Basic, C#, or Jscript.NET. VBScript is now knocked out, and the friendly and sophisticated Visual Basic is used in its place, thereby using all the features of this language. This change makes it possible for developers to access all the .NET Framework classes in ASP.NET as they would in Visual Basic.

If you have developed any ASP pages, you are very familiar with the major limitations of ASP. The new ASP.NET addresses all of these limitations and provides a much-simplified development environment. Let's see the benefits of ASP.NET over its predecessor:

- **Simplicity** ASP.NET makes it easy to perform common tasks such as form submission, client authentication, site configuration, and deployment.
- **Improved performance** Because ASP.NET is compiled to CLR, it can take advantage of early binding and JIT thus having significant performance over its interpreted predecessor.
- **Strong typed language** ASP.NET now uses Visual Basic as the programming language rather than VBScript, which supports only the

Variant data type. With VB in ASP.NET, we can take advantage of various data types.

- **Event-driven model** ASP.NET supports an event-driven model, just like Visual Basic, thus eliminating the large case statements in the beginning of the page to determine with which button the user has interacted. ASP.NET supports Session and Application events in Global.asax similar to that in ASP. In addition to these four events, Global.asax now supports more than a dozen events.
- **Nonspaghetti code** The programming model of ASP.NET separates the code from the presentation, making constructing and maintenance of code easier.
- **State management** ASP.NET provides easy-to-use Application and Session states that are familiar to ASP developers. In ASP, the Session state resides in the memory of the server, so you can't use it in a Web farm. ASP.NET eliminates this limitation by moving the sessions not only out of process but also out of machine. ASP.NET uses a dedicated state server process that runs as a Windows NT service. This state server listens on a port (default port 42424). This means that you can create a dedicated state server for your Web farm. You can even tell ASP.NET to use SQL Server to store State.
- **Security** ASP.NET provides authorization and authentication services in addition to IIS authentication services. ASP.NET takes the burden from you to authenticate and authorize users stored in a database or in a config file. Users can be authenticated and authorized using CookieAuthenticationModule and URLAuthorizationModule, which sets a cookie that contains the user credentials, and that cookie is checked for subsequent requests.
- **Configuration** ASP.NET uses an XML file to store configuration settings rather than depending on the IIS metabase. This makes the deployment of the site easier, especially in a Web farm.
- **Web services** ASP.NET allows you to expose your business functions to your business partners over standard Web protocols.
- **Cache services** ASP.NET allows you to cache the output of your dynamic page, thus increasing throughput.

- **Debugging** ASP.NET has a built-in tracing utility. You don't have to use Response.Write periodically to trace your program execution. Developers can now use the debugging features they are accustomed to using.
- **Deployment** Deployment is as simple as copying the files. This is because all the configuration settings of the site are in an XML file. Furthermore, it avoids DLL Hell (component registration, versioning, locked DLLs, and so on). You can even recompile a component and deploy it without having to restart the Web server.

At first glance, you might think that all of these benefits will probably make your development harder. Actually, the features of ASP.NET are designed to be easier to use. If you want to take full advantage of ASP.NET, you have to rewrite your current applications. The good news is that ASP and ASP.NET can co-exist on the same machine. In order to accomplish this compatibility, Microsoft introduced new file extensions for ASP.NET (.ASPX, .ASAX, and so on) so you can convert them at your own pace.

Adding Controls to Web Forms

Web form controls are server-side controls that are instantiated on the server and render HTML to the browser. These controls detect the browser and then generate HTML accordingly to provide a customized appearance to the user. Furthermore, these controls expose events similar to the Windows controls, thus working within an event-driven programming model.

Placing the Web form controls on a Web form is similar to placing a Windows control on a Windows form. The only difference is that the layout of the Web form is linear and the controls are dropped where the cursor is currently positioned. If you don't feel comfortable with this setup, you can change the layout to grid layout, which allows you to place the controls on the form similarly to a Windows form.

NOTE

Web form controls not only detect browsers such as Internet Explorer and Netscape, but they also detect devices such as Palm Pilots and cell phones and generate appropriate HTML accordingly.

CD Exec. 10.2

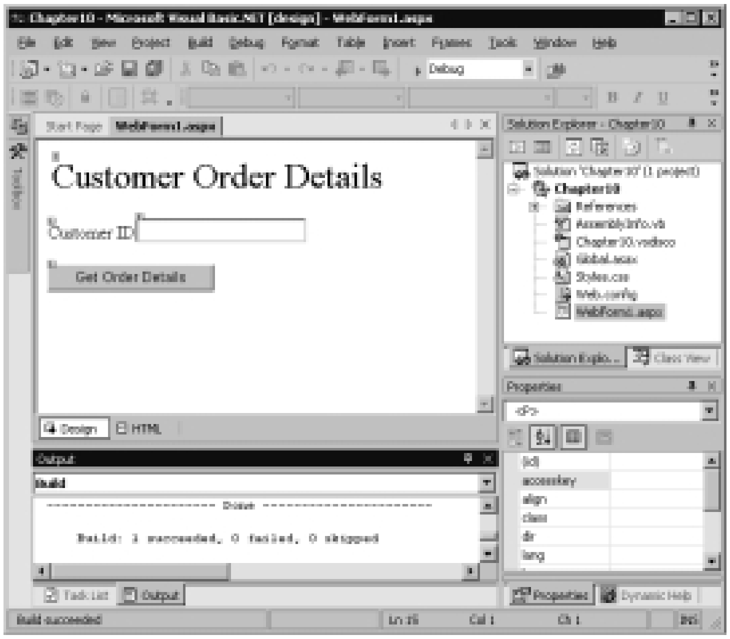
Exercise 10.2 Adding Web Controls to a Web Form

In this exercise, we first create a Web form and place four controls in it. Then we take a close look at the code, and finally we examine the different types of control available. Use the same Web form that we created in Exercise 10.1.

1. Open the Web application project.
2. In the Web form, delete the Hello World text.
3. Place some controls on the form: two label controls, one text box control, and a button control.
4. Place a label control that contains the heading of our application and set the following properties using the Properties dialog box:
   Text: **Customer Order Details**
   Font-Size: **X-Large**
   ID: **lblCustomerOrder**
5. Select the design surface and press **Enter**. Then place another label control on the form by setting the following properties:
   Text: **Customer ID**
   ID: **lblCustomerID**
6. Place a text box control next to the second label control and set its properties to:
   Text: **""** (empty)
   ID: **txtCustomerID**
7. Press **Enter** and then place a button control and change its properties to:
   Text: **Get Order Details**
   ID: **cmdGetDetails**
   After placing the controls, your Web form should resemble Figure 10.3.
8. Save the Form (**Ctrl+S**) and let's test the form we created. Press **F5** to run the application. Visual Studio builds the Web form and invokes Internet Explorer, as shown in Figure 10.4.

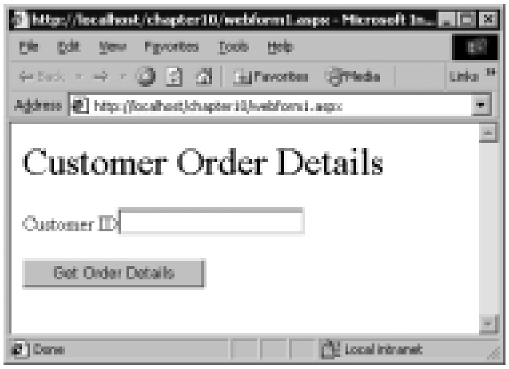
When you click the button, nothing happens. That's because we haven't added any code. Actually, when you click the button, the browser sends a request to the server with the entire user entered data; this process is called *form submission*. You might not notice this process if your local machine is the Web server because the

Figure 10.3 A Web Form with Controls



```
---------------------- Done ----------------------

    Build: 1 succeeded, 0 failed, 0 skipped
```

Figure 10.4 Viewing Web Form Controls Using a Browser


http://localhost/chapter10/webform1.aspx - Microsoft In...
File Edit View Favorites Tools Help
Address http://localhost/chapter10/webform1.aspx
Customer Order Details
Customer ID
Get Order Details
Done
Local intranet

response is quick. Try entering something in the text box and click the button; you can see that the text you entered persists in the form submission. Web forms do this for us without writing any code. In classic ASP, user-entered data does not persist in form submission and we have to write code to do this.

Now let's see the code of the Web form. In the VS.NET IDE, click the **HTML** button below the design surface to view the HTML code. The Web form code generated by IDE will look familiar to an ASP developer. Here is the code generated by IDE:

```
<%@ Page Language="vb" AutoEventWireup="false"
          Codebehind="WebForm1.aspx.vb"
              Inherits="Chapter10.WebForm1"%>
<html>
<head>
    <meta name="GENERATOR" content="Microsoft Visual Studio.NET
        7.0">
    <meta name="CODE_LANGUAGE" content="Visual Basic 7.0">
</head>
<body>
    <form id="WebForm1" method="post" runat="server">
    <p>
      <asp:Label id=lblCustomerOrder runat="server" Font-Size="X-
          Large">
              Customer Order Details
      </asp:Label>
    </p>
    <p>
      <asp:Label id=lblCustomerID  runat="server">
              Customer ID
      </asp:Label>
      <asp:TextBox id=txtCustomerID runat="server"></asp:TextBox>
    </p>
    <p>
      <asp:Button id=cmdGetDetails runat="server"
                    Text="Get Order Details"></asp:Button>
    </p>
    </form>
</body>
</html>
```

Let's go through the code. Ignore the following first line for now; we will discuss it later:

```
<%@ Page Language="vb" AutoEventWireup="false"
          Codebehind="WebForm1.aspx.vb"
              Inherits="Chapter10.WebForm1"%>
```

The next few lines contain the header information and metatags of the HTML page. After that is the form tag. Observe the *runat* clause at the end:

```
<form id="WebForm1" method="post" runat="server">
```

The *runat* clause tells .NET that this is a Web control and will expose the form events. You might have noticed that the *action* attribute is missing here. If the *action* attribute is missing, the form posts to itself—that is, to the same page. So, in ASP.NET, all the forms post to themselves. Now let's see the code for the controls you have placed in the form. First observe the heading label:

```
<asp:Label id=lblCustomerOrder runat="server" Font-Size="X-
    Large">
     Customer Order Details
</asp:Label>
```

asp:Label says that it is a label control in the ASP namespace. All the controls available in ASP.NET are under the ASP namespace and have a *runat* clause. Unlike HTML, ASP.NET is strict with closing tags. All the opened tags must be closed; otherwise, ASP.NET generates a compile error. As mentioned earlier, all the Web form controls render themselves as HTML to the client. So if you view the HTML source in the browser, you will see pure HTML code. Let's see the HTML code returned by the Web server:

```
<html>
<head>
    <meta name="GENERATOR" content="Microsoft Visual Studio.NET
         7.0">
    <meta name="CODE_LANGUAGE" content="Visual Basic 7.0">
</head>
<body>
    <form name="WebForm1" method="post"
                        action="Webform1.aspx" id="WebForm1">
       <input type="hidden" name="__VIEWSTATE"
```

```
                  value="YTB6MTk1NDExNjQxNl9fX3g=e92c2469" />
    <p>
       <span id="lblCustomerOrder" style="font-size:X-Large;">
            Customer Order Details
       </span>
    </p>
    <p>
     <span id="lblCustomerID ">Customer ID</span>
     <input name="txtCustomerID" type="text" id="txtCustomerID"/>
    </p>
    <p>
       <input type="submit" name="cmdGetDetails"
                   value="Get Order Details" id="cmdGetDetails"/>
    </p>
    </form>
</body>
</html>
```

Notice the form tag generated by the Web form. This tag has an *action* attribute and its value is the current name of the page, thus posting to itself. Under the form tag, observe the hidden field:

```
<input type="hidden" name="__VIEWSTATE"
             value="YTB6MTk1NDExNjQxNl9fX3g=e92c2469" />
```

ASP.NET uses this field to maintain the state across the page submission. As you'll recall, the text you entered in the TextBox persisted after the page submission. You can use this state management to store values similar to the Session variables. Classic ASP uses Session variables to store State across pages. (A detailed discussion of this topic is beyond the scope of this chapter.) All the label controls placed in the form rendered themselves to a *span* tag. TextBox and button controls rendered into the HTML input tag. Web forms generate HTML, so they can be viewed in any browser.

Now let's write some code. If the button is clicked, we set the text in the TextBox to **Button is clicked**. In order to write code for the *Click* event, in the VS.NET IDE design, double-click the button control to open the code window. Observe that a new window is opened and your cursor is positioned inside the cmdGetDetails_Click function. For now, ignore the rest of the code and write the following code to set the text of the TextBox:

```
txtCustomerID.Text = "Button is clicked"
```

Compile the program, run it in the browser, and click the button. You see that the text in the TextBox is changed to what we have entered in the code.

Code Behind

Before going into the VB code, let's revisit the first line in the Web form:

```
<%@ Page Language="vb" AutoEventWireup="false"
           Codebehind="WebForm1.aspx.vb"
 Inherits="Chapter10.WebForm1"%>
```

This is called a *page directive,* and it tells the compiler that the language used in this page is Visual Basic, its code is found in the file WebForm1.aspx.vb under the same folder, and this page is inherited from Chapter10.WebForm1. If you see files in the physical directory under the Web server, you will notice a Visual Basic file called WebForm1.aspx.vb. This file is named the same as the filename of the Web form with a VB extension. When you compile your Web form, Visual Studio actually compiles this file into an assembly (.DLL) under the bin directory of the Web application on the Web server. If you check the bin directory of your application on your Web server, you will see the DLL. In our case, you will find Chapter10.dll under C:\Inetpub\wwwroot\Chapter10\bin. This DLL contains the namespace Chapter10 (the name of our Web application) and all the Web forms in our application as classes inside this namespace. This structure restricts the usage of the same filename for the Web form, even though the files are in different folders. This technique is called *code behind* because code is behind the form.

Now let's see the Visual Basic code for this form. The first few lines import different namespaces commonly used in the ASP.NET. Of these namespaces, we require System.Web.UI and System.Web.UI.WebControls because they contain the page and Web control APIs:

```
Imports System
Imports System.ComponentModel.Design
Imports System.Data
Imports System.Drawing
Imports System.Web
Imports System.Web.SessionState
Imports System.Web.UI
Imports System.Web.UI.WebControls
Imports System.Web.UI.HtmlControls
Imports Microsoft.VisualBasic
```

After the imported namespaces, there is a declaration of class for our Web form; this class must inherit from the Page object, thereby exposing the Page properties and events. Then we have declarations of all the controls placed on the form with the WithEvents clause so that they can expose their events:

```
Public Class WebForm1
Inherits System.Web.UI.Page
Protected WithEvents cmdGetDetails As
    System.Web.UI.WebControls.Button
Protected WithEvents txtCustomerID As
    System.Web.UI.WebControls.TextBox
Protected WithEvents lblCustomerID As
    System.Web.UI.WebControls.Label
Protected WithEvents lblCustomerOrder As

    System.Web.UI.WebControls.Label
```

Observe the hidden region. (This is similar to what you see in Windows form code.) The hidden region is where Visual Studio places the generated code. You shouldn't modify this code:

```
#Region " Web forms Designer Generated Code "
    'This call is required by the Web form Designer.
    <System.Diagnostics.DebuggerStepThrough()> _
       Private Sub InitializeComponent()

    End Sub

    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 Designer
        'Do not modify it using the code editor.
        InitializeComponent()
    End Sub
#End Region
```

After this hidden region, we have all the events exposed by the Web form. Here you see the code we wrote for the *Button Click* event. The *WebForm1_Load* event is similar to the form *Load* event, and the *WebForm1_Init* event fires when the page is initialized. This is the first event in the page's life cycle; after that, the load event triggers and is followed by events of the controls in the form. Other events such as *PreRender* and *UnLoad* trigger in the form life cycle. Apart from these events, more than a dozen events in Global.asax trigger for every request to a Web form. Covering them is beyond the scope of this chapter:

```
    Private Sub cmdGetDetails_Click(ByVal sender As

        System.Object, _
            ByVal e As System.EventArgs) Handles
                cmdGetDetails.Click
        txtCustomerID.Text = "Button is clicked"
    End Sub
    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
```

Similar to the events in the Windows forms, every event delegate in Web forms has two arguments: the sender of the event (in the *Button Click* event, the sender is the button) and an instance of the class that holds data for the event. The Handles keyword attaches this procedure to the object declared using the WithEvents keyword. Because the sender contains the object that triggered the event, it is used to get or set the object properties. The second parameter is used to retrieve other data associated with the event—for example, an *ImageButton Control Click* event contains X and Y coordinates where a user clicked.

So far, we have seen a Web form with some Web controls. In the next section, we look at various types of control available in Web forms. Before that, we see how Web form controls differ from Windows form controls.

Developing & Deploying…

Web Form Deployment

Deploying a Web form involves simply copying the .ASPX file and DLL in the bin directory to their destinations. The DLL file should always be in the bin directory.

How Web Form Controls Differ from Windows Form Controls

Web form controls are similar to Windows form controls. They both have event-driven programming models. The main difference between them is that Web form controls generate HTML to the user, whereas Windows controls take advantage of rich user features available in the Microsoft Windows operating system.

In Windows form controls, events are raised and handled on the client side only, whereas in Web form controls, the events are raised on the client browser but they are handled on the Web server. Because of this model, Web form controls have fewer events than their counterparts.

ASP.NET Server Controls

Web form server controls are broadly classified into four categories: intrinsic controls, bound controls, custom controls, and validation controls. *Intrinsic controls* are those controls that can render themselves to equivalent HTML elements. *Bound controls* help us lay out a page in a grid or list format with data returned from a database. *Custom controls* are the rich controls that are not available in HTML, such as the calendar control. Finally, Microsoft provided *validation controls* to validate user inputs. The following sections examine each control type in detail.

Intrinsic Controls

In Exercise 10.2, we used three controls: label, text box, and button. All three controls rendered HTML to the client, so these controls are intrinsic controls. There are several other intrinsic controls besides these. For those of you who have worked with ASP previously, you know that for a text box you would write HTML as follows:

```
<input type="text" name="txtCustomerID" id="txtCustomerID">
```

In the previous exercise, you noticed that the equivalent of this code in ASP.NET controls is:

```
<asp:TextBox id=txtCustomerID runat="Server">
```

There is another way of creating a server control, simply add a *runat* clause to the HTML:

```
<input type="text" name="txtCustomerID"
             id="txtCustomerID" runat="server">
```

Adding the *runat* clause to the HTML element makes this a server control. ASP.NET exposes all of the events of this control. These controls are called *HTML server controls*. All of the HTML server controls are derived from the namespace System.Web.UI.HtmlControls. ASP server controls and HTML server controls behave in exactly the same manner. The main difference between these controls is that ASP Server controls detect the browser and generate the appropriate HTML (HTML 3.2 or HTML 4.0), whereas with HTML controls, you have to write code for browser detection if you want to take advantage of the upper-level browser. For this reason, ASP.NET server controls have a richer object model than HTML server controls. In this chapter, we use ASP server controls only. Table 10.1 shows the various ASP.NET intrinsic controls and their HTML equivalents.

Table 10.1 ASP.NET Controls and Their Corresponding HTML Elements

Intrinsic Controls	HTML Element
Label	<span> … </span>
TextBox	<input type="text">
Button	<input type="Submit">
LinkButton	<a href="javascript:form.submit()">
ImageButton	<input type="image">
HyperLink	<a href="">
DropDownList	<select> .. </select>
ListBox	<select size=""> .. </select>
CheckBox	<input type="checkbox">
RadioButton	<input type="radio">
Image	<img>
Panel	<div> ..</div>
Table	<table>… </table>

All intrinsic controls run at the server level and render HTML elements. Among these intrinsic controls, some of the controls such as DropDownList and ListBox controls can be bound to a dataset from the database. We discuss how to bind a dataset to the control in the next section.

Bound Controls

In a Web page, we usually retrieve data from the database and show it to the user in a tabular form or in a list form. Microsoft introduced three bound controls that do the work for us; they are DataGrid control, Repeater control, and DataList control. The DataGrid is the richest bound control and the easiest way to display the data in a grid control. In a Repeater control, you define the layout of individual items. When the page is run, the control repeats the layout for each item in the data source. DataList is similar to the Repeater control and provides more formatting options.

DataGrid

As mentioned earlier, DataGrid allows us to display the data returned from a data source in a tabular form. This is the most commonly used control. The control has numerous customization options. First let's see a simple DataGrid in action, and then we can apply some customizations.

CD Exec. 10.3

Exercise 10.3 Using the DataGrid Control

Use the same project that we used for Exercise 10.2. In that exercise, we placed four controls on a Web form (refer back to Figures 10.3 and 10.4). Once the user enters the Customer ID and clicks the button, it shows the orders placed by the customer. Typically, orders are shown in a tabular form, which is where DataGrid comes into the picture.

For this exercise, we use the NorthWind database, which comes with default installation of the SQL Server 2000. In the NorthWind database, the Orders table contains all the orders placed by the customers. We will use this table to get the orders placed by the user-supplied Customer ID.

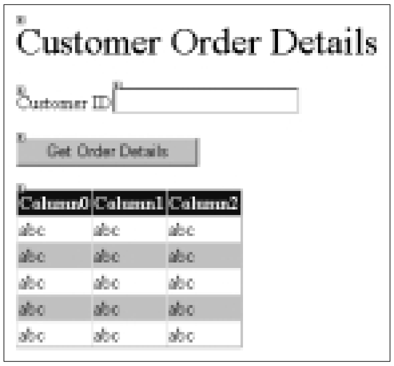
1. Open the Chapter10 Web application project.
2. Place a DataGrid control on the Web form by dragging and dropping it from the toolbox and set the following properties:
   ID: **dgOrders**
   HeaderStyle-BackColor: **Navy**
   HeaderStyle-Font-Bod: **True**

HeaderStyle-ForeColor: **White**

AlternatingItemStyle-BackColor: **Silver**

After placing the DataGrid control in the Web form, your design area should appear as shown in Figure 10.5.

Figure 10.5 DataGrid Control in Design Time



For this design, the user enters a Customer ID and clicks the Get Order Details button. So, for the *Button Click* event, we need to write code to get the orders placed by the customer. In order to reuse the code, we create a function that will return the dataset containing the orders placed by the customer for a given Customer ID.

3. Add the following code to your page in WebForm1.vb:

```
Public Function GetOrders(ByVal CustomerID As String) As
    DataSet
    Dim sConnectionString As String
    Dim sqlString As String
    Dim MyConnection As SqlConnection
    Dim MyDataAdapter As SqlDataAdapter
    Dim DS As New DataSet()
    'building the connection string
    sConnectionString = "Server=localhost;
        Database=Northwind; "
    sConnectionString += "UID=sa; pwd=;"
```

```
        'building the select statement
        sqlString = "SELECT * FROM Orders "
        sqlString += "WHERE CustomerID = '" + CustomerID + "'"

        'opening the connection
        MyConnection = New SqlConnection(sConnectionString)
        MyDataAdapter = New SqlDataAdapter(sqlString,
            MyConnection)

        'getting Data Set
        MyDataAdapter.Fill(DS, "Orders")

        Return DS
    End Function
```

Because you saw in the previous chapter how to access data from a database, we don't delve into this code here. You might have to change the username and password, depending on your SQL Server setup. Here we use SqlConnection and SqlDataAdapter instead of OleDBConnection and OleDBDataAdapter for data retrieval. SqlConnection and SqlDataSetCommand are part of the ADO.NET and are optimized for SQL Server to provide more functionality and faster access than a generic managed provider. In order to use SQL APIs in our code, we have to import that namespace.

4. To import the SQL namespace, add the import statement to the code:

```
'other import statements
Imports System.Data.SqlClient
```

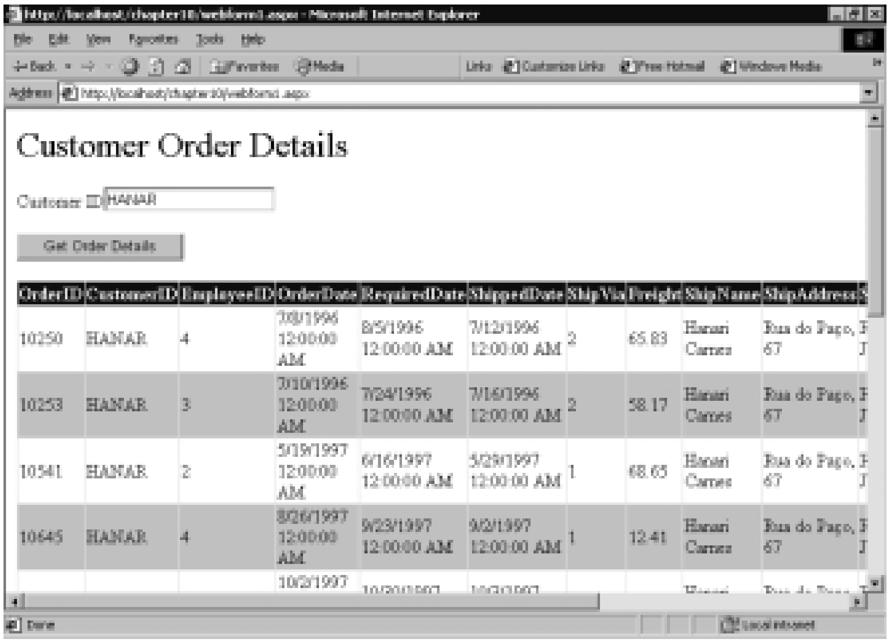
5. Place the following code in the *Button Click* event:

```
Dim DS As DataSet
'getting the DataSet with Order Details for the entered
    CustomerID
DS = GetOrders(txtCustomerID.Text)
'Binding the DataGrid
dgOrders.DataSource = DS.Tables("Orders").DefaultView
dgOrders.DataBind()
```

This code first defines a variable of type *DataSet* and then calls the function, *GetOrders*, that we wrote previously to get the dataset for the Customer ID entered by the user. After *DataSet* is returned by the function into our dataset variable (DS), the code binds this dataset to the DataGrid. Because the *DataSet* can contain multiple tables, we had to set the *DataView* of the table we want, which in our case is the Orders table, to the *DataSource* property of the DataGrid and then call the *DataBind* method to finally bind the data to the DataGrid.

6. Save the form and run the application.
7. In the browser, enter the Customer ID **HANAR** or any other valid Customer ID from the database and click the button. Once you click the button, you see all the orders for the customer HANAR. Figure 10.6 shows the output of the DataGrid.

Figure 10.6 Viewing DataGrid Control in a Browser


File Edit View Favorites Tools Help
Back Favorites Media
Links Customize Links Free Hotmail Windows Media
Address http://localhost/chapter10/webform1.aspx
Get Order Details
Done
Local intranet

Customer Order Details

Customer ID HANAR

OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	ShippedDate	ShipVia	Freight	ShipName	ShipAddress
10250	HANAR	4	7/8/1996 12:00:00 AM	8/5/1996 12:00:00 AM	7/12/1996 12:00:00 AM	2	65.83	Hanari Carnes	Rua do Paço, 67
10253	HANAR	3	7/10/1996 12:00:00 AM	7/24/1996 12:00:00 AM	7/16/1996 12:00:00 AM	2	58.17	Hanari Carnes	Rua do Paço, 67
10541	HANAR	2	5/19/1997 12:00:00 AM	6/16/1997 12:00:00 AM	5/29/1997 12:00:00 AM	1	68.65	Hanari Carnes	Rua do Paço, 67
10645	HANAR	4	8/26/1997 12:00:00 AM	9/23/1997 12:00:00 AM	9/2/1997 12:00:00 AM	1	12.41	Hanari Carnes	Rua do Paço, 67
			10/2/1997						

If you look at the HTML source of the page, you will notice that the DataGrid control generates HTML. Now let's take a look at the Web form syntax of the DataGrid control:

```
<asp:DataGrid id=dgOrders runat="server" ForeColor="Black"
```

```
                Width="188" Height="114" >
        <HeaderStyle Font-Bold="True" ForeColor="White"
      BackColor="Navy">
        </HeaderStyle>

        <AlternatingItemStyle BackColor="Silver">
        </AlternatingItemStyle>
</asp:DataGrid>
```

asp:DataGrid is the DataGrid control in the ASP namespace. In the control tag, you can see the Width and Height properties. These are used to set the width and height of the grid. During runtime, the control dynamically expands to generate a HTML table. Inside the control tag you can see the HeaderStyle and AlternatingItemStyle tags. These are the properties of the DataGrid we set using the Properties dialog box. The *HeaderStyle* property tag is used to set the style of the table Header, such as font and color. Similarly, the *AlternatingItemStyle* property sets style on the alternate column of the data row. There are other properties, such as *FooterStyle,* which is used to set the style in table footer, and *ItemStyle,* which is used to set the style of the data rows. That is the reason our grid output alternating colors for the data.

CD Exec. 10.4

Exercise 10.4 Customizing DataGrid Control

DataGrid has numerous customization options. Some of these customizations are not available through the Properties dialog box; instead, they can be handled by modifying the Web form code. In the previous exercise, the DataGrid control output all the columns from the database using the table column name as the header. Instead, let's show only four columns, OrderID, OrderDate, ShippedDate, and ShipName, and change the column headings. One way to do this is to change the *Select* statement to retrieve only the required fields from the database. We won't use this method, though; instead, we will customize the DataGrid control to show only the required columns. The DataGrid control has another property tag called Columns. We will see how to use this property tag in this exercise.

1. Open the Chapter10 Web application project from Exercise 10.3 if it is closed.
2. Place the following code between the DataGrid tags. This code allows us to show only the columns we want:

```
<Columns>
   <asp:BoundColumn datafield="OrderID" headertext="Order ID"/>
   <asp:BoundColumn datafield="OrderDate" headertext="Order
       Date"/>
   <asp:BoundColumn datafield="ShippedDate" headertext="Shipped
       Date"/>
   <asp:BoundColumn datafield="ShipName" headertext="Ship Name"/>
</Columns>
```

By explicitly creating the BoundColumn control inside the DataGrid's Column collection, we can control the order of each column and can show only the columns we want. Among the attributes of the BoundColumn control, *datafield* represents the data column in the DataSet and *headertext* represents the column heading when showing on a browser. After adding this code to the DataGrid control tag, we have to tell the DataGrid control to use this property and generate only the columns we want.

3. Set the *autogeneratecolumns* attribute of the DataGrid control tag to **False**, which forces DataGrid to use only the columns defined in the Column collection. After making these customizations, your DataGrid control code should be as follows:

```
<asp:DataGrid id=dgOrders runat="server" ForeColor="Black"
                 Width="188" Height="114"
    autogeneratecolumns="False">
   <HeaderStyle Font-Bold="True" ForeColor="White"
    BackColor="Navy">
   </HeaderStyle>

   <AlternatingItemStyle BackColor="Silver">
   </AlternatingItemStyle>

   <Columns>
      <asp:BoundColumn datafield="OrderID" headertext="Order
          ID"/>
      <asp:BoundColumn datafield="OrderDate" headertext="Order
```

```
            Date"/>
        <asp:BoundColumn datafield="ShippedDate"
                         headertext="Shipped Date"/>
        <asp:BoundColumn datafield="ShipName" headertext="Ship
            Name"/>
    </Columns>
</asp:DataGrid>
```

4. Compile the application and run it in the browser. Enter the Customer ID (**HANAR**) and click the **Get Order Details** button. The grid control will output only the columns we asked for.

   Observe the output, see that the columns Order Date and Shipped Date include the time. Here the time is always midnight. Let's apply one more customization to the DataGrid control to format the date columns. Like the BoundColumn control, DataGrid supports the TemplateColumn control via which you can specify the content of the output.
5. To format the order date, we have to replace the OrderDate BoundColumn control with this TemplateColumn code:

```
<asp:templatecolumn  headertext="Order Date">
  <ItemTemplate>
       <%# String.Format("{0:d}",
           Container.DataItem("OrderDate")) %>
  </ItemTemplate>
</asp:templatecolumn>
```

   In this code, the attribute *headertext* pertains to the column header. Because we are using TemplateColumn instead of BoundColumn, we have to specify the column content and bind this column with the DataSet column manually. BoundColumn did this automatically for us, so the syntax to bind a column in the DataSet to the DataGrid controls is as follows:

```
<%# String.Format("{0:d}", Container.DataItem("OrderDate") ) %>
```

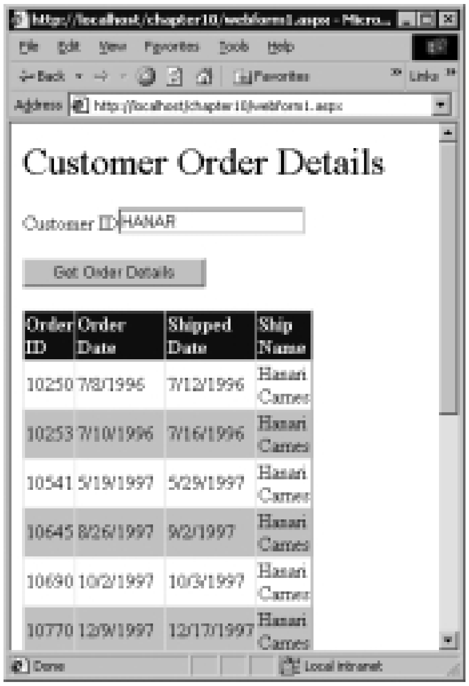
   This line might seem familiar to you if you worked with ASP previously. This line not only binds the OrderDate column, it also formats the column to *dd/mm/yyyy* format.

6. Similarly, to format the ShippedDate column, replace the BoundColumn control for ShippedDate with the following:

```
<asp:templatecolumn  headertext="Shipped Date">
  <ItemTemplate>
      <%# String.Format("{0:d}",
          Container.DataItem("ShippedDate") ) %>
  </ItemTemplate>
</asp:templatecolumn>
```

7. Compile the program and in the browser enter the Customer ID **HANAR**. After pressing the button, you will see the output nicely formatted with only the columns we want. The output is shown in Figure 10.7.

Figure 10.7 Customized DataGrid Control


http://localhost/chapter10/webform1.aspx - Micro...
File Edit View Favorites Tools Help
Back Favorites Links
Address http://localhost/chapter10/webform1.aspx
Get Order Details
Done
Local intranet

Customer Order Details

Customer ID HANAR

Order ID	Order Date	Shipped Date	Ship Name
10250	7/8/1996	7/12/1996	Hanari Carnes
10253	7/10/1996	7/16/1996	Hanari Carnes
10541	5/19/1997	5/29/1997	Hanari Carnes
10645	8/26/1997	9/2/1997	Hanari Carnes
10690	10/2/1997	10/3/1997	Hanari Carnes
10770	12/9/1997	12/17/1997	Hanari Carnes

So far, we have applied customizations to narrow the fields we want to show and format the fields using TemplateColumn. The complete code of the ASP.NET DataGrid control after the customizations is as follows:

```
<asp:DataGrid id=dgOrders runat="server" ForeColor="Black"
             Width="188" Height="114"
                 autogeneratecolumns="False">
   <HeaderStyle Font-Bold="True" ForeColor="White"
       BackColor="Navy">
   </HeaderStyle>

   <AlternatingItemStyle BackColor="Silver">
   </AlternatingItemStyle>

   <Columns>
    <asp:BoundColumn datafield="OrderID" headertext="Order ID"/>
    <asp:templatecolumn  headertext="Order Date">
     <ItemTemplate>
      <%# String.Format("{0:d}",
          Container.DataItem("OrderDate")) %>
     </ItemTemplate>
    </asp:templatecolumn>
    <asp:templatecolumn  headertext="Shipped Date">
     <ItemTemplate>
      <%# String.Format("{0:d}",
          Container.DataItem("ShippedDate") ) %>
     </ItemTemplate>
    </asp:templatecolumn>
    <asp:BoundColumn datafield="ShipName" headertext="Ship
       Name"/>
   </Columns>
</asp:DataGrid>
```

The DataGrid control also supports another useful tool: paging. Paging allows you to set the number of rows per page and navigate from one page to another with little code. It also allows editing data in a column, deleting the row, and sorting.

Web.config

In the function that we wrote to access the database and retrieve a customer's orders in a dataset, we hardcoded the connection string:

```
sConnectionString = "Server=localhost; Database=Northwind;"
sConnectionString += "UID=sa; pwd=;"
```

ASP.NET gives us the ability to store all the configurations of the application in an XML file. In the Solution Explorer of our project, open the file Web.config. In this file we can store our application settings and then retrieve them wherever we want. Let's store the connection string to the database in this file and then retrieve it in our function. As mentioned earlier, Web.config uses XML format, so we add the following XML string to the end of this file:

```
<appsettings>
    <add key="DSN"
         value="Server=localhost; Database=Northwind; UID=sa;
             pwd=;" />
</appsettings>
```

In order to retrieve the configuration settings, you must use the *GetConfig* method of the System.Web.HttpContext. The *GetConfig* method returns a hashtable similar to the Dictionary object. The following code illustrates how to access the configuration data:

```
sConnectionString = Context.GetConfig("appSettings")("DSN")
```

So, we can use Web.config for storing application settings. Additionally, Web.config is used to store all the configuration settings of the application, such as session management, tracing options, security, references to assemblies, and Web services. The settings that we specify in Web.config override the default configuration settings. You can find the default Web.config settings under C:\WINNT\Microsoft.NET\Framework\version\CONFIG\Machine.config (here *version* stands for the current version of .NET Framework).

Custom Controls

ASP.NET ships with certain custom controls to add more functionality to the Web forms. These custom controls facilitate the creation of rich user interfaces. Beta 2 ASP.NET is shipped with the Calendar control and AdRotator control. Microsoft is planning to provide more controls when it officially launches the application. Of these two controls, the Calendar control is used to display a one-month calendar on the Web form; the user can use this control to navigate to any date in any year and click to select a date. AdRotator is used to display ad banners on Web pages, similar to the AdRotator server component in ASP. It changes the displayed ad automatically each time the Web page loads.

Developing & Deploying…

Configuration Hierarchy

The Web.config file can exist in any directory in the ASP.NET application. If it exists, its settings are applied to the current directory and to all child directories. It also overrides the parent directory configuration settings.

Validation Controls

Validating user input is the boon of every program. In order to make our life easy, Microsoft introduced six validation controls. These controls help developers validate all common types of data. Furthermore, they provide a way to validate data using custom-written validation routines. Using these controls is easy and, in most cases, you don't have to write any code. The six available validation controls are:

- **RequiredFieldValidator** Ensures that user does not skip any required entry.
- **RegularExpressionValidator** Validates the user entry with a pattern defined using regular expressions.
- **CustomValidator** Checks the user's entry using validation logic we have written. This can be either a server-side script or a client-side script.
- **CompareValidator** Compares the user's entry with a constant value or with that of another control.
- **RangeValidator** Validates that user input should be in a specified range.
- **ValidationSummary** This control displays all the validation errors in a summary form or in a list form.

Each of these controls can be attached to ASP.NET server controls (input controls). You can also attach multiple validation controls to an input control. Once these controls are attached to the input control, .NET Framework validates the user inputs and, if there is an error, displays the error message on the page. Furthermore, it sets the *IsValid* property of the Page object, which can be used to check the page's validity.

Among these validation controls, the RegularExpressionValidator control is used to validate the user inputs with a predefined regular expression pattern. *Regular expressions* are a powerful tool for searching and processing text. A regular expression is a series of characters that define a pattern, and a pattern defines the criteria to search for within a string. The pattern for all capital letters is [A–Z]. If you want the user to enter at least five capital letters, your pattern will be [A–Z]{5}. The 5 inside the braces forces the user to enter at least five consecutive capital letters. If you want the user to enter *only* five capital letters, you have to add a caret (^) at the beginning and a dollar sign ($) at the end. These symbols force the target string to match the pattern at both the beginning and the end. Therefore, the pattern for only five capital letters will be ^[A–Z]{5}$. This is only a small sample of the power that expressions can provide. Regular expressions are a language all by themselves, and you can find entire books on this topic.

Exercise 10.5 Using the Validation Controls

Let's use some of the validation controls to validate the user input in our example. Customer ID is a mandatory field that the user must supply. To validate this, we use RequiredFieldValidator.

1. Open the Chapter10 Web application project from Exercise 10.4.
2. Place the RequiredFieldValidator control next to the text box and set the following properties using the Properties dialog box:

   ID: **RequiredFieldValidator**

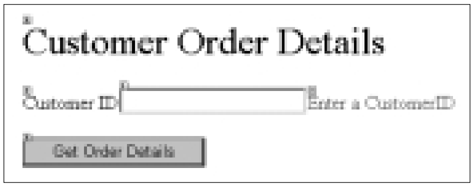
   ErrorMessage: **Enter a CustomerID**

   ControlToValidate: **txtCustomerID** (pick from the list)

   Display: **Dynamic** (*static* means the control reserves the space in the form even though there is no error; *dynamic* doesn't reserve the space beforehand)

   After setting the properties to the control, your design area should resemble Figure 10.8.

Figure 10.8 RequiredField Validation Control in Design Time



3. On the button's *Click* event, we should get data for the Customer ID only when the page is valid. So on the *Click* event we check whether the page is valid. Add the following code at the beginning of the *Button Click* event:

   ```
   If Not Page.IsValid Then Exit Sub
   ```

4. Compile the program and run it in the browser. If you don't enter anything and click the button, you will see an error message. If you are using IE or another upper-level browser, the control detects the browser and uses DHTML to show the error message.

   The RequiredFieldValidator control helps us a great deal in doing validations. RequiredFieldValidator control forces the user not to skip the text box entry. However, we want more than that; we want to validate what the user entered. To do this we need to use the RegularExpressionValidator control. As you'll observe, all the Customer IDs in the database are alphabetic capital letters and are five characters long. As explained earlier, the validation expression to match this pattern is ^[A–Z]{5}$.

5. Place the RegularExpressionValidator immediately next to the RequiredFieldValidator and set the following properties using the Properties dialog box:

   ID: **RegularExpressionValidator**
   ErrorMessage: **Enter a valid CustomerID**
   ControlToValidate: **txtCustomerID** (pick from the list)
   Display: **Dynamic**
   Validation Expression: **^[A-Z]{5}$**

6. Save the program and run it. Enter some junk value and click the button. You will get the error message, "Enter a valid CustomerID," which is the error message we set in the control.

We are able to force the user into entering something in the text box while validating the text the user has entered to be of a particular pattern. But the user can still enter a Customer ID that is of the pattern we want but does not exist in the database, so we must validate that the Customer ID exists in the database. To do so, we have to write our own function to validate and then tie the function to the validation controls. This function should return *True* if the Customer ID exists and *False* if it doesn't. For these custom validations, we use the CustomValidator control.

7. Place the CustomValidator control next to RegularExpressionValidator and set the following properties:
   ID: **CustomValidator**
   ErrorMessage: **CustomerID does not exist**
   ControlToValidate: **txtCustomerID** (pick from the list)
   Display: **Dynamic**
8. In the design view, double-click **CustomValidator** to open the code window to write the customer validator code to validate the Customer ID entered by the user.
9. Place the following code in the code window (WebForm1.vb):

```
Private Sub CustomValidator_ServerValidate( _
      ByVal source As System.Object, _
      ByVal args As
          System.Web.UI.WebControls.ServerValidateEventArgs) _
      Handles CustomValidator.ServerValidate
    Dim sConnectionString As String
    Dim sqlString As String
    Dim MyConnection As SqlConnection
    Dim MyDataAdapter As SqlDataAdapter
    Dim DS As New DataSet()

    'getting the connection string
    sConnectionString = Context.GetConfig("appSettings")("DSN")

    'building the select statement
    sqlString = "SELECT * FROM Customers "
    sqlString += "WHERE CustomerID = '" + args.Value + "'"
```

```
        'opening the connection
        MyConnection = New SqlConnection(sConnectionString)
        MyDataAdapter = New SqlDataAdapter(sqlString, MyConnection)
        'getting Data Set
        MyDataAdapter.Fill(DS, "Customers")
        If DS.Tables("Customers").DefaultView.Count = 0 Then
            args.IsValid = False
        Else
            args.IsValid = True
        End If

    End Sub
```

In this function, we check whether the user exists in the database and set the *isValid* property to **True** if the user exists and **False** if the user does not exist. This function takes two input parameters. The first parameter is the CustomValidator control; the second parameter contains the value entered by the user in that input control. Here I used DataSet to retrieve from the database, you can also use DataReader. The CustomValidator control invokes this function to validate, and if the isValid property of args is False, it shows the error message on the page.

10. Run the program to test our exercise. You will see the Customer Orders column only when you enter a valid Customer ID. In all other cases, you will get the error message.

    Among the other controls, ValidationSummary control is the most used because it can show all the errors raised by various validation controls. Using it in our exercise does not make sense, because we have only one input control. When you have a large number of input controls, you can use ValidationSummary to show consolidated error messages.

Creating Custom Web Form Controls

In addition to the ASP.NET server controls, we can author our own controls to encapsulate a custom user interface. We can create a custom control using existing server controls, thus providing an easy way to reuse code. We can create two types of custom controls. The first one is to convert an existing Web form to a control with few modifications. This is often called *user controls* or *pagelets*. The second type is creating the control programmatically. This is known as *custom controls*.

In Exercise 10.6 we concentrate on custom controls. First we create a simple custom control that has properties. Then we take a close look at the code. After that, we create a composite control that has other server controls in it. We also see how to handle events raised by child controls in a composite control. Finally, we raise events from this control.

CD Exec. 10.6

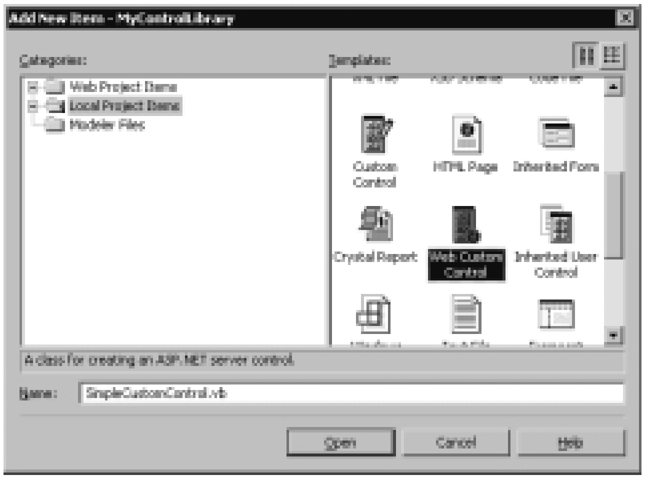
Exercise 10.6 A Simple Custom Control

Creating a custom control is very easy. You will be surprised to know that you can create one without writing any code. Let's create the standard Hello World example again. In order to create a custom control, we can use our existing project, Chapter10, but instead let's create a project that contains a library of our custom controls.

1. Begin a new Visual Basic Web project by selecting **File | New | Project** and then selecting **Web Control Library** under Visual Basic Projects. The New Project dialog box should be similar to the one that was shown in Figure 10.1.
2. Change the name of the project to **MyControlLibrary**. Enter the location where you want to create the project. Click **OK**, and Visual Studio.NET will create the Web project. Once Visual Studio creates the project, the screen looks like the one that was shown in Figure 10.2.
3. Ignore the default file WebControl1.vb. Instead, we'll create a new Web custom control. Click **Project | Add New Item** and in the open dialog box, select **Web Custom Control** and set its name to **SimpleCustomControl.vb**. Your screen should be similar to the one shown in Figure 10.9. Click **Open** for VS.NET to add a new custom control to our project.

   You can see that Visual Studio automatically generates code for you. If you observe the code in the SimpleCustomControl.vb and WebControl1.vb files, the code is exactly the same except for the class name. Changing the class name in WebControl1.vb does not help because VS.NET still remembers the original name.

Figure 10.9 Adding a Custom Control to a Project



A class for creating an ASP.NET server control.

Name: SimpleCustomControl.vb

4. Build the control by clicking the **Build | Build** menu item.

   We have created our first control. Before seeing how to use it in a Web form, let's see the code. This is the code generated by Visual Studio:

```
Imports System.ComponentModel
Imports System.Web.UI

<DefaultProperty("Text"), ToolboxData("<{0}:SimpleCustomControl
    runat=server>
    </{0}:SimpleCustomControl>")>
    Public Class SimpleCustomControl
Inherits System.Web.UI.WebControls.WebControl

Dim _text As String

<Bindable(True), Category("Appearance"), DefaultValue("")>
     Property [Text]() As String
     Get
          Return _text
     End Get

      Set(ByVal Value As String)
```

```
            _text = Value
        End Set
End Property

Protected Overrides Sub Render(ByVal output As
        System.Web.UI.HtmlTextWriter)
      output.Write([Text])
End Sub

End Class
```

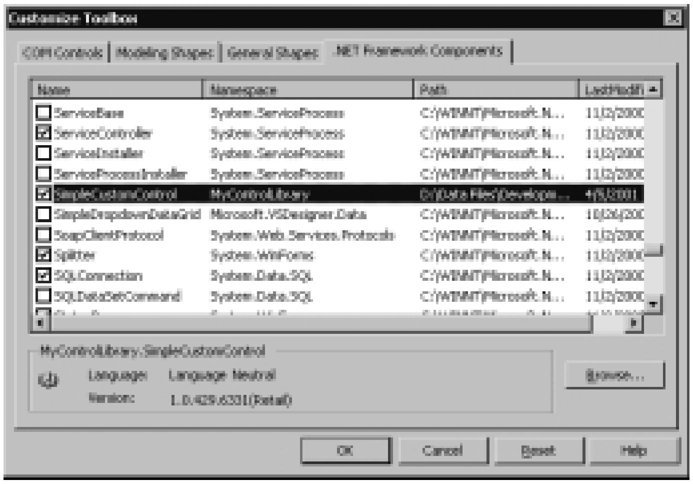
Any custom Web control must be inherited from WebControl. In this code, we defined a SimpleCustomControl class that inherits from WebControl with some attributes. The attributes are used to specify additional information during runtime and are used primarily by the IDE. The first attribute, *DefaultProperty*, tells the default property of this class. The other attribute, ToolboxData, will be used by Visual Stuido.NET when we drag and drop this control into the Web form from the toolbox. We will see how to do that when we discuss a client program for our control. Next we defined a private member of the class and used the naming convention that every private variable starts with an underscore. After that we defined a property, [Text], which is used to set and get the value of our private member. As mentioned earlier, this property is defined as the default property of the class using the attributes in the class definition. This property is also defined with attributes. These attributes are also used primarily by IDE. Among these attributes, *Bindable* lets this property be bound to any other item on the page. *Category* defines which category in the Properties dialog box this property should be shown. *DefaulValue*, as its name implies, contains the default value of the property.

The only method in this class, *Render*, overrides the parent class, WebControl, so that our class can control the output. This method is used to send HTML to the browser. This is done using the input parameter, output, to this method. In this method we are writing out the text set by the user using the Property dialog box.

Let's create a Web form that uses our control. We will create this Web form in our Chapter10 project. In that project we will add our control to the toolbox and then place it on the browser.

1. Open the Web project Chapter10.
2. Add a new Web form by clicking **Project | Add Web form**. Set the name of the Web form as **SimpleControlWebForm.aspx** and then click **Open**.
3. In order to place the control on the toolbox, select **Tools | Customize Toolbox**. This opens a Customize Toolbox dialog box, as shown in Figure 10.10.

Figure 10.10 Adding Custom Controls to the Toolbox



Customize Toolbox

COM Controls | Modeling Shapes | General Shapes | .NET Framework Components

Name	Namespace	Path	LastModifi
☐ ServiceBase	System.ServiceProcess	C:\WINNT\Microsoft.N...	11/2/200
☑ ServiceController	System.ServiceProcess	C:\WINNT\Microsoft.N...	11/2/200
☐ ServiceInstaller	System.ServiceProcess	C:\WINNT\Microsoft.N...	11/2/200
☐ ServiceProcessInstaller	System.ServiceProcess	C:\WINNT\Microsoft.N...	11/2/200
☑ SimpleCustomControl	MyControlLibrary	D:\Data Files\Developm...	4/5/2001
☐ SimpleDropdownDataGrid	Microsoft.VSDesigner.Data	C:\WINNT\Microsoft.N...	10/26/200
☐ SoapClientProtocol	System.Web.Services.Protocols	C:\WINNT\Microsoft.N...	11/2/200
☑ Splitter	System.WinForms	C:\WINNT\Microsoft.N...	11/2/200
☑ SQLConnection	System.Data.SQL	C:\WINNT\Microsoft.N...	11/2/200
☐ SQLDataSetCommand	System.Data.SQL	C:\WINNT\Microsoft.N...	11/2/200

MyControlLibrary.SimpleCustomControl

Language: Language Neutral

Version: 1.0.429.6331(Retail)

Browse...

OK | Cancel | Reset | Help

4. Click **.NET Framework Components** and then click the **Browse** button to select our DLL. When you compile the control, VS.NET creates a DLL and places it in the bin folder in that project. Select that DLL and click **Open**. Scroll to **SimpleCustomControl** (the name of our control) and click the check box next to it, and then click **OK** in the Customize Toolbox dialog box. This adds our control to the toolbox.
5. You can see our control under the General tab in the toolbox. If you want to place this control under a different tab in the Toolbox, right-click the tab wherever you want to place the control and then click Customize Toolbox, which brings up the Customize Toolbox shown in Figure 10.10, and repeat the process.
6. In the Design tab, drag and drop the SimpleCustom control from the toolbox. Observe that a reference to MyControlLibrary is added to the References folder of the project.

7. Set the following properties using the Properties dialog box:

   ID: **MySimpleControl**

   Text: **Hello World**

   Remember, *Text* is the default property that we defined in our control, and we are writing its value out as HTML in the *Render* method. Let's see the code VS generated.

   VS.NET places the following line of code at the top of the page. With this code, VS.NET sets the alias to our namespace: *MyControlLibrary*. Using the *TagPrefix* attribute, we are setting the alias to our namespace as cc1:

```
<%@ Register TagPrefix="cc1" NameSpace="MyControlLibrary"
                   Assembly="MyControlLibrary" %>
```

   After that, VS.NET uses this *cc1* as the tag prefix. We defined this tag using the Register directive. *Text* is the property we set using the Properties dialog box:

```
<cc1:SimpleCustomControl id=MySimpleControl runat="server"
                        Text="Hello

      World"></custom:SimpleCustomControl>
```

8. Save the Web form and run it. You will see *Hello World* on the browser.

We have created our first control. If you want to add more properties, copy the existing property with attributes and change the property name and the *Get* and *Set* methods. You can use the *Render* method to send out whatever HTML you want. Basically, the *Render* method is used to send HTML to the browser. You can't use this method if your control uses other ASP.NET server controls.

Now let's create a control that uses other server controls. This control is known as a *composite custom control.*

CD Exec. 10.7

Exercise 10.7 Creating a Composite Custom Control

In this exercise, we create a control that uses other ASP.NET server controls. As these controls raise events, we will see how to handle those events, and we will raise events from our control. In order to create a composite custom control, we convert some part of our previous work into a control. In Exercise 10.2, we placed four controls on the Web form (refer back to Figure 10.3). In this exercise, we create a composite control with those four controls. Unfortunately, the process of creating composite controls does not support drag and drop; we have to programmatically add those controls. In addition, our control should return the Customer ID entered by the user so that the client program can consume it. In order to do that, we must declare a read-only property that returns the Customer ID entered by the user. We will create this control under MyControlLibrary project and use our Chapter10 project as a client program.

1. Open the MyControlLibrary project.
2. Close the Chapter10 project. Remember, in this project we set a reference to the SimpleCustomControl, which is part of MyControlLibrary. Visual Studio locks the MyControlLibrary.dll, so we can't compile the MyControlLibrary project unless we close the Chapter10 project.
3. To add a new Web custom control to our project, click **Project | Add New Item** and, in the open dialog box, select **Web Custom Control** and set its name to **CompositeCustomControl.vb**. Your screen should be similar to the one shown in Figure 10.9. Click **Open** for VS.NET to add a new custom control to our project.
4. Because we are using built-in server controls in this control, we don't require the *Render* method. So delete the *Render* method from the generated code. Removing this method is important because it has higher precedence over other methods.
5. Place the following import statement at the top of the class. This statement imports all the Web controls into our class:

   ```
   Imports System.Web.UI.WebControls
   ```

6. Add a private variable, which is used to store the CustomerID inside the class:

   ```
   Dim _CustomerID As String
   ```

7. Place this code inside the class, which creates a read-only property that returns the Customer ID:

   ```
   Public ReadOnly Property CustomerID() As String
      Get
         Return _CustomerID
      End Get
   End Property
   ```

8. Place the following code inside the class:

```
Protected Overrides Sub CreateChildControls()
    'adding Para
    Me.Controls.Add(New LiteralControl("<p>"))

    'adding title
    Dim lblCustomerOrder As New Label()
    lblCustomerOrder.Text = "Customer Order Details"
    lblCustomerOrder.Font.Size = FontUnit.XLarge
    Me.Controls.Add(lblCustomerOrder)

    'closing para
    Me.Controls.Add(New LiteralControl("</p>"))

    'adding Para
    Me.Controls.Add(New LiteralControl("<p>"))

    'adding Label control
    Dim lblCustomerID As New Label()
    lblCustomerID.Text = "CustomerID"
    lblCustomerID.Font.Bold = True
    Me.Controls.Add(lblCustomerID)

    'adding Text Box
    Dim txtCustomerID As New TextBox()
    Me.Controls.Add(txtCustomerID)

    'adding closing para
    Me.Controls.Add(New LiteralControl("</p>"))
    Me.Controls.Add(New LiteralControl("<p>"))

    'adding button
    Dim cmdGetDetails As New Button()
    cmdGetDetails.Text = "Get Order Details"
```

```
     'adding event handler
      AddHandler cmdGetDetails.click, AddressOf
          cmdGetDetails_Click
      Me.Controls.Add(cmdGetDetails)

      'putting the closing para
      Me.Controls.Add(New LiteralControl("</p>"))

End Sub
```

Before going through this code, let's first see the HTML code generated by IDE in Exercise 10.2, when these four controls are placed on the Web form. The code generated by IDE is:

```
<p>
   <asp:Label id= lblCustomerOrder runat="server" Font-Size="X-
       Large">
         Customer Order Details
   </asp:Label>
</p>
<p>
   <asp:Label id=lblCustomerID  runat="server">Customer
       ID</asp:Label>
   <asp:TextBox id=txtCustomerID runat="server"></asp:TextBox>
</p>
<p>
   <asp:Button id=cmdGetDetails runat="server"
               Text="Get Order Details"></asp:Button>
</p>
```

In order for us to create a composite control, our class should have a *CreateChildControls* method that overrides its parent class method. In this method, we programmatically add all of these controls to our composite control. In the preceding HTML code, the HTML element, <p> tag, separates the controls. So we have to add the HTML elements to our composite control. To add HTML elements, we have to convert them into controls using the LiteralControl API and then add them to our control. All the HTML elements and the server controls should be added in the same order as in the preceding HTML code.

So, in the *CreateChildControls* method, we are converting the HTML elements into controls and adding them to our controls collection. The code to convert and add HTML elements is:

```
Me.Controls.Add(New LiteralControl("<p>"))
```

For adding the server controls, we first must declare the control object and then set its properties, similar to what we did in Exercise 10.2 using the Properties dialog box. Finally, we have to add it to the controls collection. For example, for the Customer Order Details label control, our code will be:

```
    Dim lblCustomerOrder As New Label()
    lblCustomerOrder.Text = "Customer Order Details"
    lblCustomerOrder.Font.Size = FontUnit.XLarge
    Me.Controls.Add(lblCustomerOrder)
```

In this code, we are declaring a variable of type *Label* and setting its Text and Font-Size properties, then adding it to the controls collection. We have to do the same for the rest of the controls. In Exercise 10.2, our program handled a *Button Click* event. In order for our control to handle events raised from child controls, our control should implement INamingContainer. When you implement this interface, the framework automatically generates unique IDs for each control.

9. After the class definition, add the *implements* clause to implement this interface:

```
Implements INamingContainer
```

10. To handle the *Click* event of its child control button, we must use the *AddHandler* method. The following code shows how to use this method:

```
    Dim cmdGetDetails As New Button()
    cmdGetDetails.Text = "Get Order Details"
    'adding event handler
     AddHandler cmdGetDetails.click, AddressOf
         cmdGetDetails_Click
     Me.Controls.Add(cmdGetDetails)
```

The *AddHandler* method takes two parameters, an event expression and the delegate to handle the event. With the *AddHandler* method we are associating the *Click* event of our button with the procedure cmdGetDetails_Click in our class, so we have to add this method to our class. Similar to other delegates, this method takes two input parameters: the sender of the event and EventArgs.

In order for the client program to handle the *Click* event, our control should raise an event when the button is clicked. For that we have to define an event in our control.

11. Add the following code in the declaration section of our class:

```
Public Event Click(ByVal sender As Object, ByVal e As
    System.EventArgs)
```

Similar to any other event delegate, this *Click* event takes two input parameters. We have to raise this event when the button is clicked. Because we associated the *Button Click* event with the procedure cmdGetDetails_Click, we will raise this event in that procedure.

12. When the button is clicked, we retrieve the Customer ID entered by the user in the text box and then set our variable with this value so that it can be passed to the client program. Then we raise a *Click* event. In order to do this, add the following procedure to the class:

```
Private Sub cmdGetDetails_Click(ByVal sender As Object,
         ByVal e As System.EventArgs)
    Dim txtCustomerID As TextBox = CType(Controls(5), TextBox)
    _CustomerID = txtCustomerID.Text
    RaiseEvent Click(Me, EventArgs.Empty)
End Sub
```

In this procedure, we first retrieve the text entered in the text box. Because TextBox is the sixth control we added to the controls collection, we have to retrieve it by the sequential number and type-cast it to TextBox. After setting the private variable with the value entered by the user, we raise a *Click* event using the *RaiseEvent* method. *RaiseEvent* raises the *Click* event to the client program; to this event we send our control class as one of the inputs. Because there is no additional information to send, we pass an empty state.

13. Build the control by clicking **Build | Build**.

Now we created a composite control. Let's create a Web form that consumes our control. We create this Web form in our Chapter10 project. In that form, we place this control along with a label control. On this form, we handle the *Click* event of our control; on the *Click* event, we set the text on the label to the user-entered value.

1. Open the Chapter10 project.
2. Add a new Web form by clicking **Project | Add Web form**. Set the name of the Web form as **CompositeControlWebForm.aspx** and then click **Open**.
3. In order to place the control on the toolbox, select **Tools | Customize Toolbox**. This opens a Customize Toolbox dialog box, as was shown in Figure 10.10.
4. Switch to**.NET Framework Components** and click the **Browse** button, then select **MyControlLibrary.dll** and click **Open**. Scroll to **CompositeCustomControl** (name of our control) and click the check box next to it, then click **OK** in the Customize Toolbox dialog box. Sometimes Visual Studio remembers the previous reference, so you might get an error. If you get error message, then click **Cancel** on the dialog box to close it. Click to open the **References** folder in the Solution Explorer, and then right-click the **MyControlLibrary** reference item and click **Remove**. Close the project and then reopen it.
5. Repeat the process of adding the control to the toolbox. This process adds our control to the General tab of the toolbox.
6. Drag and drop the **CompositeCustom** control from the toolbox and resize it to fit the page.
7. Set the following properties using the Properties dialog box:
   ID: **MyCompositeControl**
8. Place another label control on the Web form and set the following properties:
   ID: **lblTest**
   Text: **""** (remove the existing text)
9. Now let's write the code to handle the *Click* event for our control. Double-click the control to bring up the code window. As our composite control raises the *Click* event, add the following code to the Web form. The Handles keyword attaches this method to the *Click* event of our custom control:

```
Protected Sub MyCompositeControl_Click(ByVal Sender As

    System.Object,
           ByVal e As System.EventArgs) Handles
               MyCompositeControl.Click
      lblTest.Text = MyCompositeControl.CustomerID
End Sub
```

When the button is clicked, the preceding code changes the text of the label to the Customer ID entered by the user.

10. Press **F5** to run the program. Now when you click the button, you will see that the text on the label changes to whatever you have entered.

We can even add the validation controls to validate the user input, which we used in Exercise 10.5. As an assignment, try adding validation controls to this composite control.

In this section, we saw how to create a simple control that generates only HTML. After that, we created a composite control that has other server controls. We also saw how to handle the events raised by the child controls and how to raise events from our control. In addition, a custom control can read the inner content (i.e., text added between its tags), can handle postback data, and supports Templated control similar to DataGrid control.

Web Services

With the advent of the next generation of the Internet, the Internet is no longer used to render UI pages. Now it has become a bridge between many applications, such as the business-to-business (B2B) marketplace and e-procurement. This new Internet will change the application architecture to provide information when and where you want it. With this next generation of the Internet, programmable Web site companies can expose their software as a service over the Internet to their business partners to fully leverage connected computing. Such services are called *Web services*.

A *Web service* is a component that provides service to a consumer, who uses standard Internet protocols (HTTP, XML) to access these services. These Web services are the custom business components that have no user interface and are meant to be consumed by programs only. Any client that understands HTTP and XML can consume Web services. Because Web services use HTTP, they are firewall friendly and have tremendous advantages over DCOM. A simple scenario in which we can use Web services is in calculating sales tax for an e-commerce application. This application requires maintaining tables for calculating tax and frequently upgrading the data received from the vendor. Instead, if the vendor makes this a Web service accessible over the Internet, we can use it whenever we want, without the hassle of maintaining the data.

How Web Services Work

Web services are programmable components that can be accessed over Internet protocols. They use XML and Simple Object Access Protocol (SOAP) to communicate with consumers. XML provides a standardized language to exchange data in a widely accepted format. SOAP is a simple, lightweight XML-based protocol that runs over HTTP for exchanging information in a distributed, heterogeneous environment. In other words, SOAP = HTTP + XML.

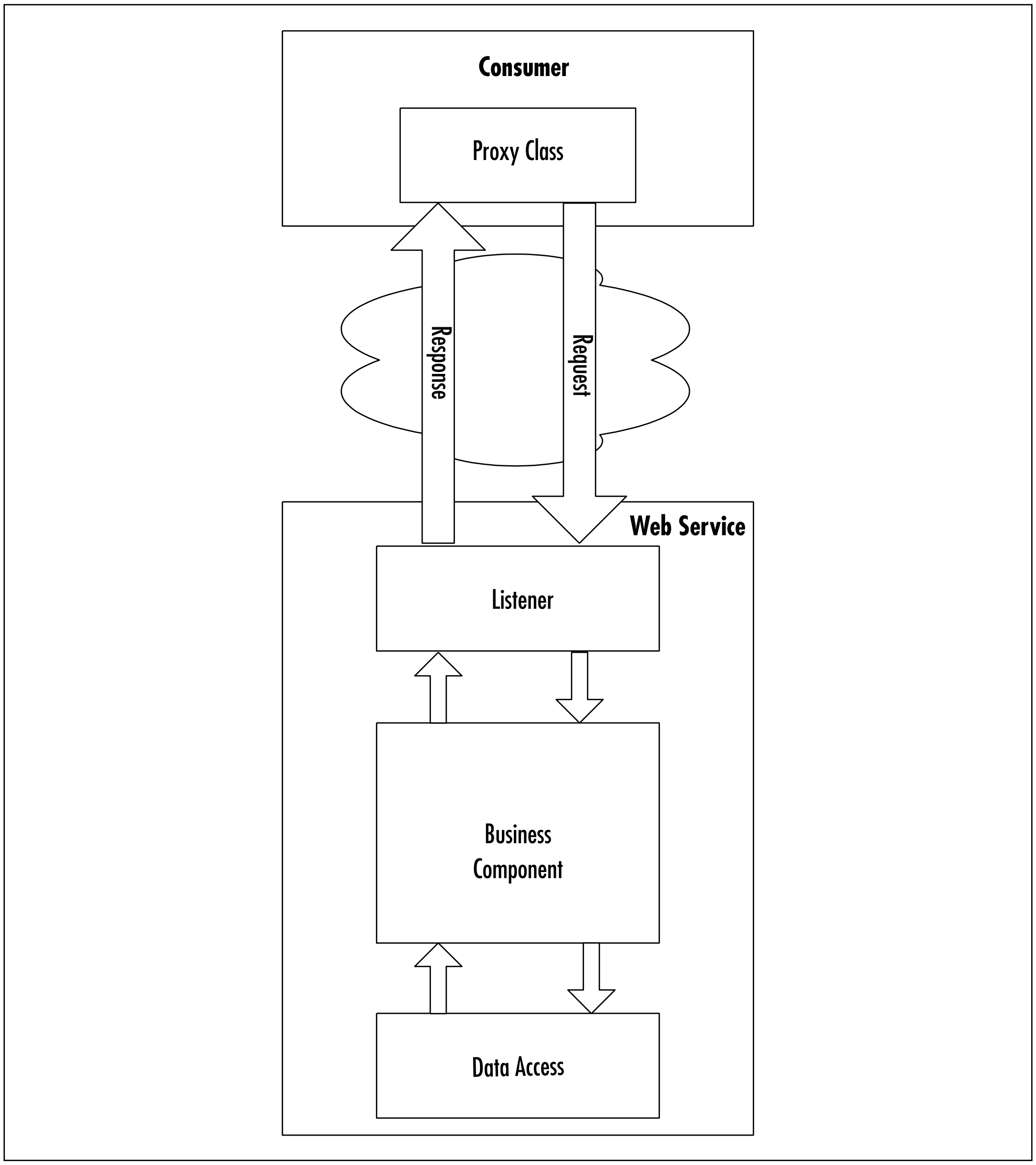
Figure 10.11 shows the architecture of a Web service. The consumer sends requests to the Web service over the Internet using the SOAP message format. Once the SOAP request arrives, IIS, the listener listening on the TCP port 80, routes the request to the ASP.NET handler, which locates the Web service, creates the business component, calls the specified method in the object, and passes it the data. This business component processes the request and, if necessary, gets data from the database or from other Web services. It then returns results to ASP.NET, which then packs it in a SOAP envelope and sends it back to the consumer. On the consumer side, .NET provides a proxy class that converts this SOAP message to a data type. This proxy class also packs the request into a SOAP envelope and sends it to the Web service. SOAP is the default communication protocol for Web services. In addition, Web services can also be accessed using HTTP-GET and HTTP-POST protocols.

When using Visual Studio to create and consume Web services, you don't necessarily need to know all this architecture. .NET Framework converts everything for you under the hood. You can create and consume Web services without knowing anything about XML and SOAP.

Developing Web Services

Because Web services are accessible over the Internet, they are saved in a file with extension .ASMX. Similarly to .ASPX files, these are compiled when they are accessed for the first time.

Figure 10.11 Web Service Architecture



CD Exec. 10.8

Exercise 10.8 Developing Web Services

In this exercise, we create a Web service in our project, Chapter10.

1. Open the Chapter10 project.
2. Click **Project | Add Web Service**, and enter the name of our Web service as **MyWebService.asmx**. Web services have an .ASMX file extension.
3. Visual Studio creates the Web service component and shows the design area where you can drag and drop controls. Because a Web service doesn't have a UI, double-click the form to get into the code. You can see that Visual Studio already prepopulated the necessary code.

   In order to make any class into a Web service component, that class must inherit the WebService class. Once the class is inherited from the WebService class, it exposes the public methods declared with the attribute Web Method over the Internet.
4. In the code generated, Visual Studio created a sample Web method, the classic *Hello World*, and commented it. Remove those comments. The sample method generated by Visual Studio is:

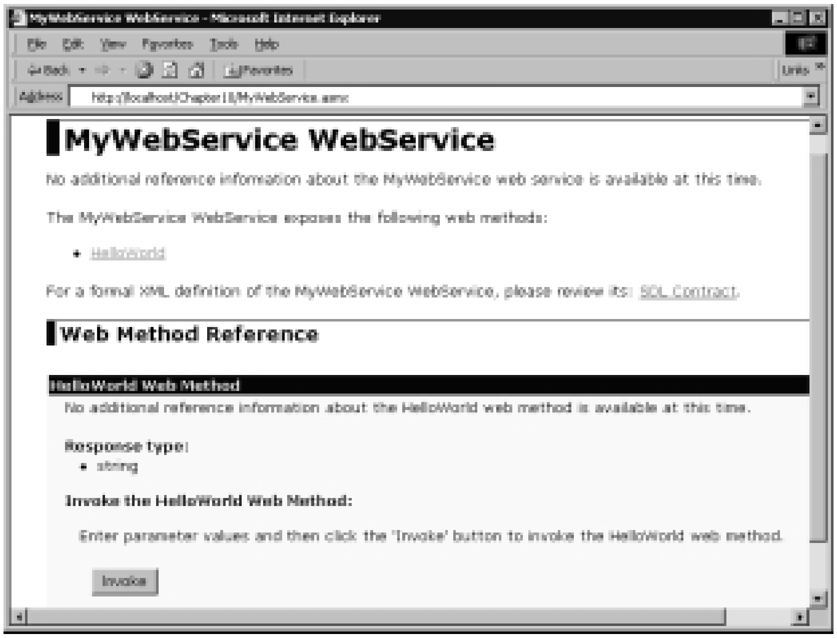
```
<WebMethod()> Public Function HelloWorld() As String
   HelloWorld = "Hello World"
End Function
```

   Simply adding the attribute *WebMethod* to this method makes it a Web method that can be accessible over the Internet.
5. Build the project by clicking **Build | Build**.
6. Test the Web service by typing its URL in the browser. In our case, the URL is **http://localhost/Chapter10/MyWebService.asmx**. Figure 10.12 shows our Web service in a browser.

   When you type the URL in the browser, ASP.NET detects it as a Web service and shows you a list of available methods. In our case, there is only one method.
7. Click the **HelloWorld** link and then the **Invoke** button in the browser. This opens another window, which returns the results in XML format. The XML returned by the Web service is:

```
<?xml version="1.0" ?>
<string xmlns="http://tempuri.org/">Hello World</string>
```

Figure 10.12 MyWebService.asmx in a Browser


MyWebService WebService - Microsoft Internet Explorer

Address http://localhost/Chapter10/MyWebService.asmx

MyWebService WebService

No additional reference information about the MyWebService web service is available at this time.

The MyWebService WebService exposes the following web methods:

- HelloWorld

For a formal XML definition of the MyWebService WebService, please review its: SDL Contract.

Web Method Reference

HelloWorld Web Method

No additional reference information about the HelloWorld web method is available at this time.

Response type:
- string

Invoke the HelloWorld Web Method:

Enter parameter values and then click the 'Invoke' button to invoke the HelloWorld web method.

The XML string returned specifies that the return data type is *string* and its value is *Hello World*. If you observe the URL of the browser returning XML, you can notice that we are trying to access the Web service over the HTTP-GET protocol.

We created our Web service without writing any line code and without knowing anything about XML and SOAP. Adding a Web method to a Web service is just adding an attribute. For example, we can convert the function we wrote in Exercise 10.3 to get orders from the database for a given customer as a Web method accessible over the Internet.

8. Copy the function GetOrders from Exercise 10.3 into this class and add the *WebMethod* attribute. Your function declaration should be like this:

```
<WebMethod()> Public Function GetOrders(ByVal CustomerID As
    String) As DataSet
```

With this declaration, the method becomes accessible over the Internet.

9. Because the GetOrders function uses ADO.NET to connect to the database, we have to import those namespaces into our class. Add this import statement to the class:

   ```
   Imports System.Data.SqlClient
   ```

10. In order to test it, build the application and type the URL of this Web service into the browser. Now you will see two methods instead of one method shown in Figure 10.12. Enter the Customer ID **HANAR** or any valid Customer ID. You will see that ASP.NET returns the orders in XML format.

We have created a Web service with two Web methods in it. Later on in this chapter we create a Web consumer and a Windows consumer to access our Web services. You will be surprised to know that accessing a Web service is much easier than creating one. You don't have to worry about SOAP and XML; Visual Studio makes everything transparent to you.

Web Service Utilities

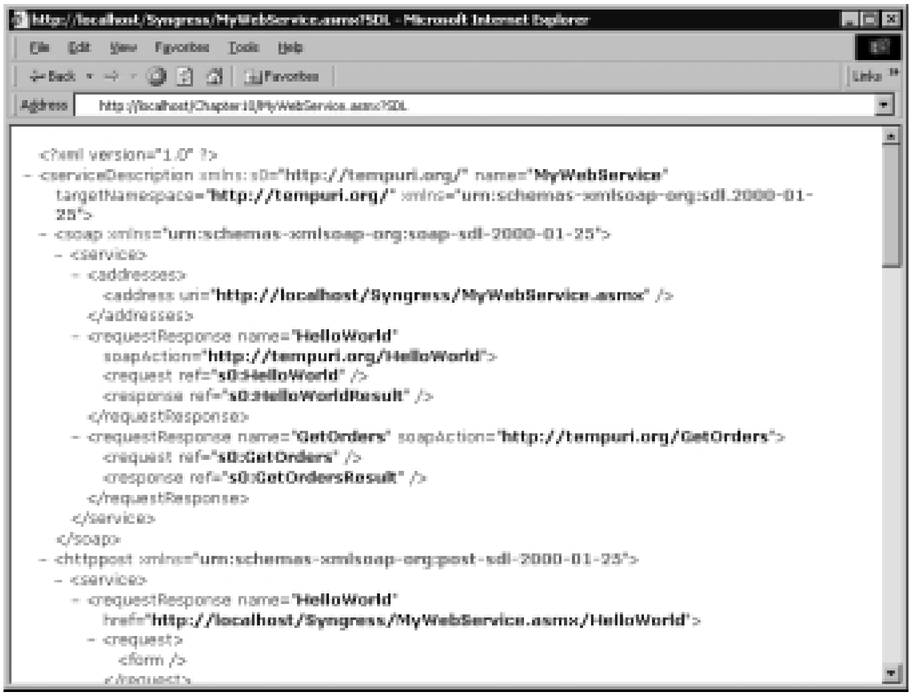
Now we have created a Web service with two Web methods in it. But how do our consumers know about this Web service? Our customers should know not only that a Web service exists but also what methods are exposed, the parameters required, and the protocols supported. In addition, our customers should be able to use these components without knowing about the architecture. Luckily, we don't have to handcraft all these things; .NET Framework takes care of it for us. ASP.NET describes all the methods and parameters in a Web service via the Service Description Language (SDL). Even ASP.NET lets you find all the Web services available on a Web server. Furthermore, ASP.NET creates a proxy class for us to consume these Web services. Let's see how ASP.NET does these things behind the scenes.

Service Description Language

A Web service can be asked for a list of methods and should respond with a description in an understandable format. SDL defines the message format the Web service understands. The SDL contract uses XML format to describe the protocols supported by the Web service (SOAP, HTTP-GET, HTTP-POST), instantiable methods, and inputs (request) and outputs (response) of these methods. SDL is like a type library in a COM object. In order to request the Web service to return the SDL contract, append the query string **?SDL** to the

URL of the Web service. In our case, the URL is **http://localhost/Chapter10/MyWebService.asmx?WSDL**. Figure 10.13 shows the SDL contract of our Web service. Alternatively, you can view the SDL contract by clicking the link when consuming the Web service on a browser (in Figure 10.12, for example, click the SDL contract link to view the SDL).

Figure 10.13 SDL Contract of the Web Service


http://localhost/Syngress/MyWebService.asmx?SDL - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Favorites Links
Address http://localhost/Chapter10/MyWebService.asmx?SDL

```
  <?xml version="1.0" ?>
- <serviceDescription xmlns:s0="http://tempuri.org/" name="MyWebService"
    targetNamespace="http://tempuri.org/" xmlns="urn:schemas-xmlsoap-org:sdl.2000-01-
    25">
  - <soap xmlns="urn:schemas-xmlsoap-org:soap-sdl-2000-01-25">
    - <service>
      - <addresses>
          <address uri="http://localhost/Syngress/MyWebService.asmx" />
        </addresses>
      - <requestResponse name="HelloWorld"
          soapAction="http://tempuri.org/HelloWorld">
          <request ref="s0:HelloWorld" />
          <response ref="s0:HelloWorldResult" />
        </requestResponse>
      - <requestResponse name="GetOrders" soapAction="http://tempuri.org/GetOrders">
          <request ref="s0:GetOrders" />
          <response ref="s0:GetOrdersResult" />
        </requestResponse>
      </service>
    </soap>
  - <httppost xmlns="urn:schemas-xmlsoap-org:post-sdl-2000-01-25">
    - <service>
      - <requestResponse name="HelloWorld"
          href="http://localhost/Syngress/MyWebService.asmx/HelloWorld">
        - <request>
            <form />
          </request>
```

Discovery

SDL is useful if you know which Web service you want. What good is a Web service if consumers don't know it exists? Web Service Discovery helps locate and interrogate Web service descriptions. Each Web site publishes all of its Web services in a .VSDISCO file. This file is an XML document that contains URLs of all the SDL descriptions. With this discovery process, consumers learn that a Web service exists, what its capabilities are, and how to interact with it.

Proxy Class

Web consumers must send messages to a Web service using SOAP. You can write SOAP marshalling to send and receive data from the Web service over HTTP, or you can use .NET Framework to create a proxy class that contains the appropriate network invocation and marshalling code to invoke and receive responses from the Web service. This proxy class can be referenced in the client program and used to invoke a Web service as though invoking a local method.

If you use Visual Studio to create and consume Web services, you don't have to worry about all this—.NET Framework does everything for you behind the scenes. With a couple of clicks, you can access any Web service as though it were a class in your assembly.

Consuming Web Services from Web Forms

In the previous exercise, we created a Web service with two Web methods in it. Now let's create another exercise in which we consume the *GetOrders* method in that Web service. In Exercises 10.3, 10.4, and 10.5, we bound the DataGrid with the DataSet returned by the GetOrders function. Later we changed this function into a Web service. So let's change our code in Exercise 10.5 to use this Web service.

CD Exec. 10.9

Exercise 10.9 Consuming Web Services from Web Forms

In this exercise, we set a reference to the Web service in order to consume it, then we change the code on the *Button Click* event in Exercise 10.5 to get order details from this Web service.

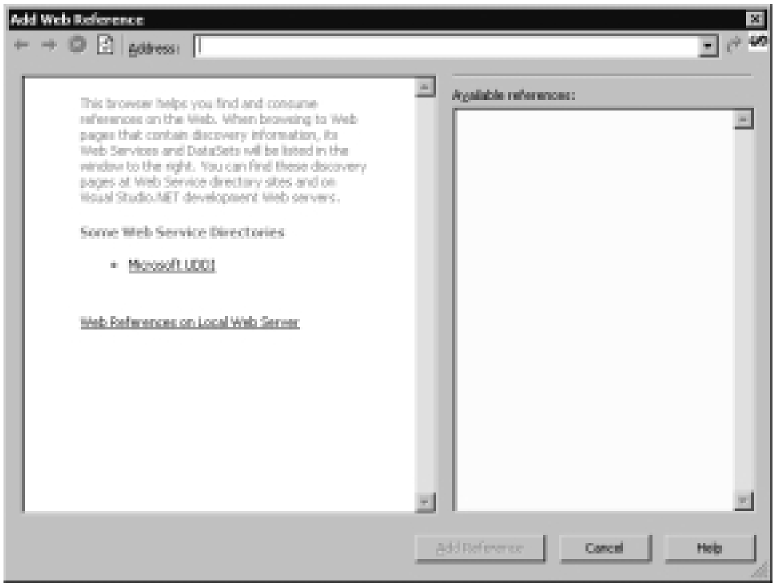
1. Open the Chapter10 project.
2. Open the Web form WebForm1.aspx.
3. Double-click the **Get Order Details** button to open the code window.
4. On the *Click* event of this button we previously wrote the following code:

```
Dim DS As DataSet
'only if the Page is Valid then only binding to the DataGrid
If Not Page.IsValid Then Exit Sub
'getting the DataSet with Order Details for the entered
    CustomerID
DS = GetOrders(txtCustomerID.Text)
'Binding the DataGrid
dgOrders.DataSource = DS.Tables("Orders").DefaultView
dgOrders.DataBind()
```

In this code, we use a function inside our class to retrieve data from the database. Instead, we now use the Web service we created, which returns a DataSet. In order to use a Web service, we have to set reference to that Web service.

5. Click **Project | Add Web Reference**, which opens a dialog box to add a Web reference, as shown in the Figure 10.14.

Figure 10.14 The Add Web Reference Dialog Box



This browser helps you find and consume references on the Web. When browsing to Web pages that contain discovery information, its Web Services and DataSets will be listed in the window to the right. You can find these discovery pages at Web Service directory sites and on Visual Studio.NET development Web servers.

Some Web Service Directories

- Microsoft UDDI

Web References on Local Web Server

If you know the address of the Web service, you can type it in the address text box right away. Alternatively, you can search for all the Web services that are registered using the discovery process on your local Web server. Microsoft Universal Description Discovery Integration (UDDI) links to all the available Web services registered on the Internet.

6. Enter the address of our Web service, **http://localhost/Chapter10/MyWebService.asmx**, in the address box, and press **Enter**. Now in the left pane you will see documentation about this Web service. Click the **Add Reference** button to add a reference to our project. Once you click the button, you can observe that a Web References folder is added to your Solution Explorer. This folder contains the reference you added.

7. Now let's access the Web method. Replace the one-line code to call the GetOrders function with this code:

   ```
   Dim WS As New localhost.MyWebService()
   DS = WS.GetOrders(txtCustomerID.Text)
   ```

   Again, we see how easy it is to invoke a method in a Web service. First, we declared a variable of data type *MyWebService* and then invoked the method in that Web service.
8. The complete *Button Click* code should be as follows:

   ```
   Dim DS As DataSet
   'only if the Page is Valid then only binding to the DataGrid
   If Not Page.IsValid Then Exit Sub
   'getting the Order details DataSet for the CustomerID using Web
       Service
   Dim WS As New localhost.MyWebService()
   DS = WS.GetOrders(txtCustomerID.Text)
   'Binding the DataGrid
   dgOrders.DataSource = DS.Tables("Orders").DefaultView
   dgOrders.DataBind()
   ```

9. Press **F5** to run the Web form. Enter the Customer ID and click the button. You will see the DataGrid populated with orders.

From the user point of view, there is no difference between getting the orders from the Web form itself or using a Web service. Visual Studio simplified the process of creating a Web service and consuming it, without worrying about the architecture.

Using Windows Forms in Distributed Applications

In the previous exercise, we got order details for a given Customer ID. This task would be the function of an administrator or a customer service representative. Because the users of this application are internal users, we could use Windows forms instead of Web forms and take advantage of the client processors. We can convert the previous example into a Windows form to provide a rich user interface. The only thing we require is that the orders placed by the customer are in the server. Because we are exposing a Web method on the server that returns the orders for a Customer ID, we can create a distributed Windows application that consumes this Web service.

CD Exec. 10.10

Exercise 10.10 Consuming Web Services from Windows Forms

In this exercise, we create a Windows form that consumes our Web service and provides the same functionality as the Web form.

1. Create a new Windows application project by clicking **File | New | Project** and selecting **Windows Application** under Visual Basic Projects.
2. Change the application name to **Chapter10WindowsApplication**. Click **OK** for Visual Studio to create the project.
3. Set the following form properties:
   Text: **Customer Order Details**
4. Place the following controls and set their properties using the Properties dialog box:
   Label
   Text: **Customer Order Details**

   Font-Size: **16**

   Font-Bold: **True**

   Name: **lblCustomerOrder**
   Label

   Text: **Customer ID**

   Font-Bold: **True**

   Name: **lblCustomerID**

   TextBox

   Text: **“”** (empty)

   Name: **txtCustomerID**

   Button

   Text: **Get Order Details**

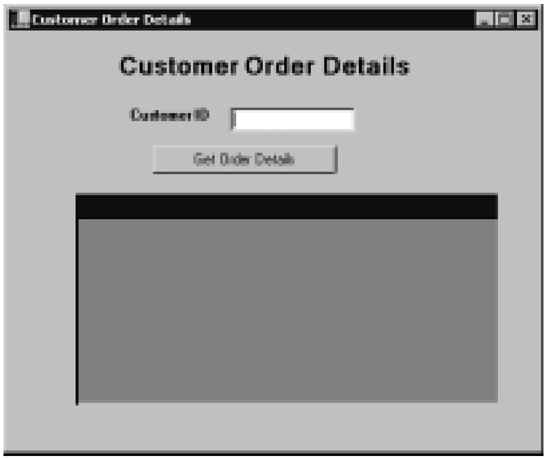
   Name: **cmdGetDetails**

DataGrid

Name: **dgOrders**

After placing these controls, your form should resemble Figure 10.15.

Figure 10.15 Windows Form View with Controls



5. Set a reference to the Web service, by clicking **Project | Add Web Reference** (see Figure 10.14) and then type the URL address of the Web service, **http://localhost/Chapter10/MyWebService.asmx**, in the address text box. Press **Enter**. Click the **Add Reference** button to add a reference to this Web service.
6. Double-click the button to open the code window and place the following code on the *Click* event:

```
Dim DS As DataSet
'getting the Orders DataSet with for the CustomerID from the web
    service
Dim WS As New localhost.MyWebService()
DS = WS.GetOrders(txtCustomerID.Text)
'Binding the DataGrid
dgOrders.DataSource = DS.Tables("Orders").DefaultView
```

This code is similar to the code we wrote for the *Click* event on the Web form; the only difference is, we don't have to invoke the *Bind* method on the DataGrid.

7. Press **F5** to run the application. Enter the Customer ID, **HANAR**, and click the button to view all the orders placed by the customer.

In this section, we have created a distributed Windows application that consumes a Web service. Because Windows applications run on the client machine and use server resources remotely, they reduce the server load. Furthermore, because they use .NET sophisticated graphics, they can provide a rich user interface with the quickest response and the highest degree of interactivity.

CD Exec. 10.11

Exercise 10.11 Developing a Sample Application

We conclude this chapter by creating a sample application that uses all of the exercises that we created in this chapter. We will create this application from scratch. Like other examples, this application shows all of the orders placed for a given Customer ID, but it uses the composite custom control we created and then consumes our Web service to retrieve orders placed by the customer.

1. Create a new Web application project by selecting **File | New | Project** and **Web Application** under Visual Basic Projects. Set the name of this application to **SampleApplication**. If you are not using your local Web server, change the Location box to the Web server you want to use. Click **OK** to create the project.
2. Since we already added our controls to the toolbox, they should show up in the General tab of the toolbox. If you don't find them, add the CompositeCustomControl to the toolbox. In order to do that, select the menu item **Tools | Customize Toolbox**. In the opened dialog box, switch to the **.NET Framework Components** tab, and click the **Browse** button to choose the control we created. Navigate to the MyControlLibrary folder, where we created our control library, and select **MyControlLibrary.dll** under the bin directory. Click **Open**, and then scroll through the list and click the check box next to CompositeCustomControl.
3. Set a reference to the Web service we created. Select the menu item **Project | Add Web Reference**, which opens the Add Web Reference dialog box. In the Address box, enter the URL of our Web reference, **http://localhost/Chapter10/MyWebService.asmx**, and press **Enter**. Click **Add Reference** for Visual Studio to create a reference to this Web service in our project.

4. Switch to design view and drag and drop **CompositeCustomControl** and a **DataGrid** control from the toolbox. Press **Enter**, and then place a **DataGrid** control on the Web form and set these properties:
   CompositeCustomControl
   ID: **MyCompositeControl**
   DataGrid
   ID: **dgOrders**
   HeaderStyle-BackColor: **Navy**
   HeaderStyle-Font-Bod: **True**
   HeaderStyle-ForeColor: **White**
   AlternatingItemStyle-BackColor: **Silver**
5. In order to customize the DataGrid control, switch to HTML view and add the following code inside the DataGrid tag:

```
   <Columns>
    <asp:BoundColumn datafield="OrderID" headertext="Order ID"/>
    <asp:templatecolumn  headertext="Order Date">
     <ItemTemplate>
      <%# String.Format("{0:d}",
Container.DataItem("OrderDate")
           ) %>
     </ItemTemplate>
    </asp:templatecolumn>
    <asp:templatecolumn  headertext="Shipped Date">
     <ItemTemplate>
      <%# String.Format("{0:d}",
           Container.DataItem("ShippedDate") ) %>
     </ItemTemplate>
    </asp:templatecolumn>
    <asp:BoundColumn datafield="ShipName" headertext="Ship
        Name"/>
   </Columns>
```

As we saw earlier, this code formats the column headings and the date fields to show only the date.

6. Add the autogenerate columns attribute with a value of **False** to the datagrid tag:

   ```
   autogeneratecolumns="False"
   ```

7. Switch to the design area and double-click our custom control to open the code window.
8. Place the following code inside the class for the *Click* event of the button in our custom control. In this code, first we instantiate a Web service and invoke it with the Customer ID returned by the custom control. Then we bind DataGrid with the orders returned by the Web service:

   ```
   Protected Sub MyCompositeControl_Click(ByVal sender As Object, _
                                          ByVal e As EventArgs)
           Dim DS As DataSet
           Dim WS As New localhost.MyWebService()
           DS = WS.GetOrders(MyCompositeControl.CustomerID)
           dgOrders.DataSource = DS.Tables("Orders").DefaultView
           dgOrders.DataBind()
   End Sub
   ```

9. Press **F5** to run the application. Enter the Customer ID **HANAR** and press the button. You will see the DataGrid populates all of the orders placed by the customer.

We have created a sample application based on the controls and Web service we created previously in this chapter.

Summary

In this chapter, you learned how to use Visual Basic and .NET Framework to create Web applications. You saw the rich controls provided by ASP.NET that you can use to bring life to a Web page. We also covered the following topics: what a Web form is and how it differs from a Windows form and how to use Web forms to create programmable Web pages. We also looked at the categories of Web form control available to provide rich user interfaces. We discussed the DataGrid bound control, and we covered when and how to use various customization options. We looked at the built-in capabilities of data validation in Web forms to validate user inputs using validation controls. We discussed the ability to author custom controls to enhance the functionality of Web form controls and create reusable components as well as how Web services change the application architecture to provide data when and where we want it. We looked at ways to consume a Web service from a Web form as well as from a Windows form, without changing the way we program. We still have more to learn about ASP.NET, but with these skills, it's easy for you to master ASP.NET with a little bit of experimenting.

Solutions Fast Track

Web Forms

☑ Web forms extend the Rapid Application Development (RAD) capabilities of Visual Basic to Web applications, allowing developers to create rich, form-based Web pages.

☑ Web forms separate the code from the content on a page, eliminating spaghetti code.

☑ Similarly to Windows forms, Web forms support an event-driven model.

Adding Controls to Web Forms

☑ Web form controls are server-side controls that are instantiated on the server and render HTML to the browser.

☑ Placing the controls on a Web form is similar to placing controls on a Windows form. The only differences are that the layout of the Web form is linear and the controls are dropped where the cursor is currently positioned.

- ☑ Web form server controls are broadly classified into four categories: intrinsic, bound, custom, and validation controls.

Creating Custom Web Form Controls

- ☑ ASP.NET allows us to author our own controls to encapsulate a custom user interface.
- ☑ We can create a custom control using existing server controls, thus providing an easy way to reuse code.

Web Services

- ☑ Web services change the application architecture to provide information when and where you want it.
- ☑ Web services are components that provide service to a consumer, who uses standard Internet protocols (HTML, XML) to access these services.
- ☑ Web services are the custom business components that don't have user interfaces and are meant to be consumed by programs only.

Using Windows Forms in Distributed Applications

- ☑ .NET gives the ability to use Windows forms as a client-side user interface in a distributed application.
- ☑ Windows applications run on the client machine and use server resources remotely. They reduce the server load.
- ☑ Windows applications use .NET sophisticated graphics, so they can provide rich user interfaces with the quickest response and the highest degree of interactivity.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: Do we have to copy the code-behind a VB file to production when we are deploying an application?

A: No. The code is compiled into the DLL, which is in the bin directory, so copying the DLL is enough.

Q: Is Web.config required in the application root directory?

A: Web.config is optional and, if present, overrides the default configuration settings.

Q: What is the compilation tag in Web.config?

A: Inside Web.config, Visual Studio creates a compilation tag with an attribute debug whose value is True. This tag is used to configure the compilation settings. When the debug property is set to True, ASP.NET saves the temporary files that are helpful when debugging. For the applications in production, this property should be set to False.

Q: Why shouldn't we use the same name for Web forms that are in different folders?

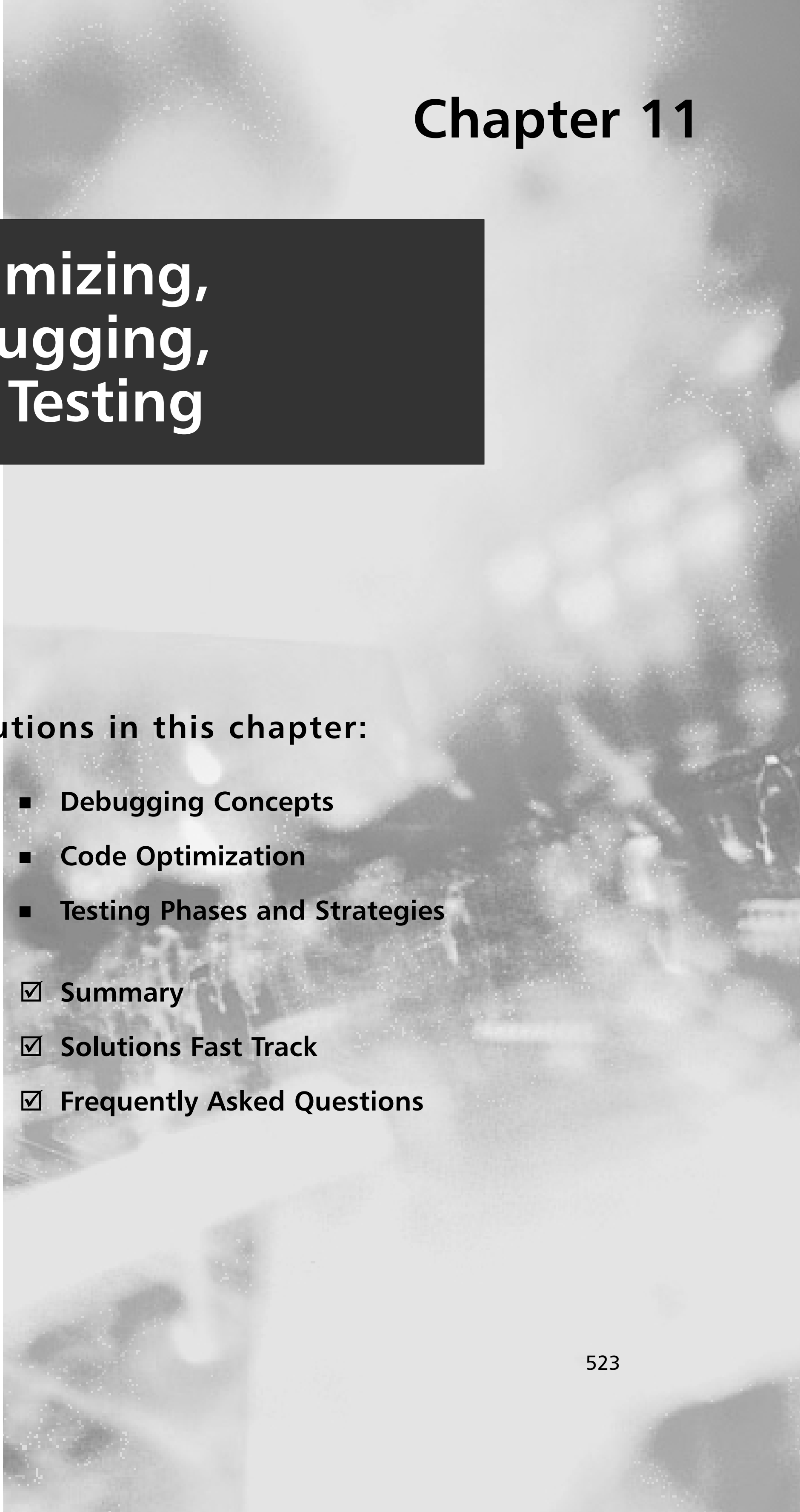
A: VS.NET uses the code-behind technique; for this reason, each Web form inherits a class in the namespace named after the Web form. In a namespace, there can be no duplicate class names. Thus no two Web forms can have the same name, even though they are in different folders.

Q: Can any client consume Web services?

A: Yes, any client that understands HTTP and XML can consume Web services.

Q: Does my current ASP code work under ASP.NET?

A: Yes, it will work. In order to support backward compatibility, Microsoft introduced a new filename (.ASPX) for ASP.NET. In order to take advantage of .NET Framework, it would be better if you could rewrite your code to ASP.NET.



Chapter 11

Optimizing, Debugging, and Testing

Solutions in this chapter:

- Debugging Concepts
- Code Optimization
- Testing Phases and Strategies

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

When developing an application, program debugging consumes a significant portion of development, and the better you understand how to use the debugging tools, the faster you can track bugs down and fix them. In this chapter, we discuss the tools available in Visual Basic .NET to assist you in debugging. You should already be familiar with some of these tools from previous versions of Visual Basic. It is important to understand the tools that are available and how to use them. Debugging will be a little different now that Visual Basic uses exceptions for runtime errors in your program.

When you release your applications, you want them to run as robustly as possible. Different aspects of program development can affect the performance of an application. Many of these concepts will be the same as in previous versions of Visual Basic, but you need to understand some new ones in order to optimize your applications. We talk about some issues in your code that can improve performance, and we also discuss some runtime performance issues and the best options to choose from when compiling your application.

Prior to releasing your applications, you should completely test them. You should not be using your customers to perform testing for you. Generally, testing is initially allocated its fair share of time. As development deadlines slip, however, the testing phase shrinks to make up for it. Most software engineers do not enjoy testing, but it is a very important part of application development. Testing involves different phases, and different personnel are needed for these phases. Independent personnel should perform the final testing because the developers understand how the program works from the inside, and it is harder for them to step back and look at it from a user perspective.

Debugging Concepts

As a developer working in any size project, there is one guarantee—there will be bugs. Bugs can come in a variety of forms, but you need to consider three main types:

- **Syntax-related** These are usually the easiest to catch, especially with advanced development environments like the one provided by Visual Basic. These occur in situations where you might misspell a reserved word or variable name.
- **Runtime errors** These occur when your code is syntactically correct (the compiler does not notice anything in error as it prepares to execute

the application), but an error occurs as the code actually executes. For example, if you were to attempt to execute a method on an object without first instantiating the object, you would get a runtime error. Unless you include some error-handling code, the application may come to a halt. These are still relatively easy to locate in an environment such as the one provided in Visual Basic .NET.

- **Logic errors** These are among the most difficult to track down. They occur when you experience unexpected behavior from your application (your program zigs when it should have zagged). These are usually a result of some logical error in the algorithms of your application. Luckily, Visual Basic provides many useful tools that aid in tracking down logic errors.

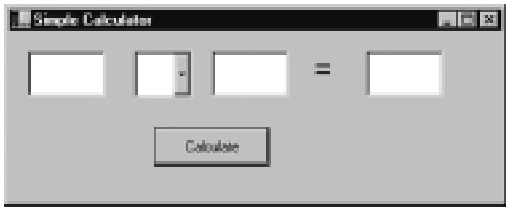
Among the tools available for debugging are watches, breakpoints, the Exceptions window, conditional compilation, and the addition of traces and assertions. We have seen most of these in previous versions of Visual Basic. But, because they do offer some new functionality, we cover each of them in their own section. In addition, the Visual Basic IDE provides a comprehensive debugging menu. First, let's set up our test project that we will use in order to practice the debugging techniques mentioned thus far. (This project and changes made throughout the chapter are included on the CD. See file Chapter 11/Simple Calculator.vbproj.)

1. Start up a session of Visual Basic .NET.
2. Select a Windows application. Make sure the application has one Windows form.
3. Place controls on the form so that the form looks like Figure 11.1. From right to left, place a **textbox1**, **combobox1**, **textbox2**, **label1**, **textbox3**, and **button1**. Set the name of the controls as listed in the Table 11.1.

Table 11.1 Simple Calculator Controls

Control	Name
textbox1	txtLeft
combobox1	cboOperation
textbox2	txtRight
label1	label1
textbox3	TxtResult
button1	button1

Figure 11.1 User Interface for Debugging Practice Project



4. Right-click on the **Forms Designer** and select **view code**. Right below the line that reads Inherits *System.WinForms.Form*, enter the following two variable declarations:

```
Private intLeftNumber  As Integer
Private intRightNumber As Integer
```

5. Enter the following code into the **New()** method of the form below the **Form1 = Me**:

```
'add available operations to combo box
cboOperation.Items.add("+")
cboOperation.Items.Add("-")
cboOperation.Items.Add(chr(247))
cboOperation.Items.Add("*")
```

6. Select **Button1** from the **Class Name** combo box at the top left of the code view pane. Then in the method name **combobox**, select the **Click()** method. In the **Button1_Click()** method, enter the following code:

```
intLeftNumber = CType(txtleft.Text, Integer)
intRightNumber = CType(txtRight.Text, Integer)
Call Calculate()
```

7. Add the code for the final routine:

```
Protected Sub Calculate()
   Dim tempResult As Integer
       'try to do the requested operation
           Try
             Select Case cboOperation.SelectedItem
```

```
     Case "+"
       tempResult = intLeftNumber + intRightNumber
     Case "-"
       tempResult = intLeftNumber - intRightNumber
     Case "*"
       tempResult = intLeftNumber * intRightNumber
     Case chr(247)
       tempResult = CType(intLeftNumber / _
           intRightNumber, Integer)
         End Select
       Catch e As Exception
     'catch any exceptions e.g. Division by zero
           tempresult = 0
       End Try
     'display the result of the operation
        txtResult.Text = CType(tempResult, String)
    End Sub
```

This completes the setup of our practice project. You should compile it to make sure that everything is in check. You will notice that VB.NET still provides color-coding of keywords and intrinsic functions. This helps to easily identify and read your code. In addition, the VB.NET IDE provides a new feature that enables you to recognize when you may have misspelled a variable name or keyed in something that it does not recognize. For example, find one of the references to the variable **intLeftNumber** in the code. Change the spelling from **intLeftNumber** to **intLeftumber**. You will notice a wavy underline appear under the word. This functionality is similar to what we are accustomed to seeing in Microsoft Word documents. It tells us immediately that there is something that it does not recognize. If you place the mouse pointer over the word, you will see Tool Tip text that gives more detail about the problem.

The example application simply performs the designated operation on two integer values. But, in keeping the example simple, we will be able to demonstrate all the beneficial features available to you when debugging your code in Visual Basic .NET.

Debug Menu

The Visual Basic .NET IDE Debug menu provides us with some very useful tools, which are very helpful for debugging in the runtime environment. Each provides a unique way to control execution of your code line-by-line. The tools include **Step Into**, **Step Out**, **Step Over**, and **Run To Cursor**. We now examine their functionality. Follow these steps:

1. Open the code view for the designer of the simple calculator. In the code, place the cursor on the line where the **Button1_Click()** method begins.
2. Place a breakpoint there by pressing **F9**. A *breakpoint* will halt execution when the compiler reaches this line of code; we cover it in greater detail in the "Breakpoints" section.
3. Run the application by selecting **Start** from the **Debug** menu.
4. When the simple calculator loads up, put any numeric value in each of the left and right text boxes. Select the **plus sign** to indicate addition in the combo box.
5. Select **Calculate**. You will notice that the execution of the program stops and that the current line where execution stands is indicated with a yellow arrow in the left margin of the code view. This yellow arrow always indicates the next line that will be executed. By using the commands in the Debug menu, we can control where the execution will go.
6. Go to the **Debug** menu now and select **Step Into**. You will see the yellow arrow move down one line, which means that the previous line executed successfully. This technique can be very useful. It helps you to follow your code one line at a time in order to determine where an error occurs or to see where a value changes.
7. Continue to select the **Step Into** command until the arrow is on the same line that calls the **Calculate()** method. At this point, the execution will move into another procedure.

   We have options here. If we know that the method in question works and is not the source of any error being investigated, then we may choose to **Step Over** the call. By selecting to **Step Over** the call, the method will be executed at real time without us seeing the execution line by line. In order to see the code in the method execute line-by-line, we must **Step Into** the method.

8. Select the **Step Into** command from the **Debug** menu. After you are in a routine, you again have two choices. You can step through the code line by line, or you can **Step Out** of the method back to the calling method. You can do this by using the **Step Out** command.
9. Select the **Step Out** command from the **Debug** menu. You will see the execution return to the calling method (**Button1_Click**) one line after the method you are returning from (the **Calculate** method). All the code in **Calculate()** is executed before returning to the calling method.

In addition to all these tools, you can also use the **Run To Cursor** option, which is handy in lengthy methods. It gives you the ability to place the cursor on a line within the current method and have the code execute up to the line where the cursor is.

NOTE

Each of the Debug menu tools has keyboard shortcuts. You can use these debugging techniques very efficiently by becoming familiar with these shortcuts:

Step Into F8
Step Over Shift+F8
Step Out Ctrl+Shift+F8
Run To Cursor Ctrl+F8

These features are very useful but are usually used along with the other useful VB.NET IDE debugging tools, which we discuss in the following sections.

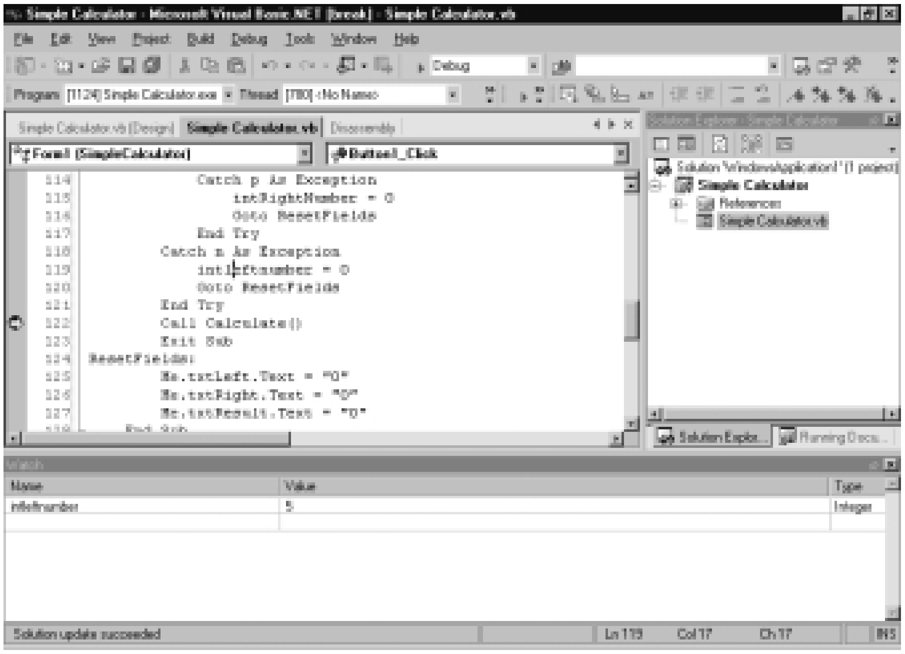
Watches

Watches provide us with a mechanism where we can interact with the actual data that is stored in our programs at runtime. They allow us to see the values of variables and the values of properties on objects. In addition to being able to view these values, you can also assign new values. This can be very handy while stepping through your code because you can see what would happen if a variable had a different value at a specific point in time. Let's use our practice project to examine the value of watches:

1. In order to use the Watch window, the application must be in break mode. So again, let's set a breakpoint in the **Button1_Click** event.

2. Run the application and enter some numbers then press the **Calculate** button so that we get to our breakpoint.
3. Place the cursor over the **intLeftNumber** variable names and select **Add Watch**. You now see a new window appear at the bottom of the IDE called the Watch window (see Figure 11.2). Now, you can see the value of the variable as it changes for as long as it is in scope. The values of variables are only visible while they are in *scope,* which means that private declarations are only visible in the classes or methods in which they are declared. Publicly declared variables will be visible throughout the application. In addition to being able to watch these values, we can also change them. Although we have the Watch window available, place the cursor into the Value field for our watch variable. Change the value to **15** and press **Enter**. Now continue execution of the program. You will see that the final result reflects the change that you made in the Watch window.

Figure 11.2 The Watch Window


Simple Calculator - Microsoft Visual Basic.NET [break] - Simple Calculator.vb
File Edit View Project Build Debug Tools Window Help
Debug
Program [1124] Simple Calculator.exe Thread [780] <No Name>
Simple Calculator.vb [Design] Simple Calculator.vb Disassembly
Form1 (SimpleCalculator)
Button1_Click
Solution 'WindowsApplication1' (1 project)
Simple Calculator
References
Simple Calculator.vb
Solution Explo... Running Docu...
Watch
Solution update succeeded
Ln 119 Col 17 Ch 17 INS

```
                Catch p As Exception
                    intRightNumber = 0
                    Goto ResetFields
                End Try
            Catch n As Exception
                intLeftnumber = 0
                Goto ResetFields
            End Try
            Call Calculate()
            Exit Sub
ResetFields:
            Me.txtLeft.Text = "0"
            Me.txtRight.Text = "0"
            Me.txtResult.Text = "0"
```

Name	Value	Type
intleftnumber	5	Integer

NOTE

You can also add variables to the Watch window by typing their names into the Name field. Or, you can also add a variable by highlighting it in the code view and then dragging it into the Watch window. You can then change and watch their values. You cannot, however, change the values of constants in the Watch window.

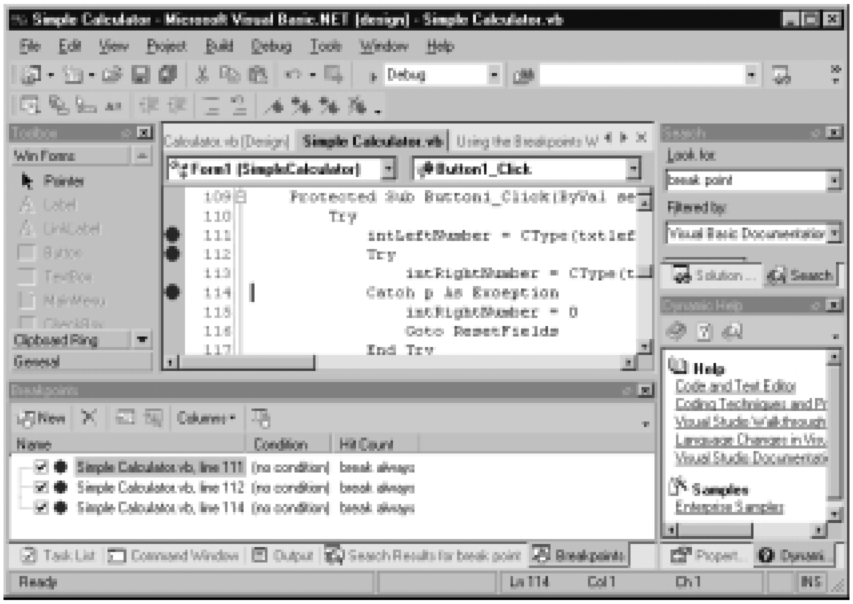
Breakpoints

We have already seen breakpoints in the earlier examples, but we visit them in more detail here. As we saw, breakpoints allow you to halt execution of your program at a specific line of code. This helps when you have narrowed down the general area of a problem you might be investigating. Breakpoints gives you the ability to halt execution just before entering into that section and then walk through the code as you desire. You can set breakpoints in a variety of ways. You can click in the left margin of the code view at the line where you want to set the breakpoint, or you can place the cursor on that line and press **F9**. In fact, you can press **F9** to toggle the breakpoint on and off. You may also set breakpoints by selecting **New Breakpoint** from the **Debug** menu.

A new feature in VB.NET is the Breakpoints window, which you can bring up by selecting **Windows** and then **Breakpoints** from the **Debug** menu. From this window, you can see a list of all the breakpoints currently set in your application (see Figure 11.3); you also have the flexibility to jump to any breakpoint in the application simply by double-clicking on it in the Breakpoints window. You will also notice that each breakpoint listed in the window has a checkbox beside it. These give the option of activating or deactivating any breakpoint without actually having to physically remove the breakpoint from the code view pane. This can be useful if you want to skip over a breakpoint one time but activate it again at a later time.

Some more advanced features are also available. Select any one of the breakpoints in the Breakpoints window and click **Properties**. Here you can set a condition for the breakpoint. Click the **Condition** button and you will see the dialog box shown in Figure 11.4. In this dialog box, we can specify any Boolean condition to determine whether the breakpoint should be enabled or disabled. If you click the **Hit Count** button from the Breakpoint properties dialog box, you can specify how many times the breakpoint must be hit before it is automatically enabled.

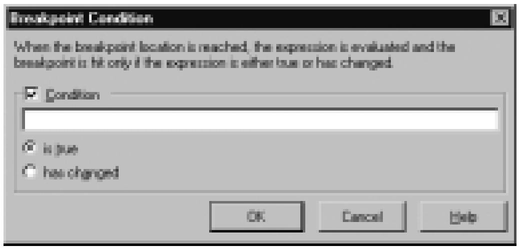
Figure 11.3 The Breakpoints Window



```
109    Protected Sub Button1_Click(ByVal se
110        Try
111            intLeftNumber = CType(txtlef
112            Try
113                intRightNumber = CType(t
114            Catch p As Exception
115                intRightNumber = 0
116                Goto ResetFields
117            End Try
```

Name	Condition	Hit Count
Simple Calculator.vb, line 111	(no condition)	break always
Simple Calculator.vb, line 112	(no condition)	break always
Simple Calculator.vb, line 114	(no condition)	break always

Figure 11.4 Setting a Breakpoint Condition



Breakpoint Condition

When the breakpoint location is reached, the expression is evaluated and the breakpoint is hit only if the expression is either true or has changed.

☑ Condition

[✓ is true] has changed

Exceptions Window

The Exceptions window is new in Visual Basic .NET. In previous versions of Visual Basic, we were able to tell the compiler what to do when it encountered errors. You might remember the options were **Break In Class Module**, **Break On All Errors**, or **Break On Unhandled Errors**. In the new Exceptions window, you can tell the compiler what to do on any specific exception or class of related exceptions. Let's take a walk through using our simple calculator application:

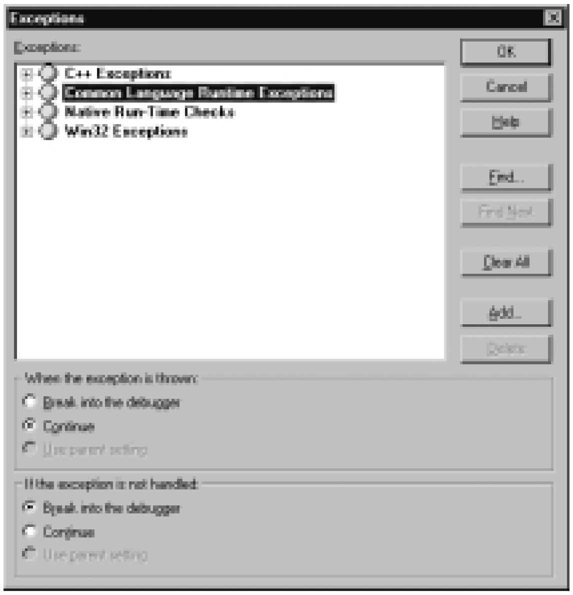
1. Remove all breakpoints (remember that you can do this with the key combination **Ctrl+Shift+F9**).

2. Run the application by selecting **Start** from the **Debug** menu.
3. Place an alpha character in the first textbox and select to multiply it by any number in the right text box.
4. Select **Calculate**.

You will see that an exception is thrown in the *Button1_Click()* method. This exception (System.FormatException) occurred because, as we know, we cannot convert a string with alpha characters into an integer. We have two ways we can handle this. We can use a *Try…Catch…Finally* block and handle the exception, or we can configure how the compiler should handle this exception. This type of configuration is done in the Exceptions window as shown in Figure 11.5. For the type of exception we are encountering here, we recommend that you use a *Try…Catch…Finally* block and handle the exception appropriately. However, we examine the functionality of the Exceptions window here. Let's begin by finding the exception in the exceptions TreeView:

1. Expand the **Common Language Runtime Exceptions** node.
2. Expand the **SystemException** node.
3. Scroll down the list until you see the **FormatException** and select it.

Figure 11.5 The Exceptions Dialog Box



When the exception is thrown:
( ) Break into the debugger
(•) Continue
( ) Use parent setting

If the exception is not handled:
(•) Break into the debugger
( ) Continue
( ) Use parent setting

The bottom of the Exceptions window has two frames. Each one gives you an option as to how to handle the exception at different points in time. The top frame labeled **When The Exception Is Thrown** tells the compiler what to do immediately after the exception is thrown but before the code to handle the exception is executed. The second frame labeled **If The Exception Is Not Handled** tells the compiler what to do if the exception is not handled or if the code to handle the exception fails.

Each of the two frames has three options: **Break Into The Debugger**, **Continue**, and **Use Parent Setting**. By selecting **Break Into The Debugger**, you are telling the compiler to go into break mode as soon as the exception is thrown. You can set this up so that it will break only on unhandled exceptions by selecting **Continue** in the top frame and **Break Into The Debugger** in the second frame. The second option is **Continue**. By selecting this, you are telling the compiler just to continue execution when an exception is thrown. If you were to select this in both frames, you would be telling the compiler to ignore the exception all the time. The final setting is **Use Parent Setting**, which is the default setting. It allows for a form of inheritance through the hierarchy of exceptions. For example, if you change the settings for Common Language Runtime Exceptions, all exceptions below that node that are set to Use Parent Setting will inherit those changes.

NOTE

You can also add your own custom exception classes to the Exceptions window. The Use Parent setting can be very useful when you develop your own exception classes. If your exception classes have child classes, they can inherit the behavior from their parent class with the use of this setting.

Command Window

When debugging in Visual Basic 6.0, we made extensive use of the Immediate window. We were able to use the Immediate window in order to determine the values of variables and to execute commands in the IDE while the program is in debug mode. Much of this functionality has been retained in Visual Basic .NET with a tool called the Command window. This window is available in two modes. The first mode is the immediate mode; you can open the command window in immediate mode by selecting **Immediate** from the **Windows** submenu on the Debug menu. The second mode is the Command mode; you can open the Command mode by selecting the **Command Window** option from the **Other Windows** menu found under the **View** menu. You can switch between the two modes by using the **immed** command in Command mode and using the **>cmd** command while in Immediate mode. You can use both of these modes to execute commands while running in debug mode.

When working in command mode, the window's title bar will read *Command Window*. While in command mode, you can execute commands using the command line instead of locating them in the menus or executing commands that are not available in the menus. For example, by typing **addproj** you can add a new project to the Solutions Explorer. Most of the options that are available for use in the Command window also have aliases. The aliases allow you to use a short form, preventing the tedious act of having to type out the long name every time. An example of this is **bl**, which is an alias name for **Debug.ToggleBreakPoint**. A full list of aliases is available from MSDN, but the most commonly used ones are listed in Table 11.2.

Table 11.2 Common Command Window Commands

Alias	Long Name	Description
?	Debug.Print	Writes the value of an expression or variable to the output window.
??	Debug.QuickWatch	Adds an expression or variable to the Quick Watch.
Alias	Tools.Alias	Creates a custom Alias name for a command.
Bl	Debug.Breakpoints	Displays the Breakpoints window.
Bp	Debug.ToggleBreakPoint	Toggles a breakpoint on the current line.
Callstack	Edit.Callstack	Writes out the callstack.
Clearbook	Edit.ClearBookmarks	Clears all the bookmarks.
Code	View.ViewCode	Displays the code pane for the current designer view.
Designer	View.ViewDesigner	Displays the designer pane for the current code view.
Cmd	View.CommandWindow	Display the command mode.
Immed	Tools.ImmediateMode	Display the Command window in immediate mode.

As we can see from just these examples, a very comprehensive list of commands is available for use in the Command window. Mastering the use of these shortcuts will cut down the cost and time involved in debugging your projects. Although you can use the Command mode of the Command window for debugging tasks such as evaluating expressions and validating the values of variables, we recommend that this be done in the Immediate mode of the Command window. The Immediate mode functions exactly the way that the Immediate mode from previous versions of Visual Basic did except for one point. The Up/Down arrow keys do not move up and down through the lines of the Command window, but they actually scroll through the list of previously issued commands on the one line. Just as in previous versions, you determine values with the **?** and evaluate expressions simply by typing them in and pressing **Enter**. While in Immediate mode, you can execute all the commands available in the Command mode by prefixing the command mode commands with a **>** character.

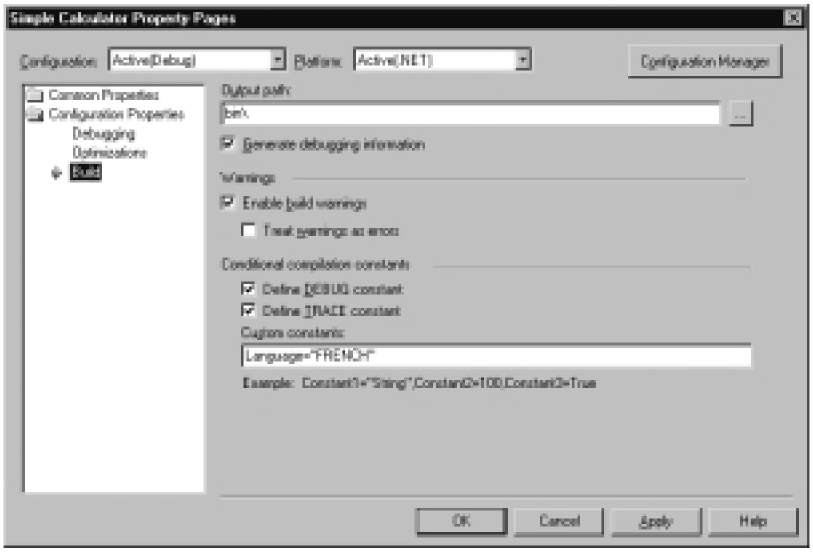
Conditional Compilation

Oftentimes during development, multiple versions of an application are required. A common driving factor is regionalization of the application. Also, a large amount of debugging code often exists throughout the source code for an application. This code can add up to quite a few lines and it would be very time consuming and tedious if we had to remove this code before compiling the application and then put it back in so that we could continue development on other features for the application. *Conditional compilation* provides a way for us to leave all this extra code in our source and to easily regionalize our applications. Any source code that is enclosed with conditional compilation may or may not be compiled into the executable file.

You can declare variables to be used for conditional compilation in a variety of ways. Most of the directives for declaring conditional compilation variables are much the same as they were in previous versions of Visual Basic. The first method is to set up all conditional compilation variables in the project properties dialog box (see Figure 11.6):

1. Open the Solution Explorer.
2. Right-click the project for which you would like to set up conditional compilation constants and select **Properties**.
3. Select **Configuration Properties**.
4. Select **Build** and then add or modify conditional compilation constants.

Figure 11.6 Project Properties Dialog Box: Conditional Compilation



Configuration: Active(Debug)
Platform: Active(.NET)
Output path: bin\
☑ Generate debugging information
Warnings
☑ Enable build warnings
☐ Treat warnings as errors
Conditional compilation constants
☑ Define DEBUG constant
☑ Define TRACE constant
Custom constants: Language="FRENCH"
Example: Constant1="String",Constant2=100,Constant3=True

You can also declare variables in code. You declare the variable as a constant by using the **#Const** keyword. This syntax tells the compiler that this variable is to be used to evaluate conditional compilation expressions. You can also pass in variable definitions as arguments when compiling from the command line by using the **/define** tag. To evaluate the value of conditional compilation variables, we use the **#if…#else…#End if** construct. Here are some code examples for using conditional compilation using these two techniques:

```
#Const Language = "FRENCH"
#If language = "FRENCH" then
     'Do french code
#Else
     'Do other language code
#end if
```

We can also have the exact same **#if** statement in the code but pass in the definition of the *Language* variable using the tag on the command line. For example, **Vbc /define:Language=FRENCH [project]**. In addition to the standard conditional compilation options that we have seen here, Visual Studio .NET provides some built-in conditional compilation options that enable you to actually trace the execution of a deployed application.

Trace

For applications where performance is vital, it would be convenient to be able to trace their performance as the end users were using them. Visual Studio.NET provides a mechanism that allows us to do just that. We can place code throughout our application that can log events and steps of execution to a console window, log file, or any other available output. For example, we may want to examine database connectivity in a multiuser environment in order to determine why some users are experience longer lags then others.

In Figure 11.6, we saw how to define conditional compilation constants. The dialog box also has two other checkboxes. The first checkbox is where you set the Debug constant, and the second is where you set the *Trace* constant. As in Visual Basic 6.0, we have access to a debug class. In previous versions of Visual Basic, we had the ability to write to the Immediate window by using lines in our code, such as **debug.print**. This class is still available in Visual Basic .NET (though the method names have changed, the overall functionality is similar). One difference is that unless the Debug constant is set, all the debugging code that is written in your application using the *Debug* class will not be compiled into the final executable. On the other hand, if you set the *Trace* option, all the debugging code that is written in your application using the *Trace* class will be compiled into the executable. It is through this mechanism that we can add *Trace* functionality to our deployed applications.

The information that you wish to have logged can be written to a few different places. These places are called *Trace Listeners.* The Listeners collection of the Trace object keep track of the Trace Listeners. Two other potential Listeners are the *TextWriterTraceListener* and the *EventLogTraceListener*. The *EventLogTraceListener,* as you might guess, will write the Trace information to an event log. The *TextWriterTraceListener* writes the information to any instance of the *TextWriter* class or *Stream* class. In the upcoming example, we use an instance of the *Stream* class to create a text file to track the operations and numbers that our users use. This information might be useful if they report bugs with the application.

Let's take a look at using traces in our simple calculator program. Add the following Code to the simple calculator window's form:

1. At the top of the *Form* class, enter the following declaration:

```
Private myfile As System.IO.Stream
Private BoolSwitch As BooleanSwitch
```

2. Add this initialization code to the bottom of the **New()** method:

```
 'set up the trace listener
Boolswitch = New BooleanSwitch("General", _
    "Simple Calculator")
myfile = system.IO.File.Create("C:\TraceFile.txt")
Trace.Listeners.Add( _
New TextWriterTraceListener(myfile))
boolswitch.Enabled = True
```

3. Add the following code to the top of the **Calculate()** method:

```
 'trace operation if trace is enabled.
trace.WriteLineIf(boolswitch.Enabled, _
    CType(intleftnumber, String) & _
    CType(cbooperation.selecteditem, String) & _
      CType(intRightNumber, String))
```

4. Lastly, select the **Finalize** method from the method name drop-down for the form and add this code:

```
Protected Overrides Sub Finalize()
'close the Trace listener
myfile.close()
  End Sub
```

Now, run the application and do some operations. When you have completed a few calculations, navigate to where you created the text file and open it. You will see all the operations that you did logged in the text file. In this case, we are writing every operation to the text file regardless of any factors. This may not be optimal for all applications because using the Trace mechanism does come with some overhead (as minimal as it may be). The *Trace* class offers other functionality that allows you to categorize your messages into different levels and Trace them only if that level of tracing is on. You may have noticed in the previous example that we used the *Trace.WritelineIf* method. This method will only write the Trace if the first parameter of the method evaluates to True. It is useful in this parameter to pass the Switch object to the *Trace* method in order to allow the trace method to determine if Trace is on or not.

Two *Switch* classes are available. The first one is called the *BooleanSwitch* (as we saw earlier). This switch is used merely to toggle *Trace* on or off by using the Enabled property. The other switch class is called the *TraceSwitch*. This class allows us to set our Traces to different levels. By setting the level of *Trace* that applies to a specific **Trace** statement, you can limit what items are written to your *Trace Listener*. This allows you to decide in your analysis what type of information a particular *Trace* is providing. This type of *Trace* has five settings: Off (no Trace), Error, Warning, Info, and Verbose. These options allow you more control over what information is logged and when that information gets logged.

Assertions

When debugging a project, it can sometimes help to narrow down situations where you know certain criteria must be met or certain expressions must be True in order for an algorithm to execute correctly. *Assertions* allow us to test for these conditions in our applications. In previous versions of Visual Basic, we had access to the *Assert* method of the Debug object. This is still the case in Visual Studio .NET (the *Assert* method is also available on the Trace object).

Just as before, the *Assert* method will evaluate an expression that must evaluate to a Boolean value. If the expression evaluates to False (the assertion fails), execution of a program will halt. The functionality of the *Assert* method has been extended in Visual Studio .NET. The method has three different overrides (all three result in a message box to the user). The message box gives the user the option to Abort, Retry, or Ignore. The first parameter in all three overrides is the expression to evaluate. The first override accepts only one parameter, and if the expression evaluates to False, it writes the call stack out to a message box. The next override takes two parameters. The second parameter is a short message. Here you can provide a short message to be displayed in the message box instead of the Call Stack. The last override allows up to three parameters. It also allows a short message as the second parameter, but it also accepts a more detailed message as the third parameter. This is useful if you would like to provide more detail to the user. The Method Signatures for *Debug.Assert* is:

```
1. debug.Assert(Condition as Boolean)
2. debug.Assert(Condition as Boolean, Message as String)
3. debug.Assert(Condition as Boolean, Message as
       String,detailmessage as String)
```

The user of assertions is more beneficial during the debugging process. Using assertions in conjunction with conditional compilation is a good idea. This would allow you to leave the code in the project while in Debug mode but also ensure that it is not compiled into the final executable.

Debugging…

Attach the Debugger to an External Process

A wonderful new feature in Visual Studio .NET is the ability to attach the debugger engine to an external process. This can be any application either on the local machine or a remote machine that has been compiled with debug information or that you have access to the source code for. You can use the debugger to debug applications that may or may not have been created in Visual Studio. You can also enable a just-in-time debugger. The JIT debugger invokes as a program crashes in order to assist us in finding the faulty code. To attach the debugger to an external program, follow these simple steps:

1. Choose **Processes** from the **Debug** menu.
2. In the **Available Processes** list box, select the application you would like to attach the debugger to.
3. Make sure to select the appropriate type of process.
4. Click **OK.**

Now, if you want to put the application into break mode, all you have to do is click **Break** in the dialog box. This is a really nifty feature in Visual Studio .NET. It will enable us to make our executable files, run them, and then attach the debugger to them so that we can find any offending code the first time an error occurs as opposed to having to re-create the error in the IDE.

Code Optimization

Given the gigahertz processors and low-priced memory on the market, optimizing code is often overlooked in the applications we develop today. Granted, optimization may not offer the same noticeable improvements it might have in days past, but it is still a worthwhile practice. Optimizing our applications can provide for a faster, scalable, more maintainable, and robust application. Nevertheless, issues such as the object model we choose, late binding versus early binding, and cleaning up at the end of our routines still fall in the hands of the developer and all influence how optimal our applications are.

Finalization

The *Finalize* method is comparable to the *Class_Terminate* method of objects in Visual Basic 6.0. It gives you a chance to do any additional cleanup tasks that are required before the object is destroyed (such as release database connections). The *Finalize* method is resource intensive and can slow down the performance of the application. When you implement the *Finalize* method, the object will take longer to remove from memory. In addition, objects are not deallocated in any particular order and there is no guarantee that your object will be *Finalize*d. In order to optimize performance, override the *Close()* method and call the *Close* method when you are done with the object. To ensure that the *Finalize* method does not get invoked (because we are using the *Close* method, the object does not need to be finalized), it can be suppressed in the *Close* method with a call to *GC.SupressFinalize*.

Transitions

When we invoke methods on unmanaged code (such as .DLLs or other COM components) we cause transitions to occur. The *transition* is what .NET invokes in order to communicate with the unmanaged code. It is comparable to marshalling data across process boundaries. When this is done, we take a performance hit. Every time a transition is invoked it brings with it some overhead (about 10 to 40 extra instructions). In order to optimize code that requires the use of transitions, organize your code such that you can accomplish as much as possible with as few calls as possible.

Parameter Passing Methods

In Visual Basic 6.0, values were passed by reference by default. What this meant is that a pointer to the address location in memory of the parameter was passed to function calls. Unless we explicitly declared the parameter to be passed by value, it would be passed by reference. When a parameter is passed by value, a copy of the data is passed into the receiving routine. We still have the two parameter passing types in Visual Basic .NET (the default when passing arguments is now by value). Passing the intrinsic datatypes (Integers, Singles, Char) by value does have some performance advantages. This is because the Value types such as these are allocated on the stack where they are accessed and removed quickly with very little overhead. Objects passed by reference are allocated on the heap, which requires interaction with the Garbage Collector and in turn generates greater overhead.

Strings

Strings in Visual Basic .NET are objects. But the difference is that you cannot modify a String object. That is to say they are immutable. Traditionally, if you modified a string, you actually modified the specific string. Now, when you modify a string (perhaps replace the middle character with something else), what is actually happening behind the scenes is a new String object is being created, and the old one is discarded. This can mean a lot of overhead for intense string operations. Instead of working with Strings directly, we can use the StringBuilder object (*System.Text.StringBuilder*). This object eliminates the overhead of creating new strings when modifying the value of a String object. For example, let's take a look at some code:

```
Dim strTemp1 as String
Dim strTemp2 as String
strTemp1 = "Hello"
strTemp2 = " Out There"
strTemp1 = strTemp1 & strTemp2
```

In this code example, even though we declared only two String objects, three will be created. The first one is created for *strTemp1*. The second is created for *strTemp2*. The third String object is created when we perform the concatenation of the *strTemp1* and *strTemp2*. Internally, a new String object is created and assigned the result of the concatenation. The pointer to the third String object is then assigned to *strTemp1*. If we had a large amount of String operations, we can see how the overhead would add up quickly. Now take a look at this segment:

```
Dim strTemp1 As String
Dim y As New StringBuilder("Hello")
strTemp2 = " Out There"
y.Append(strTemp2)
```

In this code, we can see that we are still creating two objects. One is our String; the other is the *StringBuilder*. What is different here is that the *StringBuilder* will concatenate the strings using its *Append* method without creating the third object. If we had a lot of string manipulations, we can see how we would save a lot of overhead by using the *StringBuilder* object.

Garbage Collection

When you are developing in a distributed architecture, the Garbage Collector works differently than how it was described in the earlier chapters. When objects are created from a remote class, they will acquire a lease time. The Garbage Collector will decide whether or not it is time to clean up the object based on this lease time. The benefit of this is that distributed objects also used to be destroyed with reference counting in the same way that reference counting was used in the desktop application.. This can carry high overhead with respect to network traffic pinging back and forth in order to keep count of the references. However, in .NET, the Garbage Collector will respond when the lease time expires. So, when the lease time expires, the Garbage Collector now knows that it is safe to clean up the object and does so. It is still good practice to explicitly destroy your objects when you are done with them when you are using a distributed architecture by making them equal to nothing.

Compiler Options

Compiler options have long been a way of optimizing code. The compiler options we choose can help reduce the size or our applications and increase performance if done in the right circumstances. Visual Basic .NET has a plethora of compiler options. Most of these options are available only from the command line. The following sections explore some of the more significant options.

Optimization Options

The */optimize* compiler option deals with optimization of your compiled application. It makes your application file smaller, faster, and more efficient. This option is on by default. In order to toggle this feature on and off, you use */optimize*+ and */optimize*- where the + turns it on and the – turns it off.

Output File Options

The */out* option allows you to specify the name of the output file that is generated by the compiler. The */target* option lets you specify the type of application to create. By setting */target:exe*, you generate an .exe console application, which is the default setting. In addition, you can set this to */target:module*, */target:library*, and */target:winexe*. These create a code module, a code library, and a Windows application respectively.

.NET Assembly Options

All of the options in this category allow you to modify assemblies. The */keycontainer* option allows you to specify the originator of an assembly. The */keyfile* option enables you to specify a file with a key pair to make a shareable component. Another option, */nostdlib*, tells the compiler not to include the two standard libraries: Microsft.VisualBasic.dll and Mscorlib.dll. These two files define the entire system namespace. If you were to create your own namespaces, you would want to use this option so as to not include the System namespace with your package. The */reference* option allows you to import metadata from a file that contains an assembly. Finally, the */version* option lets you create an assembly and modify the version.

Debugging…

Error-Checking Options

The debugging and error-checking options that we have seen in previous versions of Visual Basic included Favor Pentium Pro and disable array bounds checks. In Visual Studio .NET, the options have changed, with four options available. First is the */bugreport* option, which will generate a bug report consisting of items such as the files that were included in the compile, all the compiler options that were being used, and (but not limited to) the version information of the compiler. Next, we have the */cls* option. This is used to turn off (*/cls*-) or turn on (*/cls*+) whether the compiler checks for the Common Language Runtime specification. Also, we have the */debug* option. Use this option to generate builds that can be debugged. This will include the extra information required by the debugger in order to debug an executable program. Finally, we can use the */removeintchecks* option. This option can be turned on, again, by using the + or – after the option. By turning this option on, you tell the compiler to ignore overflow checks and division-by-zero type errors. Turn this on only if your application has been tested thoroughly, and you know you will not encounter any of these bugs.

Preprocessor Options

We have already seen the use of the preprocessor option. This is the */define* option. It simply allows us to declare and initialize conditional compilation variables on the command line.

Miscellaneous Options

A wide variety of miscellaneous options allow you to do different things from the command line. We cover a few of the more significant ones here:

- ***/?* and */help*** These options display help associated with the command that precedes it.
- ***/baseaddress*** Lets you assign the baseaddress of a DLL file. Doing this can improve the performance speed when the file is loaded into memory. When a DLL file is first loaded into memory, it will try to load into the first available block. If enough consecutive space isn't available for the DLL to fit, it will keep searching until it finds a big enough space. By changing the base address of the DLL, you can specify an area of memory to begin loading in order to save the time of the DLL trying to find enough space for itself.
- ***/optionexplicit***, ***/optioncompare***, and ***/optionstrict*** All of these do the same thing that they would do if they were declared explicitly in the code of the application. The */optionexplicit* option ensures that all variables are declared before they are used. The */optioncompare* option sets the method of comparing strings in the application (either binary or text). Finally, the */optionstrict* option forces strict data type usage.

Testing Phases and Strategies

Debugging is a very useful skill to have. More important, though, is having the skills to find the bugs. No matter how good your development team or program designer is, your programs will always have bugs when the system goes out the door. This is why one of the most important phases of development is the testing phase. More often than not, the testing phase gets the least attention. As functional requirements increase and as obstacles surface during development, the development phase gets dragged out into the testing phase without restructuring deliverable dates. As a result, testing usually gets the short end of the stick.

Nevertheless, by having a thorough and stringent testing strategy, you can still uncover and correct many of the bugs in a software project. The following sections outline the different phases of testing.

Unit Testing

Unit testing is the most basic of all the testing phases. When a developer receives a specification to complete an aspect of the system, the developer also has an idea as to how this aspect should function. Before the task is determined by the programmer to be completed, they will usually perform some tests on it to ensure that the application does indeed generate the desired results. Unit testing is considered the lowest level of testing.

Integration Testing

Integration testing becomes important on large teams or teams of subteams where different individuals or different groups of people are each responsible for different functional aspects of a system. For example, in an accounting system, one group may be responsible for user interface design, another for the ledger system, and yet another for the report generation. In this case, each group would be responsible for certifying that their components work properly when they are standing alone. But, will they work when they are integrated into the package? This is the purpose of *integration testing.* You determine how the independently developed components behave when they are all integrated together.

Beta Testing

After a package is put together, and the application is certified by the internal departments to be fully functional and behave correctly, beta testing takes place. *Beta testing* is where the product is put out in a prerelease format for users to use in actual live production. This environment is where the unthinkable usually happens. This is where a user will inevitably attempt to accomplish a task by doing the entire process backward and upside down. The fresh eyes of people outside the development and testing teams often are able to catch even the most minor things that have been overlooked. The purpose of beta testing is not to give your customers a product and for them to let you know whether or not it actually works. Beta software is usually distributed free in some form of a demo mode or time-limited evaluation copy so that your product has the opportunity to be fully tested in a live environment before customers spend their hard earned dollars.

Regression Testing

Another fact of software development is that what the client wants now is not going to be the same thing that the client wants six weeks from now. Needless to say, change requests will occur after the product goes live. When these changes are implemented (at a premium, of course!), some other aspects of the system will probably be affected. This is the purpose of *regression testing.* As changes are implemented, we must retest all areas of the system that may have been affected by the changes.

Stress Testing

It is good practice, before applications are released, to determine how much they can take. This is called *stress testing.* An example of this is an application where you allow people to update their address information online. At any one time, the average number of users might be 5,000. But, what if 25,000 people try to use your application at one time? How would your application react? And, what if there were now 75,000 concurrent users? This is the concept behind stress testing. You push your application beyond logical limits to see what it can handle before grinding to a halt. This gives you a very good picture as to how scalable your application is. If you do anticipate that as time goes on there will be more clients and the potential for more concurrent users, this testing will tell you how far you can go before you either have to throw more hardware at it or rearchitect your application.

Monitoring Performance

The ability to monitor performance has come a long way. With Windows 2000 (and Windows NT 4.0), we can use the Performance Monitor (see Figure 11.7) to help us track the performance of our applications. To launch the performance monitor, simply type **perfmon** on the command line.

From this dialog box, if you select **System Monitor** you will see a chart that will display the current performance of the machine. Each item that the monitor is tracking is called a *counter.* To add counters to the chart, you can right-click in the empty grid at the bottom and select **Add Counters**. You will see a dialog box such as the one shown in Figure 11.8, where you can select the performance object that you would like to monitor. Take a look at the list. Of particular interest is Memory and Processor. By watching counters for these two items, you can see exactly what type of stress your application is putting on the system. In order to get clarification on what exactly any one counter is, you can select it from the list of counters and then select **Explain**. A box will appear at the bottom of the dialog box with an explanation of what the counter is doing.

Figure 11.7 Windows Performance Monitor

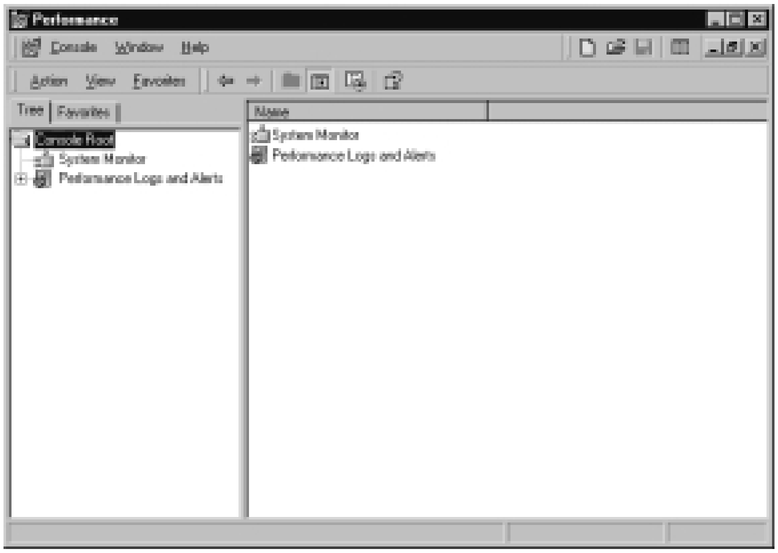
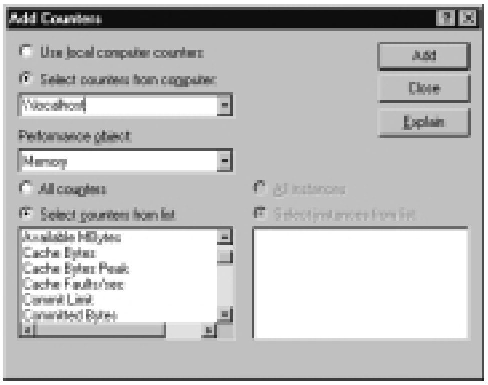


Figure 11.8 Add Counters Dialog



Use local computer counters [✓ Select counters from computer:] \\localhost

Performance object: Memory

All counters [✓ Select counters from list]

Available MBytes
Cache Bytes
Cache Bytes Peak
Cache Faults/sec
Commit Limit
Committed Bytes

Summary

A lot of competition exists in the world of software development. Not only for products on the market but for the bragging rights of who can put out the most robust application available. By mastering the concepts outlined in this chapter, your development team can be well on its way to achieving those rights. By reducing the amount of time it takes to locate bugs and fix them, you can significantly reduce the amount of time it takes to get your application out. Visual Basic .NET provides us with some strong tools to help make that possible. We covered these tools, and they are worth mentioning again here. By becoming proficient with the tools on the Debug menu and their associated shortcuts, combined with the effective use of watches, breakpoints, traces, conditional compilation, and assertions, we can become very proficient developers.

In addition to enhancing debugging techniques in order to decrease overall development time, it is equally important to ensure that ample time is allotted for a concise and complete testing strategy. By doing this, we make sure that we catch all of our bugs (or at least as many as is humanly possible) before the product goes out to paying customers. After paying customers receive a product with mistakes in it, reputations begin to dwindle. Developers need to do thorough unit testing during development and integration testing as the different components are brought together. Then, you need to beta test your application with people who are independent of the entire development process. From the beta testing, changes will likely have to be made. In order to be thorough in applying these changes, you should be sure to conduct regression testing in order to ensure that all affected components still function the way they are supposed to. Finally, before shipping your application to paying customers, put it through an exhaustive stress test. This will help you target weaknesses in the code and application that may become obstacles when you require your application to scale.

After stress testing your application, you may notice areas of weakness. This is a good time to go through the code and identify any areas where you can further optimize it. As we mentioned, you might be able to do this by passing arguments by value wherever possible, reducing the amount of transitions required, using the StringBuilder object whenever possible for string manipulation, and finalizing your objects properly without counting on the Garbage Collector to sweep by at any specific point in time. After all this is verified, you can monitor your application's performance by using tools such as the Windows Performance Monitor. When you are satisfied with these results, look into what kind of compiler options are available in order to make your executable smaller and faster.

Solutions Fast Track

Debugging Concepts

☑ Debugging is one of the most important aspects of development that we should attempt to master.

☑ The Visual Basic .NET IDE provides a very rich set of features that enable us to become effective debuggers.

☑ The most useful debugging techniques involve using watches, breakpoints, conditional compilation, and traces.

Code Optimization

☑ Passing arguments by value will have less overhead.

☑ Reducing the number of transitions in our code will increase performance.

☑ Properly cleaning up our objects when they are destroyed will prevent abandoning resources such as database connections that are not managed by the Garbage Collector.

Testing Phases and Strategies

☑ Testing consumes the largest portion of the time allocated for software development projects.

☑ Ensure that testing is done by others outside the development team in order to ensure that the maximum number of bugs are caught before delivery of the product.

☑ A solid testing strategy comprised of all of the different phases of testing will help you deliver quality, robust software.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: Can I use **Run To Cursor** to set the next executable line of code to be in a different method than the one that the yellow arrow indicator is currently in?

A: No, you can use the **Set Next Statement** and **Run To Cursor** commands only within the current method. If you wish, you may use the **Step Into** command in order to place execution into the method and then use **Run To Cursor**.

Q: Can I change the settings for the *TraceSwitch* at runtime?

A: Yes, you can provide the user with menu options to change the level of Trace at runtime. You can do this in reflection of the type of problem you may be troubleshooting.

Q: Are all the compiler options available from the Visual Studio IDE?

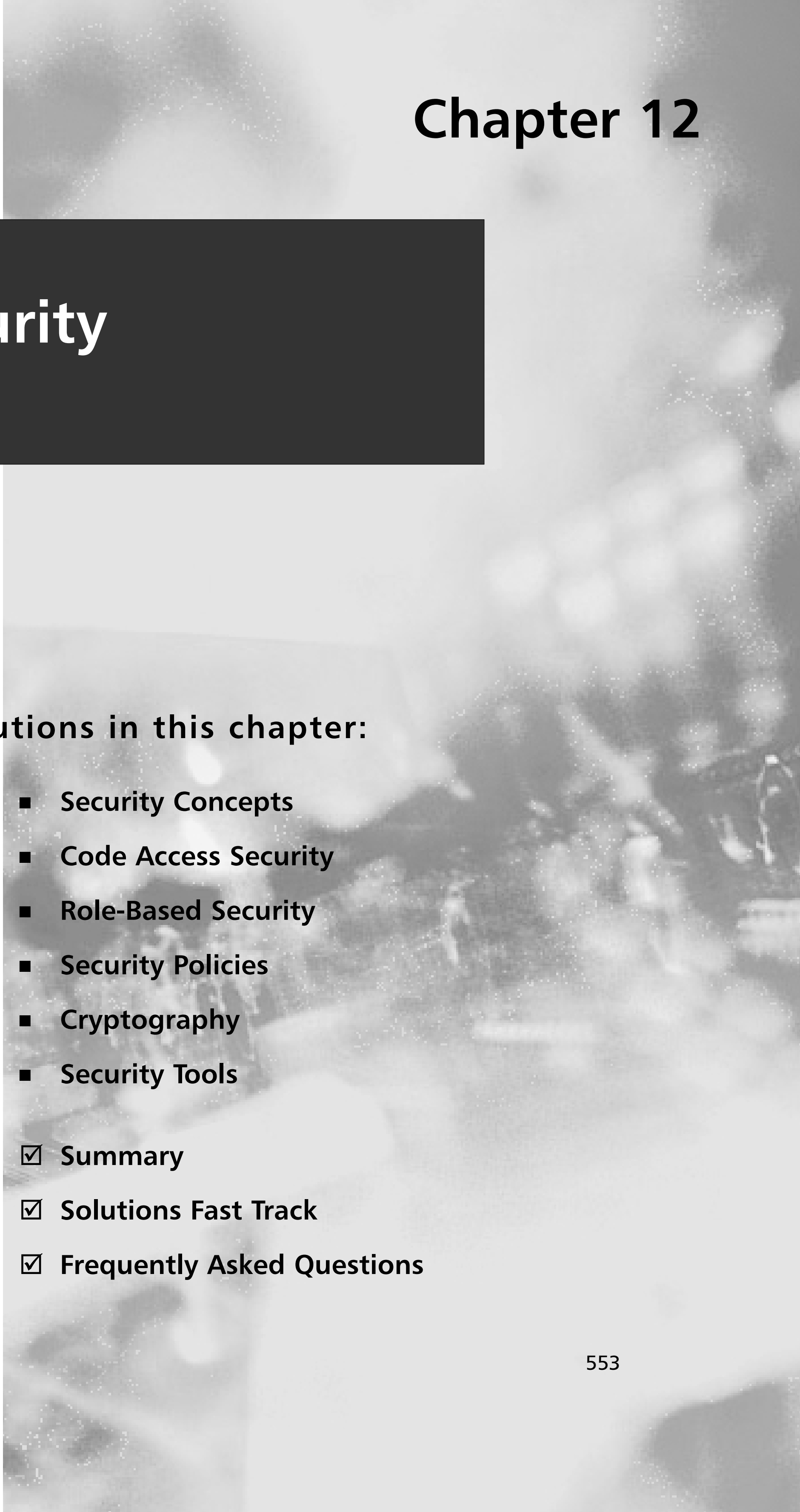
A: Unfortunately, no. A large number of the compiler options are available only on the command line.

Q: Can the debugger be attached only to processes developed in .NET?

A: No, you can attach the debugger to any process (even MS Office applications). But, the value of the debugging that you are able to do will depend on how the application was compiled. It would be necessary for the application to be compiled with debug information, or you would need to have access to the source code.

Chapter 12

Security



Solutions in this chapter:

Introduction

Security is already an increasing concern for businesses. The .NET Framework is designed to allow for distributed applications across the Internet. This concept introduces a slew of security risks. Microsoft realizes these risks and has introduced new security functionality that is incorporated in the .NET Framework. This chapter is not meant to completely cover implementing security but rather to show you the functionality that is available and how to use it.

Some of the security concepts are the same as before. You will still authenticate users prior to allowing them on the system. You will continue to use permissions and rights for user access to specific objects on the system and authentication of users are always required. This type of security is fine for systems that are physically disconnected from the Internet. With connections to the Internet, one of the concerns is for mobile code. Mobile code is code that can be executed and can come from sources outside your network. This could come from e-mail attachments, from code embedded in documents, or from code that you download from Web sites. As many of you have seen, sometimes this code can be malicious. One important mechanism that is introduced with .NET can help with this type of problem is *code access security* (CAS), which prevents mobile code from accessing sensitive resources by allowing permissions to be granted to code, or code demanding certain permissions from the caller of the code. This means that a group of code cannot access the resource unless it has the proper permissions. Another security feature is *role-based security*. Roles are generally established for types of functionality. This does not always map to typical network user accounts. Roles can be created for specific applications and their requirements. This allows threads to execute with the permissions of a designated role. This was available in the past, but .NET has extended this to both the client and the server.

The Common Language Runtime (CLR) also has security features. You can create security policies that determine what code is allowed to do. This allows administrators to restrict access to resources and the rules are enforced at runtime. Some security features are included in the Security namespace of the System object. The Cryptography namespace allows you to encode and decode data as well as other functions to support data encryption. This allows for the development of data encryption with an easy to use, object-based methodology. Security will increasingly become an important part of application development, and this chapter will help you understand the features that are available and when to use them.

Security Concepts

As we discuss in the following sections, code access security and role-based security are the most important vehicles to carry the security through your applications and systems. However, let it be clear that we are not discussing VB.NET security, but .NET security. That is, the security defined by the .NET Framework and enforced by the CLR. Since the .NET Framework namespaces make full use of the security, every call to a protected resource or operation when using one of these namespaces, automatically activates the CAS. Only if you start up the CLR with the security switched off, Code Access Security will not be activated. The CLR is able to "sandbox" code that is executed, preventing code that is not trusted from accessing protected resources or even from executing at all. This is discussed more thoroughly in the "Code Access Security" section later in this chapter. What is important to understand is that you can no longer ignore security as a part of your design and implementation phase. Not only is it a priority to safeguard your systems from malicious code, but you also want to protect your code/application from being "misused" by less-trusted code. For example, let's say that you implement an assembly that holds procedures/functions that modifies Registry settings. Because these procedures/functions can be called by other unknown code, these can become tools for malicious code if you do not incorporate the .NET Framework security as part of your code. To be able to use the .NET Security to your advantage, you need to understand the concepts behind the security.

Permissions

In the real world, permission refers to an authority giving you, or anybody else for that matter, the formal "OK" to perform a specified task that is normally restricted to a limited group of persons. The same goes for the meaning of permission in the .NET Security Framework: getting permission to access a protected resource or operation that is not available for unauthorized users and code. An example of a protected resource is the Registry, and a protected operation is a call to a COM+ component, which is regarded as unmanaged code and therefore less secure. The types of permissions that can be identified are the following:

- **Code access permissions** Protects the system from code that can be malicious or just unstable; see the "Code Access Security" section for more information.

- **Role-based security permissions** Limits the tasks a user can perform, based on the role(s) he plays or the identity he has; see the "Role-Based Security" section for more information.
- **Identity permissions** Limits the access based on the rights the user is given in the Windows 2000 environment. For example, an administrator identity will have more permissions than a default user. See the "Role-Based Security" section for more information.
- **Custom permissions** You can create your own permission in any of the other three types, or any combination of them. This demands a thorough understanding of the .NET Framework security and the working of permissions. An ill-constructed permission can create security vulnerabilities.

You can use permissions through different methods:

- **Requests** Code can request specific permissions from the CLR, which will only authorize this request if the assembly in which the code resides has the proper trust level. This level is related to the security policy that is assigned to the assembly, which is determined on the base of evidence the assembly carries. Code can never request more permission than the security policy defines; such a request will always be denied by the CLR. However, the code can request less permission. What exactly security policy and evidence consist of is discussed over the course of this chapter.
- **Grants** The CLR can grant permissions based on the security policy and the trustworthiness of the code, and it requests code issues.
- **Demands** The code demands that the caller has already been granted certain permissions in order to execute the code. This is the security part you are actively responsible for.

Principal

The term *principal* refers directly to the role-based security, being the security context of the executed code. Based on the identity and role(s) of the caller, whether it is a user or other code, a principal is created. In fact, every thread that is activated is assigned a principal that is by default equal to the principal of the caller. Although we just stated that the principal holds the identity of the caller, this is not entirely correct, because the principal has only a reference to the callers identity, which already exists prior to the creation of the principal. Three types of principals can be identified:

- **Windows principal** Identifies a user and the groups it is a member of that exists within a Windows NT/2000 environment. A Windows principal has the ability to impersonate another Windows user, which resembles the impersonate you may know from the COM+ applications.
- **Generic principal** Identifies a user and its roles, not related to a Windows user. The application is responsible for creating this type of principal. Impersonation is not a characteristic of a general principal, but because the code can modify the principal, it can take on the identity of a different user or role.
- **Custom principal** You can construct these yourself to create a principal with additional characteristics that better suits your application. Custom principals should never be exposed because doing so may create serious security vulnerabilities.

Authentication

In general, *authentication* is the verification of a user's identity, hence the credentials he hands over. Because the identity of the caller in the .NET Framework is presented through the principal, the identity of the principal has to be established. Because your code can access the information that is available in the principal, it can perform additional authentication tests. In fact, because you can define your own principal, you can also be in control over the authentication process. The .NET Framework supports not only the two most-used authentication methods within the Windows 2000 domain—NTLM and Kerberos V5.0—but also supports other forms of authentication, such as Microsoft Passport. Authentication is used in role-based security to determine if the user has a role that can access the code.

Authorization

Once a user has been authenticated, the system is able to determine the *authorization* the user has to perform specific tasks. In the case of the .NET Framework, this is done base on the identity of the principal. Authorization in relation to roles has to be part of the code and can take place at every point in the code. You can use the user and role information in the principal to determine if a part of the code can be executed. The permissions the principal is given, based on its identity, determine if the code can access specific protected resources.

Security Policy

To be able to manage the security that is enforced by the CLR, an administrator can create new or modify existing security policies. Before an assembly is loaded, its credentials are checked. This evidence is part of the assembly. The assembly is assigned a security policy depending on the level of trust, which determines the permissions the assembly is granted. The setting of security policies is controlled by the system administrator and is crucial in fending off malicious code. The best approach in setting the security policies is to grant no permissions to an assembly of which the identity cannot be established. The stricter you define the security policies, the more securely your CLR will operate. The CD contains the User Security PPolicy file, both in its original form and with the changes disscussed in this chapter. See the Chapter 12 folder on the CD.

Type Safety

A piece of code is labeled *type safe* if it only accesses memory resources that do belong to the memory assigned to it. Type safety verification takes place during the JIT compilation phase and prevents unsafe code from becoming active. Although you can disable type safety verification, it can lead to unpredictable results. The best example is that code can make unrestricted calls to unmanaged code, and if that code has malicious intent, the results can be severe. Therefore, only fully trusted assemblies are allowed to bypass verification. Type safety can be regarded as a form of "sandboxing."

Code Access Security

The .NET Framework is based on the concept of distributed applications, in which an application does not necessarily have a single owner. To circumvent the problem of which parts of the application (being assemblies) to trust, code access security is introduced. This is a very powerful way of protecting the system from code that can be malicious or just unstable. Remember that it is always active even if you do not use it in your own code. CAS helps you in:

- Limiting access permissions of assemblies by applying security policies
- Protecting the code from obtaining more permissions than the security policy initially permits

- Managing and Configuring permission sets within security policies to reflect the specific security needs
- Granting assemblies specific permissions that they request
- Enabling assemblies in demanding specific permissions from the caller
- Using the callers identity and credentials to access protected resources and code

.NET Code Access Security Model

The .NET code access security model is built around a number of characteristics:

- Stack walking
- Code identity
- Code groups
- Declarative and imperative security
- Requesting permissions
- Demanding permissions
- Overriding security checks
- Custom permissions

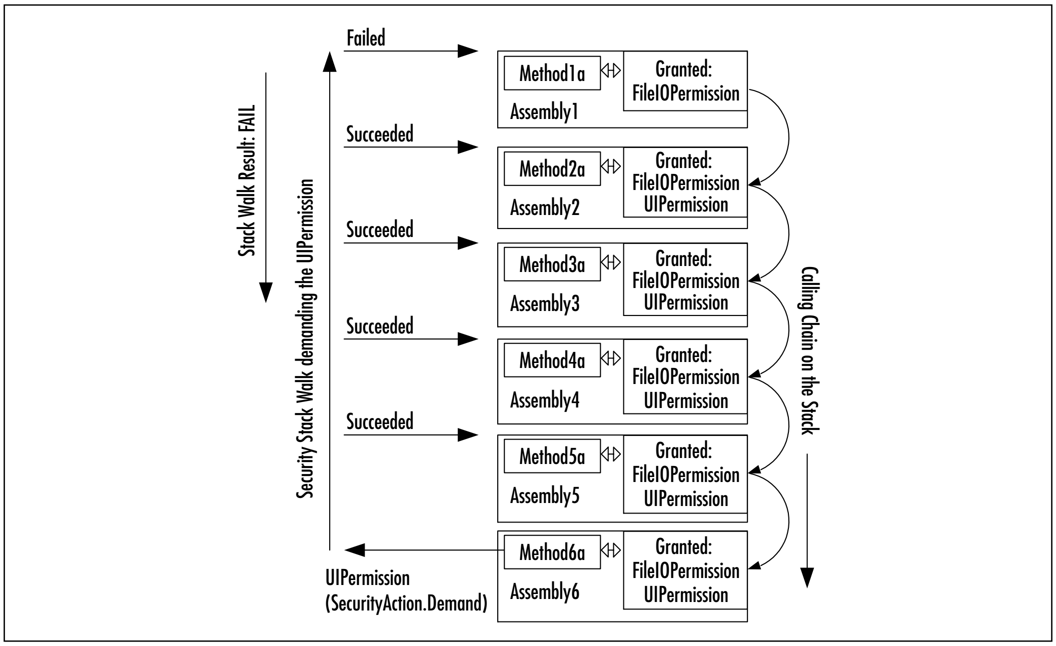
By discussing these characteristics, you will get a better understanding how CAS not only works, but also can work for you during the design and implementation of applications.

Stack Walking

Perhaps *stack walking* is the most important mechanism within CAS to ensure that assemblies cannot gain access to protected resources and code during the course of the execution. As mentioned before, one of the initial steps in the assembly load process is that the level of trust of the assembly is determined, and corresponding permission sets are associated with the assembly. The total package of sets is the maximum number of permissions an assembly can obtain.

Because the code in an assembly can call a method in another assembly and so forth, a *calling chain* develops (see Figure 12.1) with every assembly having its own permissions set. Suppose that an assembly demands that its caller have a specific permission (in the figure that is *UIPermission*) to be able to execute the method. Now the stack walking of the CLR kicks in. The CLR starts checking the stack where every assembly in the calling chain has its own data segment. Going back in the stack, every assembly is checked for the presence of this demanded permission, in our case *UIPermission*. If all assemblies have this permission, the code can be executed. If, however, somewhere in the stack an assembly does not have this permission (in our case this is in the top assembly *Assembly1*), the CLR throws an exception, and access to the method is refused.

Figure 12.1 Performing Stack Walking to Prevent Unauthorized Access


Stack Walk Result: FAIL
Security Stack Walk demanding the UIPermission
Failed
Succeeded
Succeeded
Succeeded
Succeeded
Method1a
Assembly1
Granted: FileIOPermission
Method2a
Assembly2
Granted: FileIOPermission UIPermission
Method3a
Assembly3
Granted: FileIOPermission UIPermission
Method4a
Assembly4
Granted: FileIOPermission UIPermission
Method5a
Assembly5
Granted: FileIOPermission UIPermission
Method6a
Assembly6
Granted: FileIOPermission UIPermission
UIPermission (SecurityAction.Demand)
Calling Chain on the Stack

Stack walking prevents calling code from getting access to protected resources and code for which it initially does not have the authorization. You can conclude that at any point of the calling chain the effective permission set is equal to the intersection of the permission sets of the assemblies involved.

Even if you do not incorporate the permission demand in your code, stack walking will take place because all class libraries that come with the CLR make use of demand to ensure the secure working of the CLR. The only drawback of stack walking is that it can have a serious performance impact, especially if the calling chain is long. Suppose the stack contains 8 assemblies and the top assembly makes a call to a method that demands a specific permission and does so in a 200-fold loop. After executing the loop, 200 Security Stack Walks have been triggered. Since each stack walk performs 8 security checks, the total number of security checks is 1,600.

Code Identity

The whole principle of the .NET Framework security rides on *code identity,* or to what level a piece of code can be trusted. The code identity is established based on the evidence that is presented to the CLR. Evidence can come from two sources:

- Evidence that is incorporated in the assembly, and put in there during the coding and subsequent compiling of the code, or which can later be added to the assembly.
- Evidence that is provided by the host where the assembly resides. The CLR controls the accepting of host evidence, through the security permission *ControlEvidence*, that should be granted only to trusted hosts.

Table 12.1 shows the default evidence that can be used to determine to what code group code belongs. Because you cannot control the identity of the assembly, you are never sure how reliable this evidence is, except for the signatures provided.

Table 12.1 The Available Default Types of Evidence

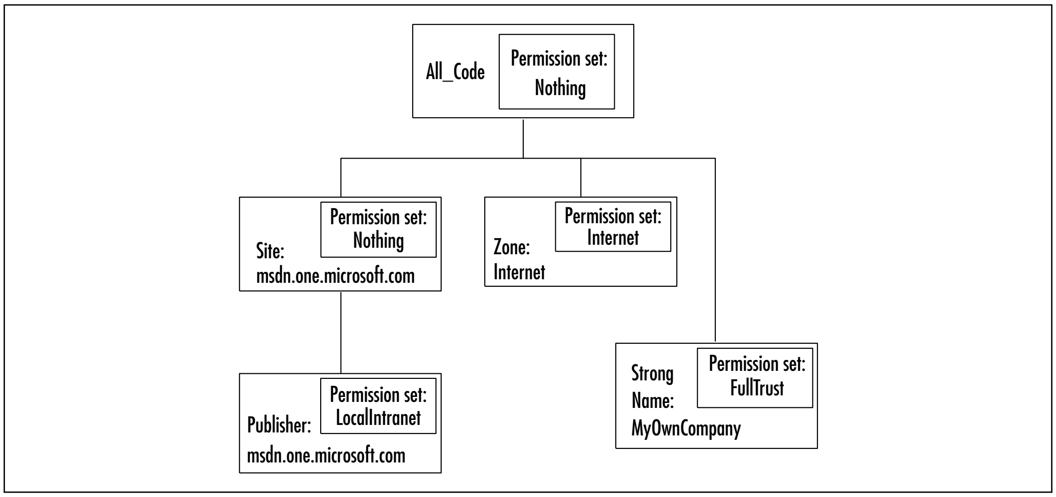
Evidence	Description
Directory	The directory where the application, hence assembly, is installed.
Hash	The cryptographic hash that is used in the code of the code: MD5 or SHA1 (see the "Cryptography" section).
Publisher	The signature of the assembly's owner, in the form of a X.509 Certificate, set through Authenticode.
Site	The name of the site the assembly originates from, for example: www.company.com (prefixes and suffixes are disregarded).
Strong name	The strong name consists of the assembly name (given name), public key (of the publisher), version numbers, and culture.
URL	The full URL, also called code base, including prefix and suffix: https://www.company.com:4330/*.
Zone	The zone where the assembly originates. Default zones are Internet, Local Intranet, My Computer, No Zone Evidence, Trusted Sites, and Untrusted (Restricted) Sites.

The more evidence you can gather about the assembly, the better you can determine to what extent you can grant it permissions. The strong name is of great importance. If you and all other serious application developers are very persistent in providing assemblies with strong names, you can prevent your code from becoming the vehicle of somebody's dubious intents. Sadly enough, malicious code can still have a convincing Strong name. That is why the best evidence is the certificate and signature that should be present with the assembly. Once you have established the trustworthiness of an assembly, based on all the evidence before you, you can determine the appropriate permission sets. Here is where your realm of control starts, by constructing appropriate code groups.

Code Groups

A *code group* can be defined as a group of assemblies that share the same value for one, and only one, piece of evidence, called *membership condition*. Based on this evidence, a permission set is attached to the assembly. Because a code group is part of a code group hierarchy (see Figure 12.2), an assembly can be part of more code groups. The effective permission set of the assembly is the union of the permissions sets of the code groups it belongs to.

Figure 12.2 Graphical Representation of a Code Group Hierarchy


All_Code
Permission set: Nothing
Site: msdn.one.microsoft.com
Permission set: Nothing
Zone: Internet
Permission set: Internet
Publisher: msdn.one.microsoft.com
Permission set: LocalIntranet
Strong Name: MyOwnCompany
Permission set: FullTrust

When an assembly is about to be loaded, the evidence is collected and the code group hierarchy is checked. When the assembly is matched with a code group, the CLR will check its child code groups. This implies that the construction of the hierarchy is very important and must be built starting with the general evidence items—for example, starting with zone and moving on to more specific ones such as publisher. A complicating factor is that there are three security levels (Enterprise, Machine, and User), with their own code group hierarchy. All three are evaluated, resulting in three permission sets, which at the end are intersected, thereby determining the effective permission set.

It is the administrator's responsibility to construct code group hierarchies that can quickly be scanned and enforce a high level of security. To do so, you must take several factors into account:

- Limit the number of levels.
- Use membership conditions at the first level that are highly discriminatory, preventing large parts of the hierarchy from being checked.
- The hierarchy's root, All Code, should have no permissions assigned, so code that does not contain at least some evidence is not allowed to run.
- The more convincing the evidence, for example the publishers certificate, the more permissions that can be granted.
- Make no exceptions or shortcuts by giving out more permissions than the evidence justifies. Assume that you have a specific application, running in the intranet zone, that needs to have full trust to operate. Because it is your own application, you implicitly trust it, without the factual evidence. If you do this, however, it can come back to haunt you.

Table 12.2 shows the available default membership conditions. You can construct your own, but that is beyond the scope of this chapter. Membership conditions are discussed in more detail in the "Security Policy" section.

Table 12.2 Default Membership Conditions for Code Groups

Membership Condition	Description
All Code	Applies to every assembly that is loaded
Application directory	Applies to all assemblies that reside in the same directory tree as the running application, hence the Application domain
Hash	Applies to all the assemblies that use the same hash algorithm as specified or have the specified hash value
Publisher	Applies to all assemblies that carry the specified publishers certificate

Continued

Table 12.2 Continued

Membership Condition	Description
Site	Applies to all assemblies that originate from the same site
Skip verification	Applies to all assemblies that request the Skip Verification permission. WARNING: This permission allows for the bypassing of type safety. Use it only at the lowest level after you have established that the code is fully trusted
Strong name	Applies to all assemblies that have the specified strong name
URL	Applies to all assemblies that originate from the specified URL, including prefix, suffix, path, and eventual wildcard
Zone	Applies to all assemblies that reside in the specified zone
(custom)	Applies to custom-made conditions that are normally directly related to specific applications

Declarative and Imperative Security

You are provided with two ways of adding security to your code. This can be a demand that callers have a specific permission or a request for a specific permission from the CLR.

The first method is *declarative security*, which can be set at assembly, class, and/or member level, so you can demand different permissions at different places in the assembly. At the member level (a Class or Method), the demand for a permission will only take place if this part of the code is actually called. The VB.NET syntax of declarative code is **<[assembly:]*Permission*(SecurityAction .*Member*, *State*)>**, for example:

```
<assembly: FileIOPermission(SecurityAction.Demand, Unrestricted :=
    True)>
<FileIOPermission(SecurityAction.Request, Unrestricted := True)>
```

The first security example is valid for the whole assembly; hence every call in this assembly needs to have the FileIOPermission. The Second example can be used for a Class or a single Method. Only a reference to a class or a call of the method will request the CLR for FileIOPermission.

As the syntax already suggests, by using **<>** this code is not treated as ordinary code. In fact, as you compile the code to an assembly, these lines are extracted and placed in the metadata part of the assembly. This metadata is checked at different points, such as during the load of the assembly or when a method in the assembly is called. Using declarative security, you can demand, request, or even override permissions before the code is even executed. This gives you a powerful security tool during the development of the code and assemblies. However, this means that you must be aware of the kind of permissions you need to request and/or demand for your code.

The second method is *imperative security,* which becomes a part of your code and can make permission demands and overrides. It is not possible to request permissions using imperative security because that makes it unclear at what point a specific permission is needed and at what point it is no longer needed. That is why permission requests are related to identifiable units of code. You may want to use imperative security to check if the caller has a permission that is specific for a part of the code. For example, just before a conditional part of the code (this may even be triggered by the role-based security) wants to access a file or a Registry key, you want to check if the caller has this *FileIOPermission* or *RegisteryPermission*. The VB.NET syntax of the imperative security is code looks like this:

```
Dim PermissionObject as New Permission()
PermissionObject.Demand()
```

Here is an example:

```
Dim CheckPermission as New FileIOPermission()
CheckPermission.Demand()
```

The permission object is valid only for the scope on which it is declared, and it will be automatically discarded at the time the code returns to a higher scope. During this scope, imperative security demands and overrides overrule the permissions demanded with a declarative security statement.

Having discussed declarative and imperative security, it is time to take a look at how you can use this to request, demand, and override permissions.

Requesting Permissions

Requesting permissions is the best way to create a secure application and prevent possible misuse of your code by malicious code. As mentioned before, based on the evidence an assembly hands over to the CLR, and then a permission set is determined, using security policies. These security policies are constructed independently from the permissions an assembly needs. Of course, if you fully trust an assembly, you can grant it all the permissions it needs. An assembly can be granted more permissions than it actually needs. Requesting permissions is not asking for more permissions than you are granted, based on the security profile, but refraining from granting permissions the code does not need. By now you have probably started to wonder what the use of requesting permissions is if the security policy decides what permissions are available to the assembly. The term *available* implies two issues:

- If an assembly requests more permissions than it is granted, based on the security policy, it will not be loaded and/or the code will not be executed. Instead, the CLR will throw an exception
- If an assembly requests less permissions, it protects itself from misuse of these additional permissions somewhere up or down the calling chain.

Requesting permissions is a characteristic of proper .NET applications and demands from the developer a good understanding of the use of permissions related to the code he writes. Because you can only request permissions by using declarative security, you can first write and test the code and then add the permission requests later. This can make the development process easier, saving you the hassle of constantly having to consider permission requests for unfinished code.

There are three types of permission requests:

- **RequestMinimum** Defines the permissions the code absolutely needs to be able to run. If the *RequestMinimum* permission is not part of the granted permission set, the code is not allowed to run.
- **RequestOptional** Defines the permissions the code may not necessarily need to be able to run but may need in certain circumstances. If the *RequestOptional* permission is not part of the granted permission set, the code is still allowed to run, however, you need the code to be able to handle the situation in which the permission is needed but not granted, thus handling exceptions.
- **RequestRefuse** Defines the permissions the code will never need and which should not be granted to the assembly. By refraining from certain permissions you prevent malicious code or unstable code from misusing these permissions.

After the code is completed and you compile assemblies, you should get in the practice of making a minimum, optional, or refuse request for *every* permission (as listed in Table 12.3), based on the permissions needed by the code. Eventually you can make it more specific to relate it to classes or members. Besides the fact that you can create secure assemblies, it is also a good way of documenting the permissions related to your code.

Table 12.3 The Default Permission Classes Derived from the *CodeAccessPermission* Class

Permission Class	Permission Type	Description
DirectoryServicesPermission	Resource	Controls access to the *System.DirectoryServices* classes
DnsPermission	Resource	Controls access to the DNS servers on the network
EnvironmentPermission	Resource	Controls access to the user environment variables
EventLogPermission	Resource	Controls access to the event log services
FileDialogPermission	Resource	Controls access to files that are selected through an Open File… dialog
FileIOPermission	Resource	Controls access to files and directories
IsolatedStorageFilePermission	Resource	Controls access to a private virtual file system related to the identity of the application or component
MessageQueuePermission	Resource	Controls access to the MSMQ services
OleDbPermission	Resource	Controls access to the OLE DB data provider and the data sources associated with it
PerformanceCounterPermission	Resource	Controls access to the performance counters of Windows 2000 (or NT)
PrintingPermission	Resource	Controls access to printers
ReflectionPermission	Resource	Controls access to metadata types

Continued

Table 12.3 Continued

Permission Class	Permission Type	Description
RegistryPermission	Resource	Controls access to the registry
SecurityPermission	Resource	Controls access to *SecurityPermission* such as Assert, Skip Verification, and Call Unmanaged Code
ServiceControllerPermission	Resource	Controls access to services on the system
SocketPermission	Resource	Controls access to socket that are needed to set up or accept a network connection
SqlClientPermission	Resource	Controls access to SQL server databases
UIPermission	Resource	Controls access to UI functionality, such as Clipboard
WebPermission	Resource	Controls access to an Internet-related resource
PublisherIdentityPermission	Identity	Permission is granted if the evidence publisher is provided by the caller
SiteIdentityPermission	Identity	Permission is granted if the evidence site is provided by the caller
StrongNameIdentityPermission	Identity	Permission is granted if the evidence strong name is provided by the caller
UrlIdentityPermission	Identity	Permission is granted if the evidence URL is provided by the caller
ZoneIdentityPermission	Identity	Permission is granted if the evidence zone is provided by the caller

Now let's look at some examples of the different types of requests:

```
<assembly: SecurityPermissionAttribute(SecurityAction.RequestMinimum, _
      Flags := SecurityPermissionFlag.ControlPrincipal)>
```

In order for this assembly to run, it needs at least the permission to be able to manipulate the principal object. This is a permission you would give only to an assembly that you trust.

```
<assembly: SecurityPermissionAttribute(SecurityAction.RequestMinimum, _
     ControleEvidence : = True)>
```

In order for this assembly to run, it needs at least the permission to be able to provide additional evidence and modify the evidence as provided by the CLR. This is a powerful permission you would give only to fully trusted assemblies.

```
<FileIOPermissionAttribute(SecurityAction.RequestOptional, _
     Write := "C:\Test\*.cfg")> Public Class ClassAct
```

The *ClassAct* class requests the optional permission to be able to write to files in the C:\Test directory with the extension .cfg. If the security policy permits *FileIOPermission*, this restricted request is given. If the *FileIOPermission* is not granted, then any subsequent write to a CFG file in C:\Test will fail.

```
<assembly: FileIOPermission(SecurityAction.RequestRefuse, Unrestricted
    := True)>
```

The assembly refuses the *FileIOPermission*, even if the security policy grants this permission. If you used this request in combination with the previous example, and the security policy grants *FileIOPermission*, only *ClassAct* will get this restricted *FileIOPermission*, and the rest of the code in the assembly will not have any *FileIOPermission*.

```
<assembly: FileIOPermission(SecurityAction.RequestRefuse, _
     All := "C:\Winnt\System32\*.*")>
```

The assembly refuses only *FileIOPermission* to the access of files in the C:\Winnt\System32 directory. If the security policy grants this permission, the assembly can access all files, except for the one in the stated directory.

Instead of making requests for every code access permission, you can also request one of the following named permission sets: *Nothing*, *Execution*, *Internet*, *LocalIntranet*, *SkipVerification*, and *FullTrust*. You can do this by issuing the following request:

```
<assembly: PermissionSetAttribute(SecurityAction.RequestMinimum, _
    Name := NamedPermissionSet)>
```

Another way of requesting more code access permissions in one statement is by using XML-coded permission sets:

```
<assembly: PermissionSetAttribute(SecurityAction.RequestMinimum, File
    := "Filename.xml")>
```

Demanding Permissions

By demanding permissions, you force the caller to have a specific permission it needs to execute the code. If the caller has this request, it is very likely that he obtained it by requesting it at the CLR. As we discussed before, a permission demand triggers a security stack walk. Even if you do not perform these demands yourself, the .NET Framework classes will. This means that you should never perform permission demands related to these classes, because they will take care of those themselves. If you do perform a demand, it will be a redundant one and only add to the execution overhead. This does not mean that you should ignore it; instead, when writing code, you must be aware of which call will trigger a stack walk and make sure that the code does not encourage a surplus of stack walks. However, when you build your own classes that access protected resources, you need to place the proper permission demands, using the declarative or imperative security syntax.

Using the declarative syntax when making a permission demand is preferable to using the imperative syntax, because the latter may result in more stack walks. There are, of course, cases that are better suited for imperative permission demands. For example, if a Registry key has to be set under specific conditions, you will perform an imperative *RegistryPermission* demand just before the code actually is called. This also implies that the caller can lack this permission, which will result in an exception that the code needs to handle accordingly. Another reason why you want to use imperative demands is when information is not known at compile time. A simple example is *FileIOPermission* on a set of files whose names are only known during runtime because they are user-related.

Two types of demands are handled differently than previously described. First, the *link demand* can be used only in a declarative way at the class or method level. The link demand is performed only during the JIT compilation phase, in which it is checked if the calling code has sufficient permission to link to your code. A security stack walk is not performed because linking exists only in a direct relation between the caller and code being called. The use of link demands can be helpful to methods that are accessible through reflection. The link demand will not only perform a security check on code that obtains the *MethodInfo* object, hence performing the reflection, but the same security check is performed on the code that will make the actual call to the method. The following two examples show a link demand at class and at method level:

```
<SecurityPermissionAttribute(SecurityAction.LinkDemand, _
                     Unrestricted := True)> Public Class ClassAct


Public Shared Function _
    <SecurityPermissiobAttribute(SecurityAction.LinkDemand)> Act1()
    As Integer
        ' body of the function
End Function
```

The second type of demand is *inheritance demand*, which can be used at both the class and method level, through the declarative security. Placing an inheritance demand on a class can protect that class from being inherited by a class that does not have the specified permission. Although you can use a default permission, it makes sense to create a custom permission that must be assigned to the inheriting class to be able to inherit from the class with the inheritance demand. The same goes for the class that inherits from the inheriting class. For example, let's say that you have created the *ClassAct* class that is inheritable, but also has an inheritance demand set. You have defined your own inherit permission *InheritAct*. Another class called *ClassActing* wants to inherit from your class, but because it is protected with an inheritance demand, it must have the *InheritAct* permission in order to be able to inherit. Let's assume that this is the case. Now there is another class called *ClassReacting* that wants to inherits from the class *ClassActing*. In order for *ClassReacting* to inherit from *ClassActing*, it also needs to have the *InheritAct* permission assigned. The inheritance demand would look like this:

```
<InheritActAttribute(SecurityAction.InheritanceDemand)> Public Class
    ClassAct
```

The inheritance demand at method level can be the following:

```
Public Overridable Function
    <SecurityPermissionAttribute(SecurityAction.InheritanceDemand)>
     Act1() as Integer
       ' Body of the function
End Function
```

Overriding Security Checks

Because stack walking can introduce serious overhead and thus performance degradation, you need to keep stack walks under control. This is especially true if they do not necessarily contribute to security, such as when a part of the execution can only take place in fully trusted code. On the other hand, your code has permission to access specific protected resources, but you do not want code that you call to gain access to these resources—so you want to have a way of preventing this. In both cases, you want to take control of the permission security checks, hence overriding security checks. You can do this by using the following security actions: *Assert*, *Deny*, and *PermitOnly* (meaning "deny everything but").

After the code sets an override, it can undo this override by calling the corresponding *Revert* method, respectively *RevertAssert*, *RevertDeny* and *RevertPermitOnly*. Get in the practice of first calling the *Revert* method before setting the override because performing a revert on a nonexisting override has no effect.

WARNING

You can place more than one override of the same type, for example *Deny*, within the same piece of code. However, this is not acceptable to the CLR. If during a stack walk the CLR encounters more than one of the same asserts it throws an exception, because it does not know which of the overrides to trust. If you have more than one place in a piece of code where you set an override, be sure to revert the first one before setting the new one.

Assert Override

When you set an assert override on a specific permission, you force a stack walk on this permission to stop at your code and not continue to check the callers of your method.

WARNING

If you use an assert, you inadvertently create a security vulnerability, because you prevent the CLR from completing security checks. You must convince yourself that this vulnerability cannot be exploited.

The use of *Assert* makes sense in the following situations:

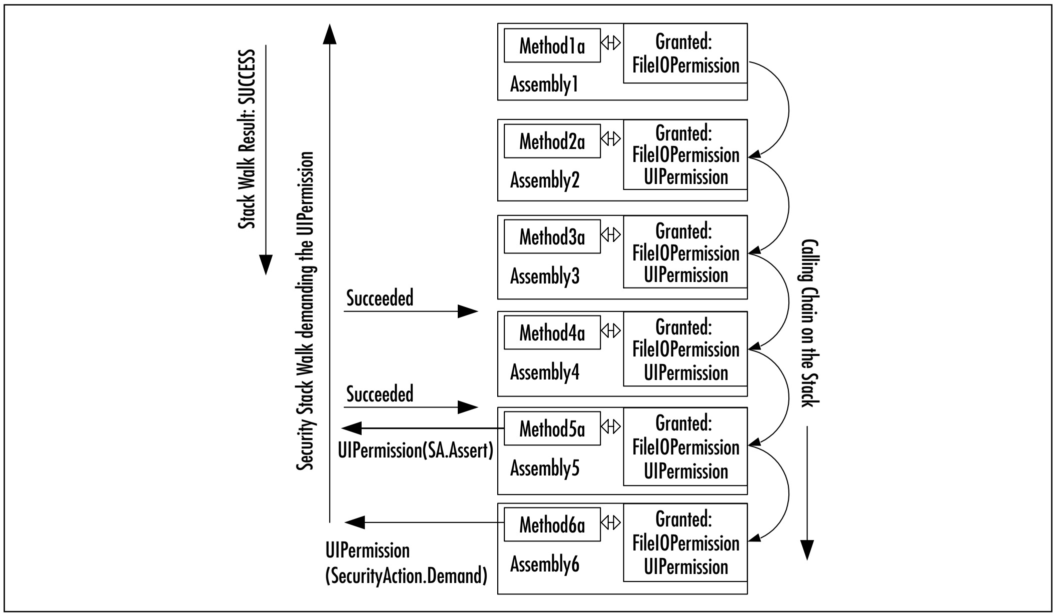
- You have coded a part of an application that will never be exposed to the outside world. The user of the application has no way of knowing what happens within that part of the application. Your code does need access to protected resources, such as system files and/or Registry keys, but because the callers will never find out that you use these protected resources, it is reasonably safe to set an *Assert* to prevent a full security check from being performed. You do not care if the caller has that permission or not.
- Your code needs to make one or more calls to unmanaged code, but because the caller of the code obtains access through your Web site, you are safe in assuming that they will not have permissions to make calls to unmanaged code. On the other hand, the callers cannot influence the calls you make to unmanaged code. Therefore, it is reasonably safe to assert the permission to access unmanaged code.
- You know that somewhere in your code you have to perform a search, using a **Do..Loop** structure that at one point has to access a protected resource. You also know that the code that calls the protected resource cannot be called from outside the loop. Therefore, you decide to set an assertion just before the call to the protected resource, to prevent a surplus of stack walks. In case the particular piece of code that does the call to the protected resource can be called by other code, you have to move up the assertion to the code that can only be called from the loop.

Let's take a look at the stack walk that was initially used in Figure 12.1, but now we throw in an assertion and see what happens (see Figure 12.3). The assert is set in *Assembly4* on the *UIPermission*. In the situation with no assert, the stack walk did not succeed because *Assembly1* did not have this permission. Now the stack walk starts at *Assembly6* performing a permission demand on *UIPermission*, and goes on its way as it usually goes. Now the stack walk reaches *Assembly4* and recognizes an assert on the permission it is checking. The stack walk stops there and returns with a positive result. Because the stack walk was short-circuited, the CLR has no way of knowing that *Assembly1* did not have this permission.

An *Assert* can be set using both the declarative and the imperative syntax. In the first example, the declarative syntax is used. An *Assert* is set on the *FileIOPermission.Write* permission for the CFG files in the C:\Test directory:

```
Public Function _
    <FileIOPermission(SecurityAction.Assert, Write := "C:\Test\*.cfg")> _
      Act1() As Integer
       ' body of the function
End Function
```

Figure 12.3 A Stack Walk Is Short-Circuited by an *Assert*



The second example uses the imperative syntax setting the same type of *Assert*:

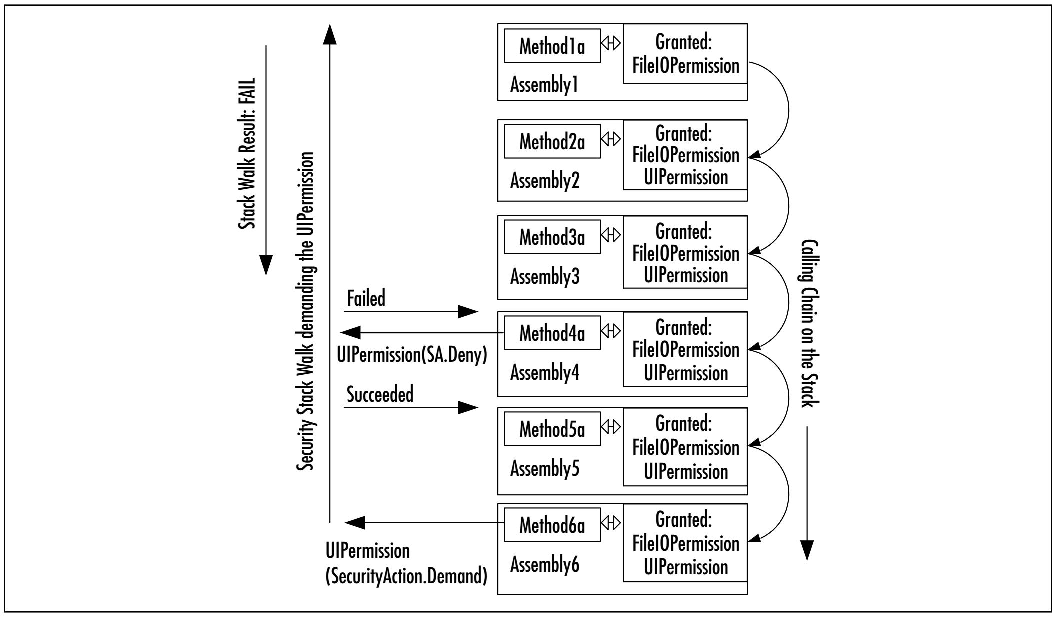
```
Public Function Act1() As Integer
    Dim ActFilePerm As New
FileIOPermission(FileIOPermissionAccess.Write, "C:\Test\*.cfg")
          ActFilePerm.Assert
           ' rest of body
End Function
```

Deny Override

The *Deny* does the opposite of *Assert* in that it lets a stack walk fail for the permission the *Deny* is set on. There are not many situations where a *Deny* override makes sense, but here is one: Among the permissions your code has is *RegistryPermission*. Now it has to make a call to a method for which you have no information regarding trust. To prevent that code from taking advantage of the *RegistryPermission*, your code can set a *Deny*. Now you are sure that your code does not hand over a high-trust permission.

Because unnecessary *Deny* overrides can disrupt the normal working of security checks (because they will always fail on a *Deny*), you should revert the *Deny* after the call ends for which you set the *Deny*.

Figure 12.4 A Stack Walk Is Short-Circuited by a *Deny*


Stack Walk Result: FAIL
Security Stack Walk demanding the UIPermission
Method1a
Assembly1
Granted: FileIOPermission
Method2a
Assembly2
Granted: FileIOPermission UIPermission
Method3a
Assembly3
Granted: FileIOPermission UIPermission
Failed
UIPermission(SA.Deny)
Method4a
Assembly4
Granted: FileIOPermission UIPermission
Succeeded
Method5a
Assembly5
Granted: FileIOPermission UIPermission
Method6a
Assembly6
Granted: FileIOPermission UIPermission
UIPermission (SecurityAction.Demand)
Calling Chain on the Stack

For the sake of the example, we use the same situation as in Figure 12.3, but instead of an *Assert*, there is a *Deny* (see Figure 12.4). Again, the security stack walk is triggered for the *UIPermission* permission in *Assembly6*. When the stack walk reaches *Assembly4*, it recognizes the *Deny* on *UIPermission* and it ends with a fail. In our example, the security check would ultimately have failed in *Assembly1*, but if *Assembly1* had been granted the *UIPermission*, the stack walk would have succeeded, if not for the *Deny*. Effectively this means that *Assembly4* revoked the *UIPermission* for *Assembly5* and *Assembly6*.

You can set a *Deny* by using both the declarative and the imperative syntax. In the first example, the declarative syntax is used. A *Deny* is set on the *FileIOPermission* permission for all the files in the C:\Winnt\System32 directory:

```
Public Function _
      <FileIOPermission(SecurityAction.Deny, All :=
"C:\Winnt\System32\*.*")> _
      Act1() As Integer
            ' body of the function
End Function
```

The second example uses the imperative syntax setting the same type of *Assert*:

```
Public Function Act1() As Integer
     Dim ActFilePerm As New
FileIOPermission(FileIOPermissionAccess.AllAccess, _
    "C:\Winnt\System32\*.*")
           ActFilePerm.Deny
           ' rest of the body
End Function
```

PermitOnly Override

The *PermitOnly* override is more like the negation of the *Deny*, by *Deny*ing every permission but the one specified. You use the *PermitOnly* for the same reason you use *Deny*, only this one is more rigorous. For example, if you permit only the *UIPermission* permission, every security stack walk will fail but the one that checks on the *UIPermission*. Take Figure 12.4 and substitute *Deny* with *PermitOnly*. If in *Assembly6* the security check for *UIPermission* is triggered, the stack walk will pass *Assembly4* with success, but will ultimately fail in *Assembly1*. If any other security check is initiated, they will fail in *Assembly*. The end result is that *Assembly5* and *Assembly6* are denied any access to a protected resource that incorporate a *Demand* request, because every security check will fail. As you can see, *PermitOnly* is a very effective way of killing any aspirations of called code in accessing protected resources. The *PermitOnly* is used in the same way as *Deny* and *Assert*.

Custom Permissions

The .NET Framework enables you to write your own code access permissions, even though the framework comes with a large number of code access permission classes. Because these classes are meant to protect the protected resources and code that are exposed by the framework, it may well be the case that the application you are developing has defined resources that are not protected by the framework permissions, or you want to use permissions that are more tuned toward the needs of your application.

You are completely free to replace existing framework permission classes, although this requires a large amount of expertise and experience. In case you are just adding new permission classes to the existing ones, you should be particularly careful not to overlap permissions. If more than one permission protects the same resource or operation, an administrator has to take this into account if he has to modify the rights to these resources.

NOTE

The subject of overlapping permissions brings up a topic not discussed earlier. Although the whole discussion of code access permission has been from the standpoint of the CLR, or .NET Framework, eventually the CLR has to access resources on behalf of the users/application. Even if the code has been granted a specific permission to access a protected resource, that does not automatically mean that it is allowed to access that system resource. Take the example of a method having the *FileIOPermission* permission to the directory C:\Winnt\System32. If the identity of the Windows principal has not been given access to this part of the file system, accessing a file in that directory will fail anyway. This implies that the administrator not only has to set up the permissions within the security policy, but he also has to configure the Windows 2000 platform to reflect these access permissions.

Building your own permissions does not only imply that certain development issues are raised, but even more so the integrity of the whole security system must be discussed. You have to take into account that you are adding to a rigid security system that relies heavily on trust and permissions. If mistakes occur in the design and/or implementation of a permission, you run the risk of creating security holes that can become the target of attacks or grant an application access to protected resources even if it is not authorized to access these. Discussing the process of designing your own permissions goes beyond the scope of this chapter. However, the following steps give you an understanding of what is involved in creating a custom permission:

1. Design a permission class.
2. Implement the interfaces *IPermission* and *IUnrestrictedPermission*.

3. In case special data types have to be supported, you must implement the interface *ISerializable*.
4. You must implement XML encoding and decoding.
5. You must implement the support for declarative security.
6. Add Demand calls for the custom permission in your code.
7. Update the security policy so that the custom permission can be added to permission sets.

Role-Based Security

Role-based security is not new to the .NET Framework. If you already have experience with developing COM+ components, you surely have come across role-based security. The concept of role-based security for COM+ applications is the same as for the .NET Framework. The difference lies in the way it is implemented. If we talk about role-based security, the same example comes up, over and over again. This is not because we can't create our own example, but because it explains role-based security in a way everybody understands. So here it is. You build a financial application that can handle deposit transactions. The rule in most banks is that the teller is authorized to make transactions up to a certain amount, let say $5,000. If the transaction goes beyond that amount, the teller's manager has to step in to perform the transaction. However, because the manager is only authorized to do transaction up to $10,000, the branch manager has to be called to process a deposit transaction that is over this amount.

So, as you can see, role-based security has to do with limiting the tasks a user can perform, based on the role(s) he plays or the identity he has. Within the .NET Framework, this all comes down to the principal that holds the identity and role(s) of the caller. As discussed earlier in this chapter, every thread is provided with a principal object. In order to have the .NET Framework handle the role-based security in the same manner as it does code access security, the permission class *PrincipalPermission* is defined. To avoid any kind of confusion, *PrincipalPermission* is *not* a derived class of *CodeAccessPermission*. In fact, *PrincipalPermission* holds only three attributes: User, Role, and the Boolean *IsAuthenticated*.

Principals

Let's get back to where it all starts: the principal. From the moment an application domain is initialized, a default call context is created to which the principal will be bound. If a new thread it activated, the call context and the principal are copied from the parent thread to the new thread. Together with the principal object, the identity object is also copied. If the CLR cannot determine what the principal of a thread is, a default principal and identity object is created so that the thread can run at least with a security context with minimum rights. There are three type of principals: *WindowsPrincipal*, *GenericPrincipal*, and *CustomPrincipal*. The latter goes beyond the scope of this chapter and is not discussed any further.

WindowsPrincipal

Because the *WindowsPrincipal* that references the *WindowsIdentity* is directly related to a Windows user, this type of identity can be regarded as very strong because an independent source authenticated this user.

To be able to perform role-based validations, you have to create a *WindowsPrincipal* object. In the case of the *WindowsPrincipal*, this is reasonably straightforward, and there are actually two ways of implementing it. This depends on whether you have to perform just a single validation of the user and role(s), or you have to do this repeatedly. Let's start with the single validation solution:

1. Initialize an instance of the *WindowsIdentity* object using this code:

   ```
   Dim WinIdent as WindowsIdentity = WindowsIdentity.GetCurrent()
   ```

2. Create an instance of the *WindowsPrincipal* object and bind the *WindowsIdentity* to it:

   ```
   Dim WinPrinc as New WindowsPrincipal(WindIdent)
   ```

3. Now you can access the attributes of the *WindowsIdentity* and *WindowsPrincipal* object:

   ```
   Dim PrincName As String = WinPrinc.Identity.Name
   Dim IdentName As String = WinIdent.Name 'this is the same as
       the previous line
   Dim IdentType As String = WinIdent.AuthenticationType
   ```

If you have to perform role-based validation repeatedly, binding the *WindowsPrincipal* to the thread is more efficient, so that the information is readily available. In the previous example, you did not bind the *WindowsPrincipal* to the thread because it was intended to be used only once. However, it is good practice to always bind the *WindowsPrincipal* to the thread because in case a new thread is created, the principal is also copied to the new thread:

1. Create a principal policy based on the *WindowsPrincipal* and bind it to the current thread. This initializes an instance of the *WindowsIdentity* object, creates an instance of the *WindowsPrincipal* object, binds the *WindowsIdentity* to it, and then binds the *WindowsPrincipal* to the current thread. This is all done in a single statement:

   ```
   AppDomain.CurrentDomain.SetPrincipalPolicy(PrincipalPolicy.
       WindowsPrincipal)
   ```

2. Get a copy of the *WindowsPrincipal* object that is bound to the thread:

   ```
   Dim WinPrinc As WindowsPrincipal =
       Ctype(Thread.CurrentPrincipal, WindowsPrincipal)
   ```

It is possible to bind the *WindowsPrincipal* in the first method of creation to the thread. However, your code must be granted the *SecurityPermission* permission to do so. If that is the case, you bind the principal to the thread by the following:

```
Thread.CurrentPrincipal = WinPrinc
```

GenericPrincipal

In a situation where you do not want to rely on the Windows authentication but want the application to take care of it, you can use the *GenericPrincipal*.

NOTE

Always use an authentication method before letting a user access your application. Authentication, in any shape or form, is the only way to establish an identity. Without it you are not able to implement role-base security.

Let's assume that your application requested a username and password from the user, checked it against the application's own authentication database, and established the user's identity. You then have to create the *GenericPrincipal* to be able to perform role-based verifications in your application:

1. Create a *GenericIdentity* object for the *User1* you just authenticated:

   ```
   Dim GenIdent As New GenericIdentity("User1")
   ```

2. Create the *GenericPrincipal* object, bind the *GenericIdentity* object to it, and add roles to the *GenericPrincipal*:

   ```
   Dim UserRoles as String() = {"Role1", "Role2", "Role5"}
   Dim GenPrinc As New GenericPrincipal(GenIdent, UserRoles)
   ```

3. Bind the *GenericPrincipal* to the thread. Again, you need *SecurityPermission*:

   ```
   Thread.CurrentPrincipal = GenPrinc
   ```

Manipulating Identity

You can manipulate the identity that is held by a principal object in two ways. The first is replacing the principal; the second is by impersonating.

Replacing the principal object on the thread is a typical action you perform in applications that have their own authentication methods. To be able to replace a principal, your code must have been granted the *SecurityPermission*, or more specifically, the *SecurityPermission* attribute *ControlPrincipal*. This will allow your own code to be able to pass on the *PrincipalObject* to other code. This attribute grants you the permission to manipulate the principal, so you are allowed by the CLR to pass on the principal. Replacing the principal object can be done by performing these steps:

1. Create a new identity and principal object and initialize it with the proper values.
2. Bind the new principal to the thread:

   ```
   Thread.CurrentPrincipal = NewPrincipalObject
   ```

Impersonating is also a way of manipulating the principal, with the intent to take on the identity of another user to perform some actions on their behalf. You can identify two variations:

- The code has to impersonate the *WindowsPrincipal* that is attached to the thread. This may seem a little odd, but you have to remember that your code is part of an application domain that runs in a process. A user—whether a system account, a service account, or even an interactive user—starts this process on the Windows platform. Although the principal can be used to perform role-based verification within the code, accessing protected resources is still done with the identity of the process user, unless you actively use the user account of principal through impersonation.

- The code has to impersonate a user that is not attached to the current thread. The first thing you have to do is obtain the Windows token of the user you want to impersonate. This has to be done with the unmanaged code *LogonUser*. The obtained token has to be passed to a new *WindowIdentity* object. Now you have to call the *Impersonate* method of *WindowsIdentity*. The old identity, hence token, has to be saved in a new instance of *WindowsImpersonationContext*.

At the end of the impersonation, you have to change back to the original user account by calling the Undo method of the *WindowsImpersonationContext*.

Remember the principal object is not changed, rather the *WindowsIdentity* token, representing the Windows account, is switched with the current token. At the end of the impersonation, the tokens are switched back again, as shown in the following steps:

1. Call the *LogonUser* method, located in the unmanaged code library advapi32.dll. You pass the username, domain, password, logon type, and logon provider to this method that will return you a handle to a token. For the sake of the example, we will call it hImpToken.
2. Create a new *WindowsIdentity* object and pass it the token handle:

   ```
   Dim ImpersIdent As New WindowsIdentity(hImpToken)
   ```

3. Create a *WindowsImpersonationContext* object and call the *Impersonate* method of *ImpersIndent*:

   ```
   Dim WinImpersCtxt As WindowsImpersonationContext =
   ImpersIdent.Impersonate()
   ```

4. At the end of the call, the original Windows token has to be put back in the Identity object:

   ```
   WinImpersCtxt.Undo()
   ```

You could have done Steps 2 and 3 in one statement that looks like this:

```
Dim WinImpersCtct As WindowsImpersonationContext = _
          WindowsIdentity.Impersonate(hImptoken)
```

Remember that you cannot impersonate when you use a *GenericPrincipal* because it does not reference a Windows identity. For generic principals, you will need to replace the principal with one that has a new identity.

Role-Based Security Checks

Having discussed the creation and manipulation of *PrincipalObject*, it is time to take a look at how they can assist you in performing role-based security checks. Here is where *PrincipalPermission*, already mentioned in the beginning of the section "Role-Base Security," comes into play. Using *PrincipalPermission*, you can make checks on the active principal object, be it the *WindowsPrincipal* or the *GenericPrincipal*. The active principal object can be one you created to perform a one-time check, or it can be the principal you bound to the thread. Like the code access permissions, the *PrincipalPermission* can be used in both the declarative and the imperative way.

To use *PrincipalPermission* in a declarative manner, you need to use the *PrincipalPermissionAttribute* object in the following way:

```
Public Shared Function
    <PrincipalPermissiobAttribute(SecurityAction.Demand, _
                   Name := "User1", Role := "Role1")> Act2()
                       As Integer
          ' body of the function
End Function
<assembly: PrincipalPermissionAttribute(SecurityAction.Demand, Role :=
    'Administrator')>
```

To use the imperative manner, you can perform the *PrincipalPermission* check as shown:

```
Dim PrincPerm As New PrincipalPermission("User1", "Role1")
PrincPerm.Demand()
```

It is also possible to use the imperative to set the *PrincipalPermission* object in two other ways:

```
Dim PrincState As PermissionState = Unrestricted
Dim PrincPerm As New PrincipalPermission(PrincState)
```

The permission state (*PrincState*) can be None or Unrestricted, where None means the principal is not authenticated. So, the user name is Nothing, the role is Nothing, and Authenticated is false. *Unrestricted* matches all other principals.

```
Dim PrincAuthenticated As Boolean = True
Dim PrincPerm As New PrincipalPermission("User1", "Role1",
    PrincAuthenticated)
```

The *IsAuthenticated* field (*Princauthenticated*) can be true or false. In a situation where you want *PrincipalPermission.Demand()* to allow more than one user/role combination, you can perform a union of two *PrincipalPermission* objects. However, this is only possible if the objects are of the same type. Thus, if one *PrincipalPermission* object has set a user/role, and the other object uses *PermissionState*, the CLR throws an exception. The union looks like this:

```
Dim PrincPerm1 As New PrincipalPermission("User1", "Role1")
Dim PrincPerm2 As New PrincipalPermission("User2", "Role2")
PrincPerm1.Union(PrincPerm2).Demand()
```

The *Demand* will succeed only if the principal object has the user *User1* in the role *Role1* or *User2* in the role *Role2*. Any other combination fails.

As mentioned before, you can also directly access the principal and identity object, thereby enabling you to perform your own security checks without the use of *PrincipalPermission*. Besides the fact that you can examine a little more information, it also prevents you from handling exceptions that can occur using *PrincipalPermission*. .You can query the WindowsPrincipal in the same way the PrincipalPermission does this:

- The name of the user by checking the value of *WindowsPrincipal.Identity.Name*:

```
If (WinPrinc.Identity.Name = "User1") or _
    WinPrinc.Identity.Name.Equals("DOMAIN1\User1") Then
End If
```

- An available role by calling the *IsInRole* method:

```
If (WinPrinc.IsInRole("Role1")) Then
End If
```

- Determining if the principal is authenticated, by checking the value of *WindowsPrincipal.Identity.IsAuthenticated*:

```
If (WinPrinc.Identity.IsAuthenticated) Then
End If
```

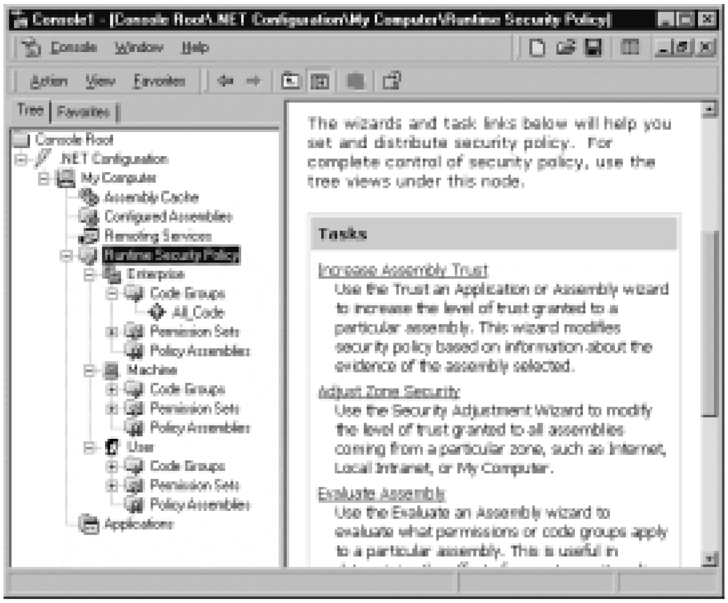
Additionally for *PrincipalPermission*, you can check the following *WindowsIdentity* properties:

- **AuthenticationType** Determines the type of authentication that is used. Most common values are NTLM and Kerberos.
- **IsAnonymous** Determines if the user is identified as an anonymous account by the system.
- **IsGuest** Determines if the user is identified as a guest account by the system.
- **IsSystem** Determines if the user is identified as the system account of the system.
- **Token** Returns the Windows account token of the user.

Security Policies

This section takes a closer look at the way security policies are constructed and the way you can manage them. To create and modify a security policy, the .NET Framework provides you two tools: a command-line interface (CLI) tool, called **caspol.exe** (see the section "Security Tools") and a Microsoft Management Console snap-in, "mcscorcfg.msc" (see Figure 12.5). The latter will be used for demonstration purposes because it is more visual and intuitive.

Figure 12.5 The .NET Configuration Snap-In



The wizards and task links below will help you set and distribute security policy. For complete control of security policy, use the tree views under this node.

Tasks

Increase Assembly Trust
Use the Trust an Application or Assembly wizard to increase the level of trust granted to a particular assembly. This wizard modifies security policy based on information about the evidence of the assembly selected.

Adjust Zone Security
Use the Security Adjustment Wizard to modify the level of trust granted to all assemblies coming from a particular zone, such as Internet, Local Intranet, or My Computer.

Evaluate Assembly
Use the Evaluate an Assembly wizard to evaluate what permissions or code groups apply to a particular assembly. This is useful in

As you can see in Figure 12.5, the security policy model is made up of the following:

- Runtime Security Policy levels:
  - **Enterprise** Valid for all managed code that is used within the whole organization (enterprise); therefore this will have "by nature" a restrictive policy because it references a large group of code.
  - **Machine** Valid for all managed code on that specific computer. Because this already limits the amount of code, you can be more specific with handing out permissions.
  - **User** Valid for all the managed code that runs under that Windows user. This will normally be the account that starts the process in which the CLR and managed code runs. Because the identity of the user is very specific, the granted permissions can also be more specific, thus less restrictive.
- A code groups hierarchy that exists for each of the three policy levels. We will look at how you can add code groups to the default structure, which already exists for user and machine.
- (Named) Permission Sets. By default the .NET Framework comes with seven named permission sets:
  - **FullTrust** Unlimited access to all protected resources and operations.
  - **EveryThing** Granted all .NET Framework permissions, except the security permission *SkipVerification.*
  - **LocalIntranet** The default rights given to an application on the local intranet.
  - **Internet** The default rights given to an application on the Internet.
  - **Execution** Has only the security permission EnableAssemblyExecution.
  - **SkipVerification** Has only the security permission SkipVerification.
  - **Nothing** Denied all access to all protected resources and operations.
- Evidence, which is the attribute that the code hands over to the CLR and on which it determines the effective permission set. Evidence is used in the construction of code groups.

- Policy assemblies that list the trusted assemblies that hold security objects used during policy evaluation. You should add your assemblies to the list that implements the custom permissions. If you omit this, the assemblies will not be fully trusted and cannot be used during the evaluation of the security policy.

Understand that the evaluation process of the security policy will result in the effective permission set for a specific assembly. For all of the three policy levels, the code groups are evaluated against the evidence presented by the assembly. All the code groups that meet the evidence deliver a permission set. The union of these sets determines the effective permission set for that particular security policy level. After this evaluation is done at all three security levels, the three individual permission sets are intersected, resulting in the effective permission set for an assembly. This means that the code groups within the three security levels cannot be constructed independently, because this may result in a situation where an assembly is given a limited permission set that is too limited to run. When you take a look at the permission set for the *All_Code* of the enterprise security policy, you will see that it is Full Trust. Doing the same for the *All_Code* of the user security policy, you will see Nothing. Because the code group tree of the enterprise is empty, it cannot make evidence decisions; therefore it cannot contribute to the determination of the effective permission set of the assembly. By setting it to Full Trust, it is up to the machine and user security policy to determine the effective permission set.

Because the user code group already has a limited code group tree, the root does not need to participate in the determination of the permission set. By setting it to Nothing, it is up to the rest of the code groups to decide what the effective permission group for the user security policy is. You can determine the permission set of a code group by performing these steps:

1. Run Microsoft Management Console (MMC) by choosing **Start | Run** and typing **mmc**.
2. Open the .NET Management snap-in, via **Console | Add/Remove Snap-in**.
3. Expand the **Console Root | .NET Configuration | My Computer**.
4. Expand **Runtime Security Policy | Enterprise |Code Groups**.
5. Select the code group **All_Code**.
6. Right-click **All_Code** and select **Properties**.

7. Select the **Permission Set** tab.
8. The **Permission Set** field lists the current value.

Creating a New Permission Set

Suppose you decide that none of the seven built-in permissions sets satisfy your need for granting permissions. Therefore, you want to make a named permission set that does suit you. You have a few options:

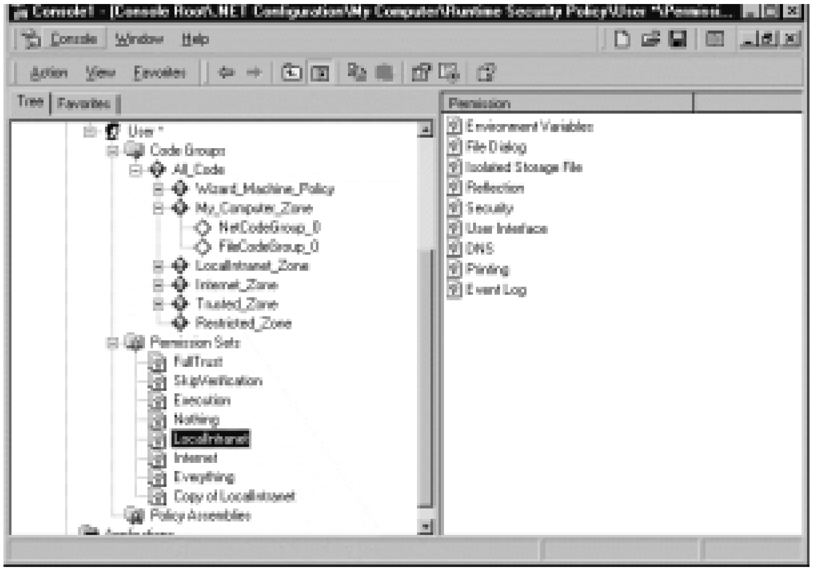
- Create a permission from scratch.
- Create a new permission set based on a existing one.
- Create a new permission from an XML-coded permission set.

To get a better understanding of the working of the security policy and to get some hands-on experience with the tool, we discuss the different security policy issues in the following exercises.

We use the second option and base our new permission set on the permission set *LocalIntranet* for the user security policy level:

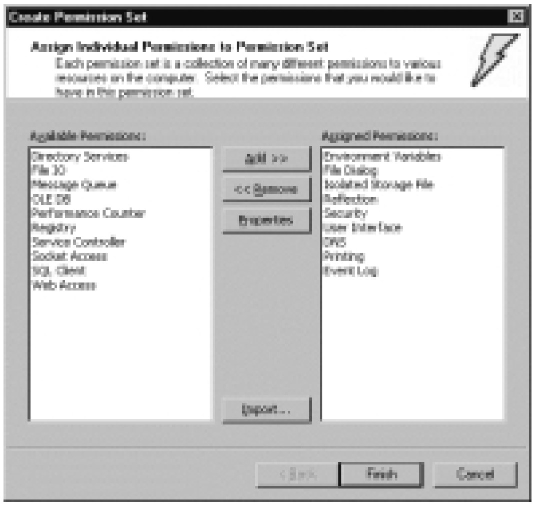
1. Expand the **User** runtime security policy and expand **Permission Sets** (see Figure 12.6).

Figure 12.6 The Users Permission Sets and Code Groups



2. Right-click the permission set **LocalIntranet** and select **Duplicate**; a permission set called **Copy of LocalIntranet** is added to the list.
3. Select the permission set **Copy of LocalIntranet** and rename it to **PrivatePermissions**. Then, right-click it and select **Properties**. Change the **Permission Set Name** to **PrivatePermissions** and, while you're at it, change the corresponding **Permission Set Description**.
4. Change the permissions of the permission set: Right-click the **PrivatePermissions** permission set and select **Change Permissions**.
5. The **Create Permission Set** dialog box appears (see Figure 12.7). You see two permissions lists: on the left, the Available Permissions that are not assigned, and on the right, the list with assigned permissions.

Figure 12.7 Modify the Permission Set Using the Create Permission Set Dialog Box


Create Permission Set

Assign Individual Permissions to Permission Set
Each permission set is a collection of many different permissions to various resources on the computer. Select the permissions that you would like to have in this permission set.

Available Permissions:
Directory Services
File IO
Message Queue
OLE DB
Performance Counter
Registry
Service Controller
Socket Access
SQL Client
Web Access

Assigned Permissions:
Environment Variables
File Dialog
Isolated Storage File
Reflection
Security
User Interface
DNS
Printing
Event Log

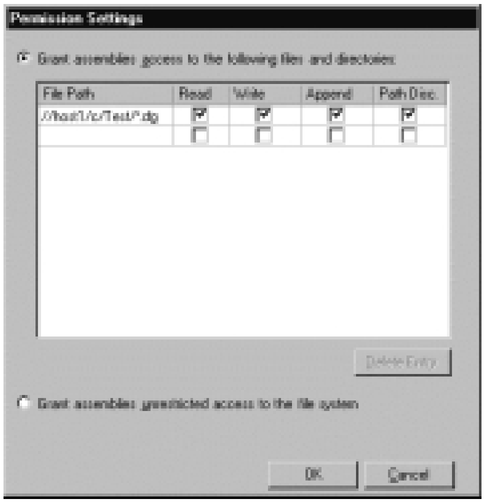
Between the two Permissions lists are four buttons. The **Add** and **Remove** buttons let you move individual permissions between the lists. Note that you cannot select more than one at the same time. This is done to prevent you from making mistakes. You will better understand a given permission if you select that permission in the Assigned Permissions list and press the **Properties** button. You can use the fourth button (**Import**) to load an XML-coded permission set. Now, let's make some modifications to the permission set, because that was the reason to duplicate the permission set:

- Add the *FileIOPermission* to the Assigned Permission list.
- Add the *RegistryPermission* to the Assigned Permission list.
- Modify the *SecurityPermission* properties.

To do so:

1. Select **FileIO** in the Available Permissions list. (Notice that if you have selected a permission in the Assigned Permissions list, this permission stays selected.)
2. Click **Add**. A **Permission Settings** dialog box for the FileIO appears (see Figure 12.8). (You can also double-click the permission to add it to the Assigned Permissions list. But do not double-click an assigned permission by accident—this will remove the permission from the assigned permission list.) On the Permission Settings dialog box, you are given the option to select between **Grant assemblies access to the following files and directories** and **Grant assemblies unrestricted access to the file system**.

Figure 12.8 Modify the Settings of FileIO Using the Permission Settings Dialog Box



Permission Settings

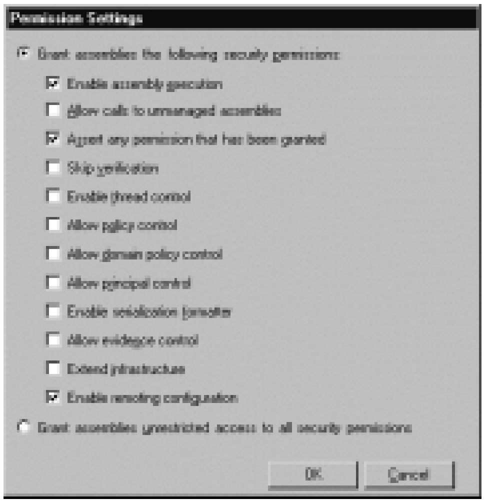
[✓ Grant assemblies access to the following files and directories]

File Path	Read	Write	Append	Path Disc.
//host1/c/Test/*.dg	✓	✓	✓	✓
	☐	☐	☐	☐

3. Choose the first one, and because it is already selected, we can focus our attention on the empty list window below the option. You may expect an Add button below the list, especially because there is a **Delete Entry** one. However, there is an auto-add list. You fill in a line, and it is automatically added. Add a second line, and a third empty line will appear.

4. As you saw earlier this chapter, this resembles the way we used *FileIOPermission* and *FileIOPermissionAttribute* to demand and request access to specific files in a specific directory. Go ahead, fill in "C:\Test\*.cfg". Surprised you get an error message? The point is that the field demands that you use UNC names. The advantage is that you can reference to files on other servers in the domain. However, the dialog box checks the existence of the path when you click **OK**, so be sure that the UNC path exists.
5. Fill the File Path with a valid UNC of the machine you are working on, and because we want to give full access, you can check all four boxes. (Note that if you do not check any of the boxes, then this is accepted, because you filled in a File Path. However if you check the properties of FileIO as an assigned permission, you will notice that the line has disappeared—hence a beta bug!)
6. Click **OK** and you have added a permission to the assigned permission list. You are now ready for the next permission.
7. Double-click the **Registry** permission and a **Permissions Setting** dialog box appears that looks a lot alike the one you just saw with **FileIO**. Keep the option **Grant assemblies access to the following registry keys**.
8. Fill the **Key** field with a valid HKEY value, such as HKEY_LOCAL _MACHINE, and check the **Read** box, so that we can give read permission to the specified Registry tree.
9. Click **OK**, and you have added your second permission to your permission set.
10. The last task is to modify the Security permission. So, select the **Security** permission in the Assigned Permissions list (do not double-click, because that will remove the permission from the list) and click **Properties**.
11. A Permission Settings dialog box (see Figure 12.9) appears. You see that the option **Grant assemblies the following security permissions** is selected, together with the properties **Enable assembly execution**, **Assert any permission that has been granted**, and **Enable remoting configuration**.

Figure 12.9 Modify the Settings of Security Using the Permission Settings Dialog Box



Permission Settings

[✓] Grant assemblies the following security permissions:

- [✓] Enable assembly execution
- [ ] Allow calls to unmanaged assemblies
- [✓] Assert any permission that has been granted
- [ ] Skip verification
- [ ] Enable thread control
- [ ] Allow policy control
- [ ] Allow domain policy control
- [ ] Allow principal control
- [ ] Enable serialization formatter
- [ ] Allow evidence control
- [ ] Extend infrastructure
- [✓] Enable remoting configuration

[ ] Grant assemblies unrestricted access to all security permissions

OK Cancel

12. We also want to grant our security policy the security permission properties. Check **Allow calls to unmanaged assemblies** because we want to make calls to unmanaged code. Also check **Allow principal control** because we want to be able to modify principal settings. Click **OK**, and you are done, for now, with modifying your first permission set.
13. Click **Finish**. You will probably get a warning message stating that you changed your security policy and you have to save it. Up until the point you save the policy, an asterisk (*) will mark the user policy.
14. You can save the policy by right-clicking the **User** runtime security policy and selecting **Save**.

If you want this permission set to also become part of the machine and/or enterprise permission sets, you can simply copy and paste it.

You will also notice two other options: **Reset** and **Restore Policy**. The first one resets the policy back to the default setting of the policy. You can try it, but it will wipe out all the changes you made up until now. The latter makes it possible to go back to the previous save. This is possible because for each of the runtime security policies, the settings are saved in an XML-coded file that becomes the current one. Before this happens, it renames the old one with the extension .old. The current one has the extension .cch. The default policy has no extension, so to speak. For the user security policy, you have the following files:

- **security.config** The default security; used by the **Reset** option.
- **security.config.cch** The current/active policy.
- **security.config.old** The last saved policy version; used by the **Restore Policy** option.

The enterprise security uses the name enterprisesec.config and the machine uses the name security.config. This is possible because the user security policy is saved in the user's directory tree in the following folder:

```
Document and Settings\User_Name\Application Data\Microsoft\CLR Security
    config\v1.0.xxxx
```

The enterprise and machine security policies are saved in the following directory:

```
WINNT\Microsoft.NET\Framework\v1.0.xxxx\CONFIG
```

This directory is located by the CLR through the HiveKey:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Catalog42\NetFrameworkv1\
    MachineConfigdirectory
```

Because the configuration files are XML-coded, you can open them with a Web browser and examine them. This will give you additional understanding how the permission sets are set up. This also means that you can modify the default security policies.

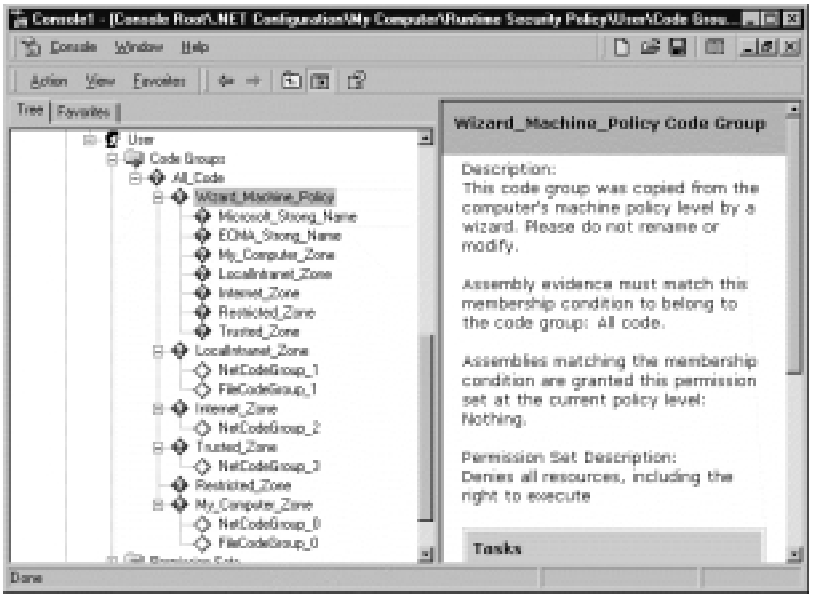
Modifying the Code Group Structure

Now that we have created a security permission set, it makes sense to start using it. We can do so by attaching it to a code group. We are going to modify the code groups structure of the user security policy. By default, the user already has a basic structure (see Figure 12.10). A few things may strike you at first sight:

- There is a code group called *Wizard_Machine_Policy*. The description of this group tells you that a wizard, called the Adjust Security Wizard, copied this group from the computer's policy level and that you should not modify it. This description is not totally true. In fact, if you take a closer look at these code groups, you will see that all groups that end with *_Zone* have a permission set of Nothing. This means that you, the user, cannot make use of the permission sets of the machine that are based on the zone evidence. However, if you are given more permissions based on the zone evidence, this will be toned down by the zone-based permission of the machine policy. The user can have permissions based on zoned evidence that is equal to or less than allowed by the machine. However, you do see zone-based code groups at the same level as the *Wizard_Machine_Policy*, because these are the code groups that are copied from the machine policy.

- The zone-based code groups contain *NetCodeGroup* and *FileCodeGroup*. As the description states, they are generated by the .NET Configuration Tool, hence the tool we are working with at the moment. The custom code groups are based on XML-code files and can therefore not be edited by the tool. However, you can use the **caspol.exe** tool to do so. Without going into detail regarding what exactly these groups entail, it suffices to state that they are necessary for you to use the .NET Configuration Tool. If you remove or modify them, you may lock yourself out from using this tool.

Figure 12.10 The Default Code Group Structure for the User Security Policy



Wizard_Machine_Policy Code Group

Description:
This code group was copied from the computer's machine policy level by a wizard. Please do not rename or modify.

Assembly evidence must match this membership condition to belong to the code group: All code.

Assemblies matching the membership condition are granted this permission set at the current policy level: Nothing.

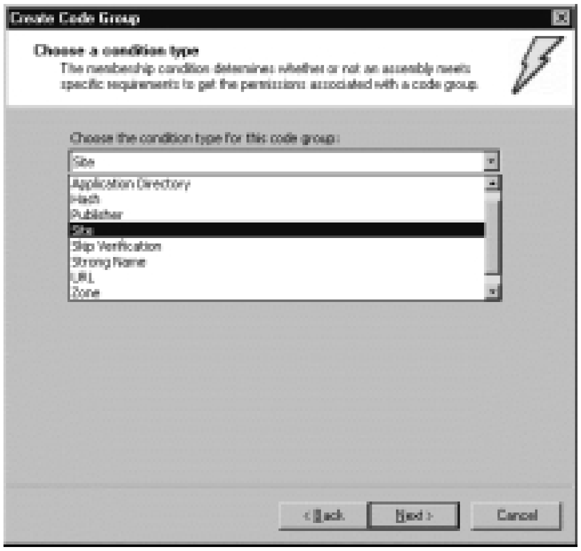
Permission Set Description:
Denies all resources, including the right to execute

Tasks

Let's create a small code groups structure that is made up of two code groups directly under the *All_Code* group and apply our own custom-made permission set *PrivatePermissions* to the *LocalIntranet_Zone* group:

1. If you do not have the MMC with the .NET Management snap-in open, open it now.
2. Expand the tree to **.NET Configuration | My Computer | Runtime Security Policy | User**.
3. Now expand **Code Groups | All_Code**.
4. Right-click **All_Code** and select **New**; the Create Code Group dialog box appears.
5. You are given two options: **Create a new code Group** and **Import a code group from a XML File**. Use the first option. (Note: For the *NetCodeGroup* and *FileCodeGroup*, the latter is used).
6. You have to enter at least the **Name** field. For this example, we choose *PrivateGroup_1*. Now click **Next**.
7. The dialog box shows you a second page called **Choose a condition Type** and has just one field called **Choose the condition type for this code group**. The field has a pull-down menu containing the values you can choose from. All of these, except the first and last one—All Code and (custom)—are evidence-related (see Figure 12.11).

Figure 12.11 Select a Condition Type for a Code Group



Create Code Group

Choose a condition type

The membership condition determines whether or not an assembly meets specific requirements to get the permissions associated with a code group.

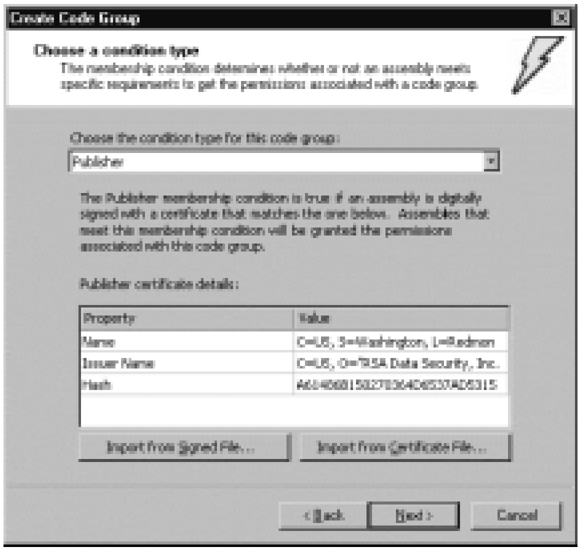
Choose the condition type for this code group:

Site

Application Directory
Hash
Publisher
Site
Skip Verification
Strong Name
URL
Zone

8. Select **Site** from the drop-down menu. A new field, called **Site Name** appears and is related to the **Site** condition. For the sake of the example, we choose the MSDN Subscribers download site, so we enter the value **msdn.one.microsoft.com** in the site field.
9. Click **Next** and the third page, called **Assign a Permission Set to the Code Group**, appears.
10. You can choose between the options **Use existing permission set** and **Create a new permission set**. Because the site comes from the Internet, that permission set will do.
11. Select the value **Nothing** from the drop-down menu. (Note: The permission set we just made is also part of the list) and click **Next**.
12. Click **Finish**, and you have created your first code group. While we are at it, let's create the second code group, which will be the child of the code group we just created.
13. Right-click the code group *PrivateGroup_1* and select **New**.
14. Create a new code group named **PrivateGroup_2** and click **Next**.
15. Select the value **Publisher** from the drop-down menu. Below the field, a new box called **Publisher Certificate Details** appears and has to be filled by importing a certificate. You can do this by reading out of a signed assembly using the **Import from Signed File** button (Note: it should say Import from signed Assembly). Or, you can import a certificate file, using the **Import from Certificate File** button.
16. For the purpose of this example, we use the Certificate from the msdn.one.microsoft.com site. (Note: In case you have forgotten how this is done, you go to a protected site, thus using SSL. You double-click the icon indicating that the site is protected. This opens up the certificate. Go to the **Details** tab and click the **Copy to File** button.) See CD file Chapter 12/MSDN-One.cer.
17. Click the **Import from Certificate File** button, browse to the certificate file (the extension is .cer) and open it. You will see that the field in the certificate box will be filled (see Figure 12.12).
18. Click **Next**.
19. Select the existing permission group ***LocalIntranet***. We can give more permissions now we know that the signed assemblies indeed comes from Microsoft MSDN, but also originates from the corresponding Web site.

Figure 12.12 Importing a Certificate for a Publisher Condition in a Code Group



Choose a condition type

The membership condition determines whether or not an assembly meets specific requirements to get the permissions associated with a code group.

Choose the condition type for this code group: Publisher

The Publisher membership condition is true if an assembly is digitally signed with a certificate that matches the one below. Assemblies that meet this membership condition will be granted the permissions associated with this code group.

Publisher certificate details:

Property	Value
Name	C=US, S=Washington, L=Redmon
Issuer Name	C=US, O="RSA Data Security, Inc.
Hash	A6040681502703640E537AD6315

20. Click **Next** and **Finish**.

Before tackling our last task, let's recap what we have done. We were concerned with creating a permission set for signed assemblies that come from the msdn.one .microsoft.com site. So what if the assembly comes from this Web site but is not signed? It meets the condition of *PrivateGroup_1*, so it will get the permission set of this code group. Because this is Nothing, this would mean that these assemblies are granted no permission. But because the msdn.one.microsoft.com site comes from the Internet Zone, it also meets the condition of the code group *Internet_Zone*, which grants any assembly from this zone the Internet permission set. And because a union is taken from all the granted permission sets, these assemblies will still have enough permissions to run.

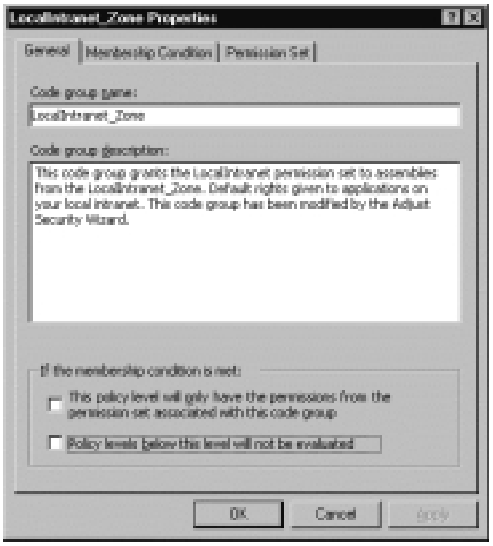
Why not make the *PrivateGroup_2* a child of *Internet_Zone*, because unsigned assemblies from msdn.one.microsoft.com are granted the Internet permission set any way? The reason is simple: We only want to give signed assemblies from msdn.one.Microsoft.com additional permission if they also originate from the appropriate Web site. In case such a signed assembly originates from another Web site, we treat it as any other assembly coming from an Internet Zone. The reason for giving *PrivateGroup_1* the Nothing permission set is that it is only there to force assemblies to meet both conditions, and *PrivateGroup_1* is just an intermediate stage to meet all conditions.

What you have to keep in mind is that we only discussed how the actual permission set is determined at the user security policy level. This will be intersected with the actual permission set determined on the machine level. And because at the machine level the assembly will be given only the Internet permission set, our signed assembly will wind up with the effective permission set of Internet. Normally, the actual permission set of the enterprise is also taken into the intersection, but because that code group tree has only the *All_Code* code group with full trust, it will play no role in the intersection of this example.

Our last task is replacing a permission set:

1. Right-click the code group *LocalIntranet_Zone* and select **Properties**. The **LocalIntranet_Zone Properties** dialog box appears (see Figure 12.13).

Figure 12.13 Setting Attributes in the General Tab of the Code Group Permission Dialog Box



Code group name:
LocalIntranet_Zone

Code group description:
This code group grants the LocalIntranet permission set to assemblies from the LocalIntranet_Zone. Default rights given to applications on your local intranet. This code group has been modified by the Adjust Security Wizard.

If the membership condition is met:
- [ ] This policy level will only have the permissions from the permission set associated with this code group
- [ ] Policy levels below this level will not be evaluated

2. Select the **Permission Set** tab.
3. Open the pop-up menu with available permission sets and select ***PrivatePermissions***. You will see that the list box will reflect the permissions that make up the *PrivatePermissions* permission set.
4. Click **Apply** and go back to the **General** tab.

On this tab, there is a frame called **If the membership condition is met**, which shows two options:

- **This policy level will have only the permissions from the permission set associated with this code group.** This refers to the code group attribute *Exclusive*.
- **Policy levels below this level will not be evaluated.** This refers to the code group attribute *LevelFinal*.

Both need some explanation, so let's go back to our msdn.one.microsoft.com example. Suppose you open the properties dialog box of the *Internet_Zone* code group and check the **Exclusive** option (of course, you have to save it first for it to become active). We received a signed assembly from msdn.one.microsoft.com that also originates from this site. We had established that it would be granted the *LocalIntranet_Zone* permission at the user policy level. But now the **Exclusive** option comes into play. Because our signed assembly also meets the *Internet_Zone* condition, the Internet permission set is valid. The exclusive that is set for the *Internet_Zone* code group forces all other valid permission sets to be ignored by not taking a union of these permission sets. Instead, the permission set with the exclusive attribute becomes the actual permission set for the user policy level. Because it will be intersected with the actual permission sets of the other security levels, it also determines the maximum set of permissions that will be granted to the signed assembly. Use this attribute with care, because from all the code groups an assembly is a member, hence meets the condition, only one can have the exclusive attribute. The CLR determines if this is the case. When the CLR determines that an assembly meets the condition of more than one code group with the Exclusive attribute, it will throw an exception, and it fails to determine the effective permission set and the assembly is not allowed to execute.

The way the *LevelFinal* is handled is more straightforward. Understand that by establishing the effective permission set of an assembly, the CLR evaluates the security policies starting at the highest level (enterprise, followed by user and machine). Again take our MSDN example. We set a *LevelFinal* in the *PrivateGroup_2* code group and removed the Exclusive attribute from *Internet_Zone*. When the effective permission set for a signed assembly from msdn.one.microsoft.com that originates from that Web site has to be established, the CLR starts with determining the actual permission set of the enterprise policy level. This is for *All_Code* Full Trust, effectively taking this policy level out of the intersection of actual permission sets. Now the user policy level gets its turn in establishing the actual permission set. As you know by now, this will be equal to the *LocalIntranet_Zone* permission set. But the CLR has also encountered the *LevelFinal* attribute. It refrains from establishing the actual permission set of the machine policy level and intersects the actual permission sets from the enterprise and user policy level. The actual permission set will be equal to *LocalIntranet_Zone*.

Because the machine policy level is not considered the actual permission set in this case has more permission than in the situation where the *LevelFinal* attribute has not been set.

Remoting Security

Discussing security between systems always provides a new set of security issues. This is no exception for remoting. Let's start with the communication between systems. If you use an *HttpChannel*, you can make use of the SSL encryption. The *FtpChannel* does not have encryption, but if both servers support IPSec, you are able to create a secured channel, through which the *FtpChannel* can communicate.

The next issue is to what extent you trust the other system. Even with a secure channel in place, how do you know that the other system has not been compromised? You need at least a sturdy authentication mechanism in place and need to avoid the use of anonymous users, although this will not always be possible. At least try to use NTLM or Kerberos for authentication. The latter is a perfect vehicle for handling impersonation between multiple systems. If you need to use anonymous users, you can use IIS as the store-front and let the IIS handle the impersonation. You can also use a proxy to prevent a user from directly accessing your IIS.

The messages that are exchanged should always be signed so you are able to verify the sender and/or origin. Even when you are sure that a message is transported over a secured channel, you are never sure if the message that is put in this channel, has been sent out of ill-intent.

This chapter has discussed the use of code access and role-base security. The more thoroughly you use this runtime security instrument, the better you can control the remoting security.

Cryptography

There is no subject about security that does not reference cryptography. Although it is an absolute necessity to create a secure environment, it is not the "Holy Grail" of security. This section highlights the cryptography features that come with the .NET Framework. If you already have worked with Windows 2000 Cryptographic Service Providers (CSPs) and/or used the CryptoAPI, you know nearly everything there is to know about cryptography in the .NET Framework.

The most important observation is that the ease-of-use of crypto functionalities have improved a lot over the way we had to use the CryptoAPI, which only was available for C/C++. An important addition in the design concept of the cryptography namespace is the use of *CryptoStreams*, which make it possible to chain any cryptographic object that makes use of CryptoStreams together. This means that the output from one cryptographic object can be directly forwarded as the input of another cryptographic object without the need of storing the output result in an intermediate object. This can enhance the performance significantly if large pieces of data have to be encoded or hashed. Another addition is the functionality to sign XML code, although only for use within the .NET Framework security system. To what extend these methods comply with the proposed standard RFC 3075 is unclear. Within the .NET Framework, three namespaces involve cryptography:

- *System.Security.Cryptography* The most important one; resembles the CryptoAPI functionalities.
- *System.Security.Cryptography* .X509 certificates Relates only to the X509 v3 certificate used with Authenticode.
- *System.Security.Cryptography.Xml* For exclusive use within the .NET Framework security system.

The cryptography namespaces support the following CSP classes that will be matched on the Windows 2000 CSPs, by the CLR. If a CSP is available within the .NET Framework, this does not automatically implies that the corresponding Windows 2000 CSP is available on the system the CLR is running:

- *DESCryptoServiceProvider* Provides the functionalities of the symmetric key algorithm Data Encryption Standard.
- *DSACryptoServiceProvider* Provides the functionalities of the asymmetric key algorithm Data Signature Algorithm.
- *MD5CryptoServiceProvider* Provides the functionalities of the hash algorithm Message Digest 5.
- *RC2CryptoServiceProvider* Provides the functionalities for the symmetric key algorithm RC 2 (name after the inventor: Rivest's Cipher 2).
- *RNGCryptoServiceProvider* Provides the functionalities for a Random Number Generator.

- *RSACryptoServiceProvider* Provides the functionalities for the asymmetric algorithm RSA (named after the inventors Rivest, Shamir, and Adleman).
- *SHA1CryptoServiceProvider* Provides the functionalities for the hash algorithm Secure Hash Algorithm 1.
- *TripleDESCryptoServiceProvider* Provides the functionalities for the symmetric key algorithm 3DES.

To be complete, a short description of symmetric key algorithm, asymmetric key algorithm, and hash algorithm are given. A *symmetric key algorithm* enables you to encrypt/decrypt data that is sent between you and another party. The same key is used to both encrypt and decrypt the data. That is why it is called a symmetric algorithm. This algorithm forces you to exchange the key with your counter party, but this must be done in a way that no other party can intercept this key. Because symmetric key algorithms are often used for a short exchange of data, it is also referred to as session key algorithm. For the exchange of session keys, the parties involve use an asymmetric key algorithm.

An *asymmetric key algorithm* makes use of a *key pair.* One is private and is kept under lock and key by the owner and the other is public and available for everyone. Because the algorithm uses two related but different keys to encrypt and decrypt, it is called an asymmetric algorithm, but is also referenced as a *public key algorithm*. The public key is wrapped in a certificate that is a "proof of authenticity," and that certificate has to be issued by an organization that is trusted by all involved parties. This organization is called a certificate authority, of which Verisign is the best known. So what about using an asymmetric key algorithm to exchange symmetric keys? The best example is two Windows 2000 servers that need to regularly set up connection between both servers on behalf of their users. Each connection, hence session, has to be secured and needs to use a session key that is unique in relation to the other secured sessions. The servers exchange a session key for every connection. Both have an asymmetric key-pair and have exchanged the public key in a certificate. So if one server wants to send a session key to the other server, it uses the public key of the other server to encrypt the session key before it sends it. The server knows that only the other server can decrypt the session key because that server has the private key that is needed to decrypt the session key.

A *hash algorithm,* also referred to as a one-way hash algorithm, can take a variable piece of data and transform it to a fixed-length piece of data, called a *hash* or *message digest* that is nearly always much shorter, for example 160 bits for SHA-1.

One-way means that you cannot derive the source data by examining only the digest. Another important feature of the hash algorithm is that it generates a hash that is unique for each piece of data, even if just one bit of data is changed. You can see a hash value as the fingerprint of a piece of data. Let's say, for example, you send somebody a plain text e-mail. How do you and the receiver of the e-mail know that the message has not been altered while it was sent? Here is where the message digest comes in. Before you send your e-mail, you apply a hash algorithm on that message, and you send the message and message digest to the receiver. The receiver can perform the same hash on the message, and if both the digest and the message are the same, the message has not been altered. Yes, somebody who alters your message can also generate a new digest and obscure his act. Well, that is where the next trick comes in. When you send the digest, you encrypt it with your own private key, of which you know the receiver has the public part. Because this not only prevents the message from being changed without you and the receiver discovering it, but it also confirms to the receiver that the message came from you and only you. How?

Well, let's assume that somebody intercepts your message and wants to change it. He has your public key, so he can decrypt your message digest. But, because he doesn't have your private key, he is unable to encrypt a newly generated digest. So he cannot go forward with his plan to change the e-mail without anybody finding out. Eventually the e-mail arrives at the receiver's Inbox. He takes the encrypted digest and decrypts it using your public key. If that succeeds, he knows first of all that this message digest must have been sent by you because you are the only one who has access to the private key. He calculates the hash on the message and compares both digests. If they match, he not only knows that the message hasn't been tampered with, but also that the message came from only you because every message has a unique hash. And because he already established that the encrypted hash came from you, the message must also come from you.

Security Tools

The .NET Framework comes with ten command-line security tools (see Table 12.4) that help you to perform your security tasks. For a more thorough description of these tools, you should consult the .NET Framework documentation.

Table 12.4 Command-Line Security Tools

Name of Tool	Name of Executable	Description
Code Access Security Policy Utility	Caspol.exe	This tool can perform any operation in relation to the code access security policy. Because it can do more than the .NET Configuration Tool we have been using in this chapter, it is important that you familiarize yourself with it.
Certificate Verification Utility	Chktrust.exe	With this tool, you can check a file that has been signed using Authenticode.
Certificate Creation Utility	Makecert.exe	Creates a X.509 certificate for testing purposes. A option you may consider is to install the Certificates Services on Windows 2000, which makes it a lot easier to create and maintain certificates for development and testing purposes.
Certificate Manager Utility	Certmgr.exe	This utility manages your certificates, certificate trust lists, and so on. Use the Microsoft Management Console with the certificates snap-in, which enables you to maintain not only your own certificates, but also (if you have the rights) the certificates of your computer and service accounts.
Software Publisher Certificate Test Utility	Cert2spc.exe	This tool create a software publishers certificate for one or more X.509 certificates.
Permissions View Utility	Permview.exe	This tool enables you to view the requested permissions of an assembly.
PE Verify Utility	Peverify.exe	This tool enables you to verify the type safety of a portable executable file.
Secutil Utility	Secutil.exe	This tool extracts strong name or public key information from an assembly and converts it so that you can use it directly in your code (for example, for a permission demand).

Continued

Table 12.4 Continued

Name of Tool	Name of Executable	Description
File Signing Utility	Signcode.exe	This tool enables you to sign a PE file with an Authenticode signature. If this utility is called with no command-line options, a Digital Signature Wizard is started.
Strong Name Utility	Sn.exe	This tool enables you to sign assemblies with strong names.
Set Registry Utility	Setreg.exe	This tools enables you to set Registry keys for use of public key cryptography. If you call this utility without options, it will just list the settings.
Isolated Storage Utility	Storeadm.exe	This tool enables you to manage isolated storage for the current user.

Summary

Positioning the .NET Framework as a distributed application environment, Microsoft was well aware that they had to pay attention to how an application can be secured, due to the great risks that distributed security incorporate. That is why they introduced a rights- and permission-driven security mechanism, that is flexible as well as rigid. Flexible because you can own your designed and customized permissions and rigid because it is always there, even if the application takes no notice of permissions. To add to that, the CLR will check the code on type safety (it checks whether the code is trying to stick its nose in places it does not belong) during the JIT compilation.

The .NET Common Language Runtime (CLR) will always perform a security check—called code access security—on an assembly if it wants to access a protected resource or operation. To prevent an assembly from obscuring its restricted permissions by calling another assembly, the CLR will perform a security stack walk. It checks every assembly in a calling chain of assemblies to see if every single one has this permission. If this is not the case, the assembly is not given access to this protected resource or operation.

What permissions an assembly is granted and what permission an assembly requests is controlled in two ways. The first one is controlled by code groups that grant permissions to an assembly based on the evidence it presents to the CLR. The assembly itself controls the latter. Secure conscious assemblies request only the permissions it needs, even if the CLR is willing to grant it more permissions. By doing this, the assembly insures itself from being misused by other code that wants to make use of its permission set. A code group hierarchy has to be set up by an administrator, which he can do at different security policy levels: enterprise, user, and machine.

To establish the effective set of permissions, the CLR uses a straightforward and robust method: It determines all valid permission sets based on the evidence an assembly presents per security policy level, and the actual permission set per policy level is the union of the valid permission set. The CLR does this for all the policy levels and intersects the actual permission set to determine the effective permission set of an assembly.

Added to the code access security, the CLR still supports role-based security, although its implementation is slightly different than you were accustomed to with COM. Every executing thread has a security context called principal that reference the identity of the user. The principal is also used for impersonation of the executing user. The principal comes in a few forms: based on Windows user accounts and the authentication mechanisms that come with it; not based on Windows account, called "Generic" that can be controlled by custom made authentication services and a "Base" form that enables you to custom make your own principal and identity. The code can reference the principal to check if the user has a specific role.

Still, the most important security feature is security policies, which not only allow you to create code groups but to also build your own permission set that can be enriched with custom permissions. The custom permissions can be added to the .NET Framework without opening up the security system, provided that you make no security mistakes in the coding of the permissions.

As can be expected from every framework that relies on security, the .NET Framework comes with a complete set of cryptography functionalities, equal to what we had with the CryptoAPI, only the ease-of-use has improved a lot and is no longer dependent on C/C++. To control cryptographic functionalities, such as certificates and code signing, the .NET Framework has a set of security utilities that enables you to control and maintain the security of your applications during its development and deployment process.

Solutions Fast Track

Security Concepts

☑ Permissions are used to control the access to protected resources and operations.

☑ Principal is the security context that is attached to every executing thread in the CLR. It also holds the identity of the user, such as Windows account information, and the roles that user has. It also contributes to the ability of the code to impersonate.

☑ Authentication and authorization can be controlled by the application itself or rely on external authentication methods, such as NTLM and Kerberos. Once Windows has authorized a user to execute CLR-based code, the code has to control all other authorization that is based on the identity of the user and information that comes with assemblies, called evidence.

☑ Security policy is what controls the whole CLR security system. A system administrator can build policies that grant assemblies permissions access to protected resources and operations. This permission granting is based on evidence that the assemblies hands over to the CLR. If the rules that make up the security policy are well constructed, it enables the CLR to provide a secure runtime environment.

☑ Type safety is related to the prevention of assembly code to reach into memory/storage of other applications. Type safety is always checked during JIT compilation and therefore before the code is even loaded into the runtime environment. Only code that is granted the Skip Verification permission can bypass type safety checking, unless this is turned off altogether.

Code Access Security

☑ Code access security is based on granting assemblies permission and enforcing that it can never gain more permissions. This enforcing is done by what is known as security stack walking. When a call is made to a protected resource or operation, the assembly the CLR demanded from the assembly that has a specific permission. But instead of checking only the assembly that made the call, the CLR checks every assembly that is part of a calling chain. If all these assemblies have that specific permission, the access to the protected resource/operation is allowed.

☑ To be able to write secure code, it is possible to refrain from permissions that are granted to the code. This is done by requesting the necessary permissions for the assembly to run, whereby the CLR gives the assembly only these permissions, under the reservation that the requested permissions are part of the permission set the CLR was willing to grant the assembly anyway. By making your assemblies request a limited permission set, you can prevent other code from misusing the extended permission set of your code. However, you can also make optional requests, which allows the code to be executed even if the requested permission is not part of the granted permission set. Only when the code is confronted with a demand of having such a permission, it must be able to handle the exception that is thrown, if it does not have this permission.

☑ The demanding of a caller to have a specific permission can be done using declarative and imperative syntax. Requesting permissions can only be done in a declarative way. Declarative means that it is not part of the actual code but is attached to an assembly, class, or method using a special syntax enclosed with **<>**. When the code is compiled to the intermediate language (IL) or a portable executable (PE), these demands/request are extracted from the code and placed in the metadata of the assembly. This metadata is read and interpreted by the CLR before the assembly is loaded. The imperative way makes the demands part of the code. This can be sensible if the demands are conditional. Because a demand can always fail and result in an exception being thrown by the CLR, the code has to be equipped in handling these exceptions.

- ☑ The code can control the way the security stack walk is performed. By using *Assert*, *Deny*, or *PermitOnly*, which can be set with both the declarative and imperative syntax, the stack walk is finished before it reaches the end of the stack. When CLR comes across an *Assert* during a stack walk, it finishes with a Succeed. If it encounters a *Deny*, it is finished with a Fail. With the *PermitOnly*, it succeeds only if the checked permission is the same or a subset of the permission defined with the *PermitOnly*. Every other demand will fail at the *PermitOnly*.
- ☑ Custom permissions can be constructed and added to the runtime system.

Role-Based Security

- ☑ Every executing thread in the .NET runtime system has a identity that is part if the security context, called principal.
- ☑ Based on the principal, role-based checks can be performed.
- ☑ Role-based checks can be performed in a declarative, imperative, and direct way. The direct way is by accessing the principal and/or identity object and querying the values of the fields.

Security Policies

- ☑ A security policy is defined on different levels: enterprise, user, machine, and application domain. The latter is not always used.
- ☑ A security policy has permission sets attached that are built-in—such as FullTrust or Internet—or custom made. A permission set is a collection of permissions. By grouping permissions, you can easily address them, only using the name of the permission set.

- ☑ The important part of the policy are the security rules, called code groups; these groups are constructed in an hierarchy.
- ☑ A code group checks the assembly based on the evidence it presents. If the assembly's evidence meets the condition, the assembly is regarded as a member of this code group and is successively granted the permissions of the permission set related to the code group. After all code groups are checked, the permission sets of all the code groups the assembly is a member of are united to an actual permission set for the assembly at that security level.
- ☑ The CLR performs this code group checking on every security level, resulting in three or four actual permission sets. These are intersected to result in the effective permission set of permissions granted to the assembly.
- ☑ Remoting limits the extent to which the security policy can be applied. To create a secure environment, you need to secure remoting in such a way that access to your secured CLR environment can be fully controlled.

Cryptography

- ☑ The .NET Framework comes with a cryptography namespace that covers all necessary cryptography functionalities that are at least equal to the CryptoAPI that was used up until now.
- ☑ Using the cryptography classes is much easier than using the CryptoAPI.

Security Tools

- ☑ The .NET Framework comes with a set of security tools that enable you to maintain certificates, sign code, create and maintain security policies, and control the security of assemblies.
- ☑ Two comparable tools enable you to maintain code access security. **Caspol.exe** (Code Access Security Policy Utility) has to be operated from the command-line interface. The .NET Configuration Tool comes as a snap-in for the Microsoft Management Console (MMC) and is therefore more intuitive and easier to use than **caspol.exe**.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: I want to prevent an overload of security stack walk, how can I control this?

A: This can indeed become a major concern if it turns out that the code accesses a significant number of protected resources and/or operations, especially if they happen in a long calling-chain. The only way to prevent this from happening is to put in a *SecurityAction.Assert* just before a protected resource/operation is called. This implies that you need a thorough understanding of when a stack walk, hence demand, is triggered and on what permission this stack walk will be performed. By just placing an *Assert*, you create an uncontrolled security hole. What you can do is the following, which can be applied in the situation in which you make a call to a protected resource but do this from within a loop-structure. You can also use it in a situation in which you call a method that makes a number of calls to (different) protected resources/operations that trigger the demand for the same type of permission.

The only way to prevent a number of stack walks is to place an imperative assertion on the permission that will be demanded. Now you know that the stack walk will be stopped in its tracks. To close the security hole you just opened, you place an imperative demand for the permission you asserted in front of the assertion. If the demand succeeds, you know that in the other part of the calling-chain everything is OK in regard to this permission. And because nothing will change if you check a second or third time, you can save yourself from a lot of unnecessary stack walks. Think about a 1,000-fold loop: You just cleared your code from doing redundant 999 stack walks.

Q: When should I use the imperative syntax and when should I use the declarative?

A: First, make sure that you understand the difference in the effect they take. The imperative syntax makes a demand, or override for that matter, on part of your code. It is executed when the line of code that holds the demand/

override is encountered during runtime. The declarative syntax brings these demands and overrides right into the metadata of the assembly. During the load phase of the assembly, the metadata is extracted and interpreted, meaning that the CLR already takes action on this information. If a stack walk takes place, the CLR can handle overrides much quicker than if they would occur during execution, thus the imperative way. However, demands should only be made at the point they are really necessary. Most of the time demands are conditional—think about whether the demand is based on a role-based security check. If you would make a demand declarative for a class or method, it will be trigger a stack walk every time this class or method is referenced, even if demands turns out to be not needed. So to recap: Make overrides declarative and place them in the header of the method, unless all methods in the class need the assertion; then, you place it in the class declaration. Remember that an assembly cannot have more than one active override type. If you cannot avoid this, you need to use declarative overrides anyway. Make demands imperative and place them just before you have to access a protected resource/operation.

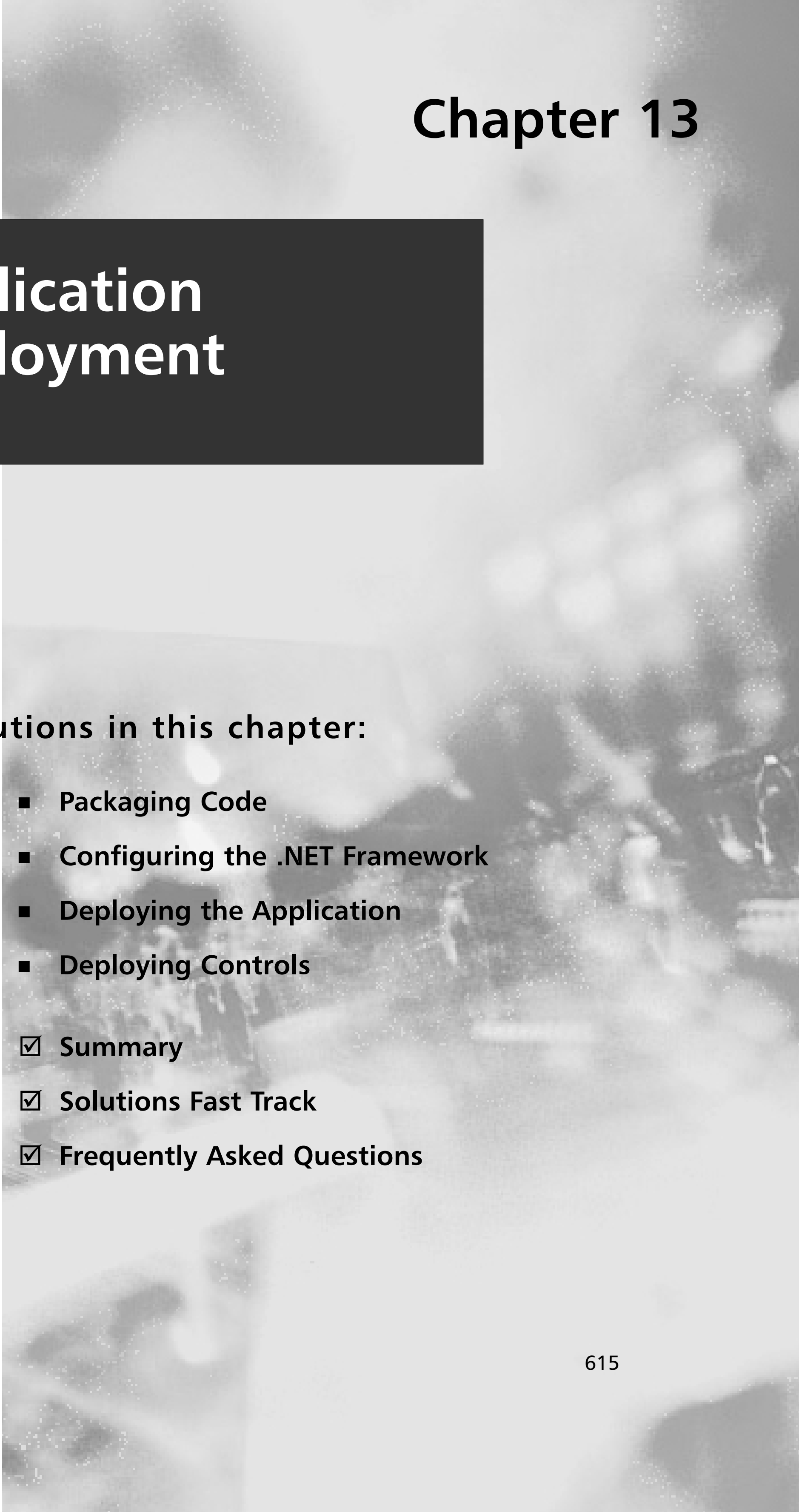
Q: How should I go about building a code group hierarchy?

A: You need to remember four important issues in building a code group hierarchy:

- An assembly can not be a member of code groups that have conflicting permissions; for example, one with unrestricted *FileIOPermission* and one with a more restricted *FileIOPermission*.
- The bigger the code group hierarchy, the harder it is to maintain it.
- The larger the number of permission sets; the harder it is to maintain them.
- The harder it is to maintain code groups and permissions sets, the more likely it is they contain security holes.

Anyhow the best approach is the largest common denominator. Security demands simplicity with as few exceptions as possible. Before you start creating custom properties sets, convince yourself that this is absolutely necessary. Nine out of ten times, one of the built-in permission sets suffices. The same goes for code groups—most assemblies will fit nicely in a code group based on their zone identity. If you conclude that this will not do, add only code groups that are more specific than the zone identity, like the publisher identity, but still apply to a large group of assemblies. Use more than one level in the code group hierarchy only if it is absolutely necessary to check on more than one membership condition, hence identity attribute. Add a permission set to the lowest level of the hierarchy only and apply the Nothing permission set to the parent code groups.

Take into account that the CLR will check on all policy levels, so check if you have to modify the code group hierarchy of only one policy level, or that this has to be done on more levels. Remember: The CLR will intersect the actual permission sets of all the policy levels.



Chapter 13

Application Deployment

Solutions in this chapter:

- Packaging Code
- Configuring the .NET Framework
- Deploying the Application
- Deploying Controls

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

The final stage in developing an application is preparing it for deployment. The first thing a user sees of your application is the installation. If problems arise during installation, the customer already has a negative perception of your application. Thankfully, deploying your application in Visual Basic .NET is simpler. How many times have you heard customers say they installed your application and now something else doesn't work? Windows applications can get complicated with many DLLs needed and so many versions available. The .NET Framework will allow different versions of a component on the same computer. You don't have to worry about registration problems anymore.

Packaging your application can be as simple as copying all of the files into a common directory. Your application will be comprised of one or more assemblies. Because assemblies are self-describing, you don't need to do much. You don't have to worry about all the correct Registry entries being set and whether or not a version of your component(s) already exists on the computer. If you want your application to set up the Start menu or maybe even create an icon in the Quick Launch toolbar, you may want to package your application to be installed by the Windows Installer.

The complexity of configuring your application varies. If you use private assemblies, all you have to do is copy all the files to the same directory. If you want to use public assemblies or use different directories for some assemblies, you will need to create a configuration file. This is just an XML file that contains configuration information. It allows for easy backup and can be created on an application, user, or machine basis. It also allows for easier administration, because you only have to worry about a file, you don't have to concern yourself with the Registry.

Deploying your application can be as simple as copying the files from a CD-ROM or across the network to a directory on the user's computer. Because the assemblies in your application are self-describing and contain all needed references internally, when the user runs the application, it will search for these references itself. No more runtime errors about components not being registered. However, this simple installation may not always be suitable for your needs. In this chapter, we cover how to install your Visual Basic .NET applications using the Windows Installer and creating Web downloads. When deploying controls, you need to take some additional factors into account. This chapter shows you how to get your applications ready to deploy.

Packaging Code

The first step in getting your VB.NET application deployed is getting it packaged (although for the .NET Framework it does not matter in what language the application is written). Depending on the complexity of your application, it will consist of one or more DLL and/or EXE files, also called *portable executables*. You have to package them into one or more *assemblies*. An assembly consists of at least two and at most four parts:

- **Assembly manifest** Mandatory because it contains the metadata that the CLR needs to execute the code.
- **Type metadata** Describes the types (class and methods) that are contained in the assembly.
- **Portable executables** The actual IL code.
- **Resources** Can be any type of nonexecutable file that needs to be used by code in the assembly.

Let's take a little closer look at the manifest because this is the "passport" of the assembly. By using the Intermediate Language Disassembler (ildasm.exe), you can see what is contained in an assembly. Figure 13.1 shows a part of the manifest of a sample that comes with the .NET Framework SDK. The part under *.assembly graphic* is interesting, not only because it states that the version (.ver) is 0.0.0.0, which is not allowed if you want to distribute your code, but it also lacks a public key, which is mandatory for sharing an assembly. What is actually missing is a strong name—a prerequisite in deploying. Let's set out to create an assembly that has a strong name so that we have a distributable package:

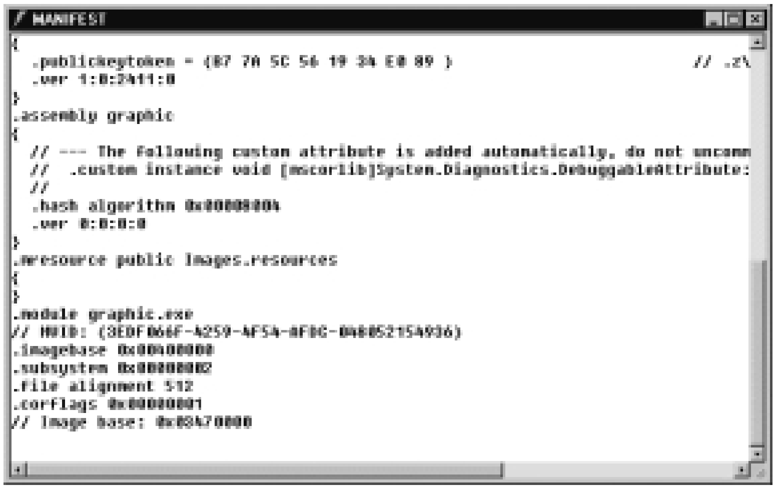
1. Use the strong name utility **sn.exe** to generate a key-pair in a file name GrphKey.snk:

   ```
   sn –k GrphKey.snk
   ```

2. Copy this file to a directory where it is easily accessible if you compile the program.
3. Add the necessary declarative statements to the code that take care of the generation of a strong name in the manifest of the assembly. They should be placed after the last **import** line and looks like this:

   ```
   <assembly: System.Reflection.AssemblyVersion("1.0.0.1")>
   <assembly: System.Reflection.AssemblyKeyFile("GrphKey.snk")>
   ```

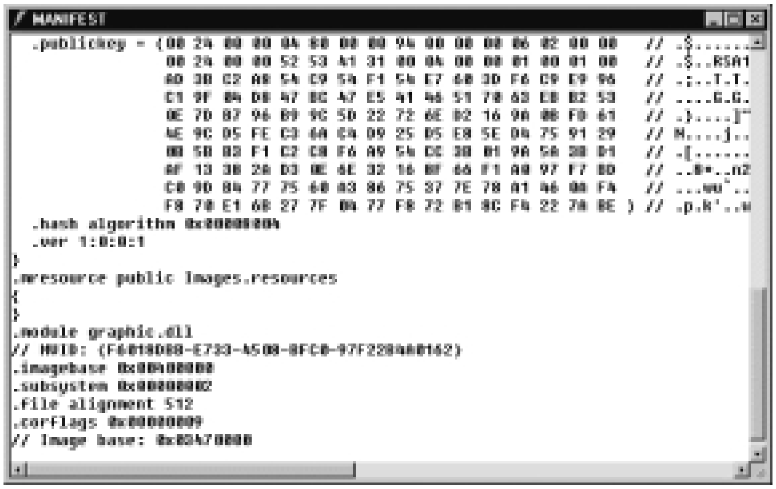
Figure 13.1 Part of the Manifest from the Private Graphic.exe



```
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         // .z\
  .ver 1:0:2411:0
}
.assembly graphic
{
  // --- The following custom attribute is added automatically, do not uncomm
  //  .custom instance void [mscorlib]System.Diagnostics.DebuggableAttribute:
  //
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.mresource public Images.resources
{
}
.module graphic.exe
// MVID: {3EDF066F-4259-4F54-8F0C-0A6052154936}
.imagebase 0x00400000
.subsystem 0x00000002
.file alignment 512
.corflags 0x00000001
// Image base: 0x03470000
```

4. Recompile the program and check the manifest, which will look something like Figure 13.2.

Figure 13.2 Part of the Manifest from the Public Shared Graphic.dll



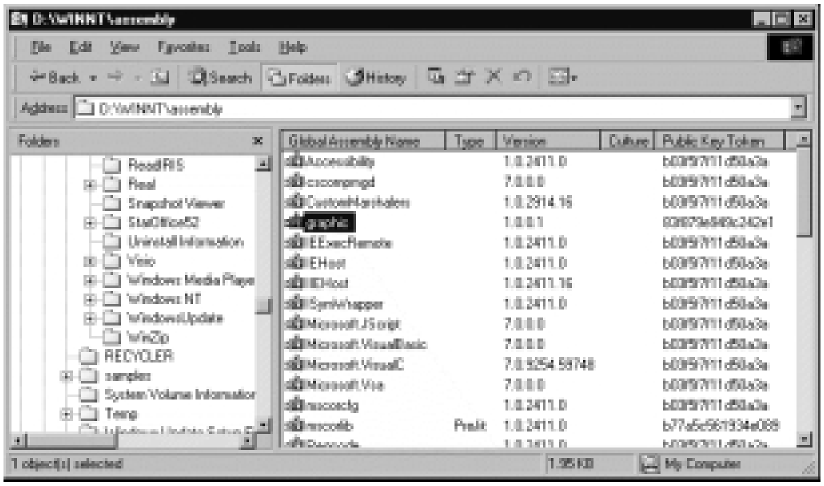
```
  .publickey = (00 24 00 00 04 80 00 00 94 00 00 00 06 02 00 00   // .$......
                00 24 00 00 52 53 41 31 00 04 00 00 01 00 01 00   // .$..RSA1
                AD 3D C2 A8 54 C9 54 F1 54 E7 60 3D F6 C9 E9 96   // .;..T.T.
                C1 9F 84 D8 47 BC 47 E5 41 46 51 70 63 EB B2 53   // ....G.G.
                0E 7D 87 96 B9 9C 5D 22 72 6E B2 16 9A 00 F0 61   // .}....]"
                4E 9C D5 FE C3 6A C4 D9 25 D5 E8 5E D4 75 91 29   // N....j..
                00 5B 83 F1 C2 C8 F6 A9 54 DC 3B 01 9A 5A 3B D1   // .[......
                AF 13 38 2A D3 0E 6E 32 16 8F 66 F1 A0 97 F7 BD   // ..8*..n2
                C0 9D 84 77 75 60 83 86 75 37 7E 78 A1 46 00 F4   // ...wu`..
                F8 70 E1 6B 27 7F 0A 77 F8 72 B1 8C F4 22 78 BE ) // .p.k'..w
  .hash algorithm 0x00008004
  .ver 1:0:0:1
}
.mresource public Images.resources
{
}
.module graphic.dll
// MVID: {F6018D88-E733-4508-8FC0-97F2284A0162}
.imagebase 0x00400000
.subsystem 0x00000002
.file alignment 512
.corflags 0x00000009
// Image base: 0x03470000
```

5. To be publicly available, it has to be placed in the general assembly cache, by using the General Assembly Cache utility tool (**gacutil.exe**). Issue the following command:

```
Gacutil.exe –/i Graphic.dll
```

6. **Gacutil.exe** returns with the message *Assembly successfully added to the cache*. Open Windows Explorer and go to the directory %WinDir%\ Assembly. There you will find graphic.dll, ready for you to use (see Figure 13.3).

Figure 13.3 Listing the Public Assemblies Available in the General Assembly Cache



Global Assembly Name	Type	Version	Culture	Public Key Token
Accessibility		1.0.2411.0		b03f5f7f11d50a3a
cscompmgd		7.0.0.0		b03f5f7f11d50a3a
CustomMarshalers		1.0.2914.16		b03f5f7f11d50a3a
graphic		1.0.0.1		93f079e840c242e1
IEExecRemote		1.0.2411.0		b03f5f7f11d50a3a
IEHost		1.0.2411.0		b03f5f7f11d50a3a
IIEHost		1.0.2411.16		b03f5f7f11d50a3a
ISymWrapper		1.0.2411.0		b03f5f7f11d50a3a
Microsoft.JScript		7.0.0.0		b03f5f7f11d50a3a
Microsoft.VisualBasic		7.0.0.0		b03f5f7f11d50a3a
Microsoft.VisualC		7.0.9254.59748		b03f5f7f11d50a3a
Microsoft.Vsa		7.0.0.0		b03f5f7f11d50a3a
mscorcfg		1.0.2411.0		b03f5f7f11d50a3a
mscorlib	Prejit	1.0.2411.0		b77a5c561934e089
[illegible]		1.0.2411.0		b03f5f7f11d50a3a

This does not mean that you cannot distribute an assembly that has no strong name; you can use it only for private use. If you try to add it in the general assembly cache, it will be rejected. After you have added a strong name to all the public shared assemblies and have set up all the private assemblies and other files that are needed for the application, you have to decide how you are going to package it to distribute. These are the two most commonly used methods:

- **Creating Cabinet files** You can do this with the utility **makecab.exe**. The advantage is that you compress the files, reducing the amount of data you have to distribute. Cabinet files are often used to download controls over the Internet using a Web browser.
- **Creating .msi files** You can use Visual Studio .NET to create MSI files for the deployment of your application.

Configuring & Implementing…

Assembly Versioning

Versioning of executables has always been important. How often did you see a dialog box that told you that you needed a DLL with version 1.2.3456 or higher? Whatever you did, you had just one version of that DLL, and a program ran with it or broke. This has changed with the .NET Framework because you can have as many different versions of an assembly on your system as are available. You can have different applications that use an assembly with the same name, but with a different version. The version number has become more of a "compatibility number" and also controls the way the CLR locates the appropriate assembly.

A version number must have the following structure:

```
Major.Minor[.Build[.Revision]]
```

This means that the **Major** and **Minor** are mandatory, and that **Build** and **Revision** are optional. However, if you want to use **Revision**, you must have a **Build**. The value of all the four parts can range from 0 to 65534 (included). This will be enough, especially with the speed in which Microsoft changes technologies.

As mentioned, the version number can also be regarded as a compatibility number, so a change in version number can reflect the following compatibility phases:

- **Compatible** If only the **Revision** number changes. Change of revision is seen as a quick fix engineering (QFE) update.
- **Possibly compatible** If the **Build** number has changed. There is no guarantee for backward compatibility.
- **Incompatible** If **Minor** and/or **Major** changes.

Because you no longer need to be concerned with backward compatibility—because any version can be kept available—you need to take care to use proper versioning. The versioning helps the CLR in finding a compatible assembly. Let's look at that process step-by-step:

1. The CLR reads the application configuration file, the machine configuration file and the publisher policy configuration file to determine what the correct version number is for the assembly that is referenced, and thus needs to be loaded.

Continued

The publisher policy configuration file is omitted if the application configuration file has put the version resolving in Safe Mode.

2. If the correct version has been established, the CLR checks if this assembly has already been requested in the application domain. If that is the case, the already loaded assembly is used. Note that this check is based on the assembly's full name: name, version, culture and public key token (strong name). You can get in trouble if you have two assemblies with the same name, although one has the .dll and the other the .exe extension. After the .exe version is loaded and another assembly makes a reference to the .dll version, the CLR will conclude that that assembly is already loaded because the CLR makes the distinction based on the full name, and that does not include a file extension.
3. In case the assembly is not loaded yet and is a strong name base assembly (meaning that it can be a shared assembly), the CLR checks the Global Assembly Cache (GAC).
4. If the assembly is not located in the GAC, the CLR goes on a search mission:
   - It checks the configuration file if a <codeBase> is provided. If so, this directory is checked for the presence of the assembly. In case the assembly is not located in the <codeBase> directory, the lookup fails.
   - If no <codeBase> is provided, the application base is checked.
   - If there is still no success, and the referenced assembly has a culture, the appropriate culture directories are checked. (By now the CLR is getting pretty desperate).
   - It checks the privatePath directories and is satisfied with less than a full name.

Two final remarks on this subject: If you reference an assembly, but it does not supply all the fields of the full name, called a *partial reference,* the CLR will quickly decide that it found the right assembly, even if it turns out not to be the case. In this case, your assembly binds with the wrong assembly (version). With partial references, the CLR goes for a "best effort approach."

Continued

Second, if an assembly has no strong name, (remember, this is only mandatory for the GAC), the CLR will not be checking on the correct version, even if you supply a version with the reference. Now the CLR finds itself thrown into a wild goose chase and again goes for the best effort approach.

A good rule of thumb is that you should always supply every assembly with a full name; it costs hardly any effort but makes it possible for the CLR to find the correct assembly and bind it.

Configuring the .NET Framework

An important issue for deploying an application is to make sure that the installation process on a computer goes smoothly and that the application executes as intended. You should also remember how and where the CLR finds the right assemblies, chooses which assembly version to use; and how it sets security. The list goes on and on. Before .NET, you mainly needed the Registry to accomplish this. Now, you create the same information in XML-coded configuration files. Perhaps you thought when you first read about the .NET Framework and how it would free you from the hassles of the Registry that you would no longer have to be concerned with configuration issues. If so, think again! There are three types of configuration files—machine, application, and security—so you can fine-tune the settings according to the needs of administrators and developers.

Creating Configuration Files

The configuration files, like nearly all other setting files within the .NET Framework, are XML-coded, adhering to a well-formed XML schema. In general, a configuration file consists of the following sections:

- **Startup** Holds settings that are related to the CLR to use.
- **Runtime** Holds settings that are related to the CLR working, especially how and where the CLR can find the proper assemblies.
- **Remoting** Holds settings related to the remoting system.
- **Crypto** Holds the settings related to the cryptography system.
- **Security** Holds the settings of the security policy.
- **Class API** Holds the settings related to the use of API.
- **Configuration** Holds the settings that are used by the application.

Technically speaking, you can find or put any of these sections in any configuration file. However, if a section does not apply to the use of the configuration file, it will be ignored.

NOTE

Configuration files, especially machine and security, have an impact on the workings of all applications that make use of the .NET Framework. Be very careful with making changes to these files before assessing the impact they will have. When you deploy a new application, you should not assume that certain modifications to general configuration files can be made to suit your application needs. These changes may influence the working of other applications.

A second warning about protecting your configuration files: Because they are so readable, making changes to them is easy. Persons with ill intent that have access to these files can do a lot of harm. Be sure that you limit the access to these files and make them at least read-only to also prevent accidental changes.

Machine/Administrator Configuration Files

Every machine that has the .NET runtime system installed has a machine configuration file named machine.config. You can find this file in the directory %CLR_InstallDir%\config. This file is especially important to reflect the correct assembly binding policy of that machine. The file also holds the settings for remoting channels. The settings in the machine configuration file take precedence over those in any other configuration file and cannot be overridden by any other file. These settings are "etched in stone," so to speak. The reason is obvious: The machine.config is expected to reflect the machine; any change to it may result in the breaking of the CLR. Nevertheless, you need to find the right balance between putting certain settings in the application configuration file or in the machine configuration file.

As an example, take a look at an excerpt from the machine.config file regarding remoting:

```
<system.runtime.remoting>
  <application>
   </application>
   <channels>
    <channel id="http"
      type="System.Runtime.Remoting.Channels.Http.HttpChannel,
          System.Runtime.Remoting" />
      <channel id="http server"
         type="System.Runtime.Remoting.Channels.Http
            .HttpServerChannel,
         System.Runtime.Remoting" />
      <channel id="tcp"
type="System.Runtime.Remoting.Channels.Tcp.TcpChannel,
         System.Runtime.Remoting" />
      <channel id="tcp server"
         type="System.Runtime.Remoting.Channels.Tcp.TcpServerChannel,
         System.Runtime.Remoting" />
     </channels>
     <channelSinkProviders>
      <serverProviders>
       <formatter id="soap"
          type="System.Runtime.Remoting.Channels
              .SoapServerFormatterSinkProvider,
          System.Runtime.Remoting" />
       <formatter id="binary"
          type="System.Runtime.Remoting.Channels
              .BinaryServerFormatterSinkProvider,
          System.Runtime.Remoting" />
       <provider id="wsdl"
          type="System.Runtime.Remoting.MetadataServices
              .SdlChannelSinkProvider,
          System.Runtime.Remoting" />
      </serverProviders>
     </channelSinkProviders>
    </system.runtime.remoting>
```

Application Configuration Files

The application configuration file is located in the installation directory of the application and is named after the application's program executable with *.config* added to the name, thus program.exe.config. The CLR checks the application directory for that file. Because an application does not need its own configuration file, it can completely depend on the machine configuration file, but nothing will happen if it is not there. Take notice of this! If you put it somewhere else, or use a different suffix, the CLR will not find it, which may mean that the CLR is not able to load the application. In the case of a browser-based application, the HTML page should use a link element to give the location of the configuration file, which resides in a directory on the Web server.

The application configuration file is especially useful for assembly binding settings that relate to specific assembly versions an application needs and the places the CLR has to look for the application's private assemblies, called *probing*. A possible configuration file for our earlier example of Graphic.dll may look like this:

```
<configuration>
    <runtime>
       <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
            <probing privatePath=".\SubDir1;.\SubDir2"/>
            <publisherPolicy apply="no"/>
           <dependentAssembly>
                <assemblyIdentity name="Graphic"
                                 publicKeyToken="83f879e949c242e1"
                                 culture=""/>
                <publisherPolicy apply="no"/>
                <bindingRedirect oldVersion="1.0.0.0"
                                 newVersion="1.0.0.1"/>
            </dependentAssembly>
        </assemblyBinding>
    </runtime>
</configuration>
```

This sample configuration shows the use of probing, telling the CLR that it's private assemblies reside in the directories *SubDir1* or *SubDir2*, which are subdirectories of the application directory. It also shows the use of *binding redirection*—

applications that want to bind with version 1.0.0.0 of Graphic.dll can bind with version 1.0.0.1 instead, without getting into compatibility problems.

The line **<publisherPolicy apply="no"/>** is worth mentioning. It refers to a publisher policy. A publisher policy file is a special kind of configuration file. It can be issued by the publisher and holds compatibility information regarding a fix or update of an existing component. It is used to let an assembly bind in a proper way with a new version of a component. This publisher policy configuration file resides in the assembly of a shared component. In our example, there can be a publisher policy in the new assembly, version 1.0.0.1. Note that the information in the publisher policy *always* overrides the settings in the application configuration file. The only way to stop this is, as the example shows, to put **<publisherPolicy apply="no"/>** in the application configuration file. This is called *Safe Mode* and is only valid for the assembly it is part of.

Security Configuration Files

Security configuration files describes the security policy settings. There are at least three security configuration files applicable:

- **Enterprise** Resides in the directory %CLR_InstallDir%\Config and is called Enterprise.config.
- **User** Resides in the directory %USERPROFILE%\Application Data\Microsoft\CLR security config*x.x.xxxx* (*x.x.xxxx* is the build number) and is called Security.config.
- **Machine** Resides in the directory %CLR_InstallDir%\Config and is called Security.config.

What these configuration files do and how they are used is described in Chapter 12. For the purpose of the example, take a look at some edited code from the user Security.config file, listing the modifications that were made to it in the exercises in Chapter 12:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <mscorlib>
   <security>
    <policy>
     <PolicyLevel version="1">
      <NamedPermissionSets>
  <PermissionSet class="NamedPermissionSet" version="1"
                 Name="PrivatePermissions"
          Description="My Private Permission Set">
   <IPermission class="EnvironmentPermission" version="1"
                Read="USERNAME;TEMP;TMP"/>
   <IPermission class="FileDialogPermission" version="1"
         Unrestricted="true"/>
   <IPermission class="FileIOPermission" version="1"/>
   <IPermission class="IsolatedStorageFilePermission" version="1"
             Allowed="AssemblyIsolationByUser"
           UserQuota="9223372036854775807"
              Expiry="9223372036854775807"
           Permanent="True"/>
   <IPermission class="ReflectionPermission" version="1"
               Flags="ReflectionEmit"/>
   <IPermission class="RegistryPermission" version="1"
                Read="HKEY_LOCAL_MACHINE"/>
   <IPermission class="SecurityPermission" version="1"
               Flags="Assertion, Execution,
                   RemotingConfiguration"/>
   <IPermission class="UIPermission" version="1"
         Unrestricted="true"/>
   <IPermission class="DnsPermission" version="1"
         Unrestricted="true"/>
   <IPermission class="PrintingPermission" version="1"
               Level="DefaultPrinting"/>
   <IPermission class="EventLogPermission" version="1">
        <Machine name="." access="Instrument"/>
              </IPermission>
              <IPermission class="MessageQueuePermission"
                  version="1" Unrestricted="true"/>
 </PermissionSet>
</NamedPermissionSets>
```

```
       <CodeGroup class="UnionCodeGroup" version="1"
         PermissionSetName="Nothing"
                   Name="PrivateGroup_1"
            Description="">
         <IMembershipCondition class="SiteMembershipCondition"
             version="1" Site="msdn.one.microsoft.com"/>
          <CodeGroup class="UnionCodeGroup" version="1"
            PermissionSetName="LocalIntranet"
                      Name="PrivateGroup_2"
               Description="">
           <IMembershipCondition class="PublisherMembershipCondition"
               version="1"
X509Certificate="3082025A308201C702101DD1CB6CAEA347000491E0419A84A91E300D
06092A864886F70D0101040500305F310B3009060355040613025553312030 1E06035504
0A131752534120446174612053656375726974792C20496E632E312E302C060355040B13
2553656375726520536572766572204365727469666963617469 6F6E20417574686F7269
7479301E170D3031303331353030303030305A170D3032303331353233353935395A3081
80310B3009060355040613025553311330110603550408130A57617368696E67746F6E31
10300E060355040714075265646D6F6E64311230100603550 40A14094D6963726F736F66
7431153013060355040B140C456D6572616C642043697479311F301D060355040314166D
73646E2E6F6E652E6D6963726F736F66742E636F6D30819F300D06092A864886F70D0101
01050003818D0030818902818100BFD980FAD50DBC19919C765F2B80EB84B4336C0FE1CB
979B859AD13E9858276BC28F1B3CD82AC24B6205EFEF05F928AAE5DB45724B805BE97ACD
5334EE24F7BD18AC48B648B8FFBD5DCFF3D6362C1E3DB8514247C6D2069EBA5FA7EE09C9
8428D6EED261E250A80E74894BD36D70712F7FC019E8A40F17832659749FAB87F6B90203
010001300D06092A864886F70D010104050003 7E007DFCF465F5BB7E171028D8D57C1A39
A9F630DE0F3C6F6924A6F5D50D31A096D26208957168E8F3E81BE6A4DD4B04BDD6DF8F22
63C309BE82D4B880CEAC5927BEB386D1DADA736C3F2432B15C7D3A1849BE564AA1B7F4DF
772FC8EE4A41236E0290130DDDE391E115C2103015CB3D4EB6AC91CC72F7F7F4E234E0C9
FA7B"/>
          </CodeGroup>
         </CodeGroup>
    </PolicyLevel>
   </policy>
  </security>
 </mscorlib>
</configuration>
```

WARNING

You should under no circumstance edit the Security.config and Enterprise.config files directly. It is very easy to compromise the integrity of these files. Always use the Code Access Security Policy utility (**caspol.exe**) or the .NET Configuration tool; these will guard the integrity of the files and will also make a backup copy of the last saved version.

Deploying the Application

Although preparing the deployment of an application still calls for a lot of attention, things have become far more easy to handle with the .NET environment. Many people assume that deploying is nothing more than a XCOPY of the application to the destination to get the application up and running. In essence this is true, but it assumes that you have created correct working configuration files, that the CLR is already installed, and that the application is self-contained (does not integrate with other applications). Remember we are still in the Beta phase of a new integrated application environment that gives the Microsoft Windows environment possibilities it never had before. Although the signs are good, we still have to wait for the final verdict until the first full-blown .NET applications are rolled out. To prepare yourself for deploying your first .NET application, we discuss a number of topics that can help you.

Common Language Runtime

In order to run a VB.NET application on a system, it needs to have the .NET runtime environment. At this time it is very likely that a system does not have it installed. Up to the point where it becomes available, remember that you are still working with the Beta, and you will have to install it yourself. You need the .NET Framework Full version, available on the Visual Studio .NET Windows Component Update CD under the dotNetFramework directory with the name setup.exe and residing under the directory dotNETRedist. Installing this version enables you to run all .NET applications. There is also a limited runtime version available, called Control Version that can be used if you use only a Web browser download and run .NET controls.

If (and how) Microsoft is going to deal with licensing and distribution of the .NET runtime environment is not known at this point. It is likely that the .NET runtime environment will become a default component of the Windows XP distribution because they put so much emphasis on the .NET Architecture.

Windows Installer

Using the Windows Installer 2.0 to install a complex .NET application is highly recommended, because this Installer version can recognize assemblies and work with them accordingly. Some of these features are the following:

- Adding and removing, and repairing if necessary, assemblies in the Global Assembly Cache
- Installing and removing, and repairing if necessary, private assemblies in the application's directories
- Rollback of failed assembly operations
- Patching of assemblies

To be able to let the Windows Installer do all the work for you in a controlled way, you need to group all files that directly relate to the assembly. The Windows Installer handles such a group as a single component. If you uninstall the assembly, all files of the component will be uninstalled also. Because assemblies can be used by more than one application, you must prevent the Windows Installer from removing an assembly that is still in use. If you install all assemblies through Installer, this will be no problem because Installer keeps track of the Installer components reference an assembly. It will only remove an assembly if all the components that reference that component are previously removed.

Here is where you should start paying extra attention. Suppose you added a few assemblies to the cache using **gacutil.exe** and one of them references an assembly, let's call it Assembly X, installed by Windows Installer. Times goes by, and Assembly X is uninstalled, but because other assemblies installed by Windows Installer reference X, it was not removed. A bit more times goes by and the last assembly referencing Assembly X is uninstalled. This is noticed by Windows Installer, and it removes assembly X from the cache. The next time the assembly (which you manually installed) runs, it will make a futile attempt to bind to assembly X and fail.

If you are ever confronted with a .NET assembly that breaks, it most likely tried to bind to an assembly that is not available. You can use the Fusion Log

Viewer (**fuslogvb.exe**) to examine where in the loading process things go wrong. By the way, the loading and binding ID is performed by a program called Fusion, hence the name of the viewer.

CAB Files

You can create Cabinet files in a few ways. You can use **makecab.exe** or the deployment tool of Visual Studio .NET. The most important reason to use CAB files is that the compression can decrease the size of the file significantly, thus cutting down on the download time. When you create a CAB file you have to take notice of the following:

- A Cabinet file can contain only one assembly.
- The Cabinet file must have the same name as the file in the assembly holding the manifest. Take our first example, where we had the single file assembly graphic.dll that also holds the manifest. The Cabinet file has to be named graphic.dll. In a lot of cases, the assembly's name is equal to the name of this file holding the manifest.

After you have created the Cabinet files you deploy them, making them available for remote clients through a Web server. You can do this by referencing them through the following:

- A configuration file, using the <codeBase> tag
- A Web page, using the <OBJECT> tag

An example for the <codeBase> may look like this:

```
<configuration>
 <runtime>
  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
   <dependentAssembly>
    <assemblyIdentity name = "graphic"
             publicKeyToken="83f879e949c242e1"
                    culture=""/>
    <codeBase version="1.0.0.1"
           href="http://www.company.com/CABS/Graphic.cab"/>
   </dependentAssembly>
  </assemblyBinding>
 </runtime>
</configuration>
```

An example for the <OBJECT> may look like this:

```
<HTML>
  <OBJECT
      codebase="CABS/Graphic.CAB#version=1,0,0,1">
  </OBJECT>
</HTML>
```

A final remark on the use of CAB files: The first reference to the CAB file will extract the assembly and load it. However, subsequent references to the assembly will fail because the CLR does not automatically expand Cabinet files.

Internet Explorer 5.5

Internet Explorer 5.5 and above can help you in deploying assemblies. You can reference a managed executable, hence assembly, from a Web page and it will be downloaded and executed. You can embed this in an installation guide that embodies all information that is needed to install the application.

Some issues surrounding this method of deployment are related to security policy and application domain. If you load an assembly from a Web site, the zone of the Web site, which is *Internet*, is used as evidence to determine the permission set. If the administrator did not make changes to be more specific with permissions, for example *site* or *strong name*, then these assemblies are assigned the limited permission set of *Internet*. To establish a broader permission set, you have to create appropriate Code Groups (see Chapter 12).

If you reference a Web site, the runtime creates an application domain for that site, for example www.company.com. If an application domain already exists for that site, the assembly referenced by this page will be added to that AppDomain. As long as the Web site holds assemblies that belong to one application, assemblies sharing one application domain are not a problem. However if the site holds more than one application, for example http://www.company.com/app1 and http://www.company.com/app2, this may be an unwanted situation. You can solve this by placing a <LINK> tag on the Web page that points to an application configuration file. Based on that configuration file, an AppDomain is created and all assemblies on that page run in this application domain context. This is also the case if more than one Web page has a <LINK> tag pointing to the same application configuration file. Having mentioned this, a warning is in order: If you forget to supply the <LINK> tag to a page, the assemblies of this page are loaded in the site-related application domain. This will ultimately result in problems with the execution of the application.

In the "CAB Files" section earlier in the chapter, the use of the <OBJECT> tag was discussed. However, there are two other ways of referencing to an assembly that gets downloaded and loaded into a new application domain:

- Using a HREF link, for example <A HREF="Graphic.exe">
- Pointing your browser directly to the assembly, for example by entering http://www.company.com/EXECS/Graphic.exe

In the first option, it is assumed that the assembly resides in the same directory that the Web page is pointing at. The CLR will also check this directory for a configuration file. It is also assumed that the application base (AppBase) is set to the directory the Web page is pointing at. In the case of the HREF link, this may be www.company.com/App1. In the example of the second option, this is www.company.com/EXECS. The AppBase is seen as the root directory from which the CLR searches for subsequent referenced assemblies.

NOTE

As you install Visual Studio .NET Beta, the installation process first wants to install a few "enhancements" before it installs the actual Visual Studio .NET. One of these enhancements is the Installation of Internet Explorer 6.0 (Public Beta). The reason is that this version of Internet Explorer has all the features to deal with advanced .NET applications. Take note that you must be very cautious with installing IE 6.0 on your regular system if you use it for tasks other than developing. Experience has shown that some e-commerce Web sites have problems with IE 6.0, hence are so well-tuned on IE 5.5. And because IE 6.0 is also still a Beta, it may contain some incomplete functioning code.

Resource Files

A resource is a nonexecutable data file that is related to an assembly and packaged in a resource format, so it can be automatically deployed together with the assembly. A few examples of resource files would include Help text for the applications, icons, images, and sound-files. The most important advantages of resource files are as follows:

- Resources can contribute to a better localization of your application; by using the Culture identifier of the assembly, the correct localized resource is picked.
- Resource files make access to different types of data files uniform by using resource objects.
- Resources can be part of an assembly file or compiled into a satellite assembly.

Localization has always been a hassle in selecting the correct localized files. Now this is handled by the CLR, presuming that you have packaged your resource files in assemblies. If the Culture field is not empty, also called *neutral,* the CLR checks the localization settings of the system and tries to find a referenced assembly with the correct culture. If this is not the case, it will fall back on the neutral version. This implies that when you use localized assemblies, you must always have a neutral assembly available. If no localized assembly exists that matches the localization setting of the system, the CLR always falls back on the neutral version. In case the CLR cannot find the neutral version, it throws an exception, and the loading process will fail. Also, for localized resource files, the principal of satellite assemblies is used. So what about satellite assemblies?

They are called *satellite* because they do not contain any executable code. They must be in close contact with an assembly that does contain executable code and use the resources that are contained in the satellite assemblies. This concept is also referenced to as *hub and spoke,* where the executable assembly is the *hub* and the resource assemblies the *spokes*. This solution has the following advantages:

- You can very easily add other localized versions to the application by simply copying them to the correct directory.
- You have to deploy only the localized versions that will be used.
- Satellite assemblies can be replaced without having to change or (partially) recompile the application.

The only drawback is that you must test your application against every localized resource set.

NOTE

The localized resource assembly has to follow a specific naming convention. It is preferred that you use the Culture field to identify the localization signature, although the CLR can also check the assembly name for the localization signature. The format is *ll-CC*, what stands for two lowercase characters for the language, followed by a dash, followed by two uppercase characters for the country. These codes are standardized in the ISO 3166 standard. Be sure to use these codes because there are some exceptions to the rule. A few examples are the following:

- **en-UK** English as spoken in the United Kingdom
- **en-US** English as spoken in the United States
- **du-BE** Dutch as spoken in Belgium
- **fr-BE** French as spoken in Belgium

Let's take a look at how you create the resource files from the command line. The following code is a sample that comes with .NET Framework SDK and is located in <*SDK_InstallDir*>\FrameworkSDK\Samples\tutorials\resourcesandlocalization\graphic\vb.

The build.bat file, which is edited for the sake of readability, reads this way:

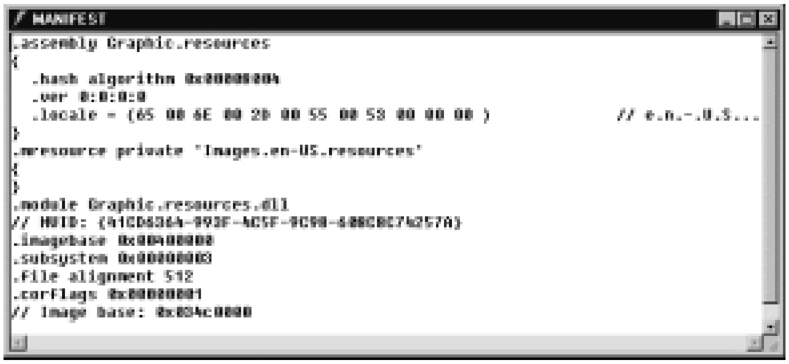
```
resxgen /i:un.jpg     /o:Images.resx /n:flag
cd en
resxgen /i:en.jpg     /o:Images.en.resx /n:flag
cd ..\en-us
resxgen /i:en-US.jpg  /o:Images.en-US.resx /n:flag
cd..
resgen Images.resx Images.resources
resgen en\images.en.resx en\images.en.resources
resgen en-us\images.en-us.resx en-us\images.en-US.resources

al /out:en\Graphic.resources.dll /c:en
      /embed:en\Images.en.resources,Images.en.resources,Private
al /out:en-us\Graphic.resources.dll /c:en-US
     /embed:en-us\Images.en-US.resources,Images.
         en-US.resources,Private
```

```
vbc /target:library /optionstrict+ /r:System.DLL
/r:System.Drawing.DLL
    /r:System.Windows.Forms.DLL /r:System.Data.DLL
    /res:Images.resources,Images.resources graphic.vb
```

You see three commands that you most likely never encountered before:

- **Resxgen** A utility that converts a JPG image into a RESX file. The latter is an XML-coded resource file. You need this utility to be able to generate a resource file for the JPG-image.
- **Resgen** The command line Resource Generator utility that converts the RESX file to a .resource-file. In fact, it can convert an input file with the .txt, .resource, or .resx extension to an output file with a .txt, .resource, or .resx extension, assuming that the file extension represents the format of the file.
- **Al** The Assembly Generation utility, but *al* is short for assembly linker. It is used to generate an assembly (including a manifest) from one or more MSIL files (without a manifest) or resource files. Let's take a look at the parameters in our example:
  - **/out** Gives the exact name of the output file.
  - **/c** Gives the culture string that has to be attached to the output file. By the way, **/c** is short for **/culture**.
  - **/embed** The full syntax is: **/embed[resource]:*file*[,*name* [,private]]**, whereby ***file*** the name of the resource file that has to be embedded in the output file, containing the manifest; ***name*** is the internal identifier that will be used in the assembly to reference the resource; **private** takes care that the assembly can only be used by the application it is meant for and therefore will not be visible for other assembles. In Figure 13.4 this all comes back in the line **.mresource private'Images.en-US.resources**.

Figure 13.4 The Manifest of the Resource Assembly en-US/ Graphic.resources



```
.assembly Graphic.resources
{
  .hash algorithm 0x00008004
  .ver 0:0:0:0
  .locale = (65 00 6E 00 2D 00 55 00 53 00 00 00 )          // e.n.-.U.S...
}
.mresource private 'Images.en-US.resources'
{
}
.module Graphic.resources.dll
// MVID: {41CD6364-993F-4C5F-9C98-6A8C8C7A257A}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
// Image base: 0x034c0000
```

Deploying Controls

Up until now, we have discussed the deployment of assemblies, or managed code, in general. When you start deploying .NET controls, you come across additional issues that are similar to the issues you had to solve when you deploy ActiveX controls, although you no longer have to bother with Registry settings and CLSIDs (as long as your controls do not interact with COM+ components). Let's first list the steps involved in the deployment process and then discuss them in further detail:

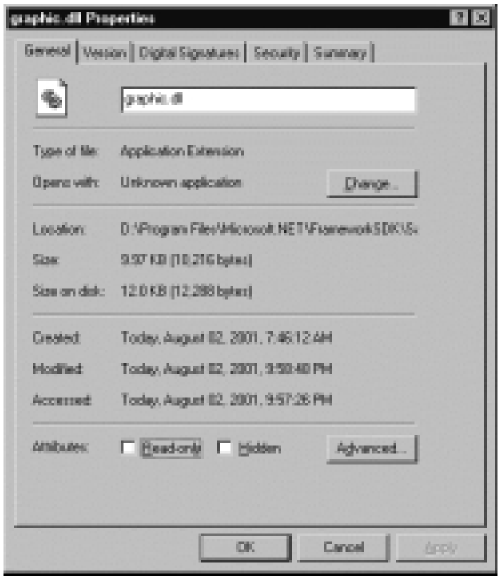
1. Obtain a X.509 Authenticode Certificate from a CA, such as Verisign, or use a Software Publisher Certificate (SPC).
2. License your .NET control.
3. Sign the .NET control assembly.
4. Package the .NET control in a CAB file.
5. Make an application configuration file.
6. Create the Web page that makes an <OBJECT> reference to your CAB file and a <LINK> reference to the application configuration file.
7. Test it.

Step 1 speaks for itself. For testing purposes, you can make use of the **makecert.exe** tool. Step 2 involves the always important issue of software licensing. The .NET Framework comes with a License Compiler utility (**lc.exe**), which creates a .licenses file that will be included in the assembly as a .resource file.

Step 3 can be done using the File Signing tool (**signcode.exe**). Although it is a command-line utility that requests a whole series of options to be filled in, you should locate the file with the Windows Explorer and double-click it. The Digital Signature Wizard will be started, which takes you through the whole process in an easy and straightforward way. You can check if the assembly is indeed signed by performing the following steps:

1. Use the Windows Explorer to locate the assembly you just signed.
2. Right-click the file and select **Properties**.
3. Select the **Digital Signature** tab (see Figure 13.5).
4. Select the Certificate in the **Signature List** and click on **Details** to view the Certificate. It should look familiar.

Figure 13.5 The Proof of a Signed Assembly



graphic.dll

Type of file: Application Extension
Opens with: Unknown application
Location: D:\Program Files\Microsoft.NET\FrameworkSDK\S
Size: 9.97 KB (10,216 bytes)
Size on disk: 12.0 KB (12,288 bytes)
Created: Today, August 02, 2001, 7:46:12 AM
Modified: Today, August 02, 2001, 9:50:40 PM
Accessed: Today, August 02, 2001, 9:57:26 PM
Attributes: ☐ Read-only ☐ Hidden

Steps 4, 5, and 6 have already been discussed. Step 7 should speak for itself—testing has always been something that does not get the highest priority. But it cannot be emphasized enough: Testing is a very important step in the development *and* deployment process of an application. It doesn't matter whether you're dealing with a large and complex application or a single control. Never cut back on testing—you earn it back in all the hours you don't have to spend on troubleshooting.

Summary

With the .NET Framework application deployment has become easy again—no battles with DLL versions and Registry settings. You are able to focus on what is really important: going smoothly through the deployment phases. The first step in deploying is *versioning.* This is sort of new from what you were used to. But now you can run multiple versions of the same assemblies next to each other. It is important that assemblies have correct version numbers so that assemblies can bind with correct assemblies. To achieve this, it is important that assemblies get full names, consisting of a name, strong name, culture (also known as locale or localization), and a version number. The use of full names is also necessary to share them. After this is taken care of, you can go to the next step, which is the way the assemblies are going to be packaged for deployment. There are three choices: do not package them, just XCOPY them; package every assembly in a Cabinet file; or package the complete application in a Windows Installer file. The packaging method depends through which channels you want the application to distribute. Assemblies attribute also to the organization of data. All programs need different kinds of resources, such as Help files, images, and audio files. By converting these files into resource files, you can package them in assemblies without executable code. These are called satellite assemblies, which are very useful to solve deploying localized applications. By given them a culture identification, the CLR will automatically pick up the resource assemblies with the correct localization. Localized assemblies can be added without having to bring the application down or to restart it.

Although you do not need to get involved in all kinds of registering of DLL files, using Class Identifiers and other Registry settings, a lot of configuring is still going on. Only now, they are different types of XML-coded configuration files. Every application can have its application configuration file; every machine has its machine configuration file (also called an administrator configuration file) and security configuration files. Every possible setting is controlled by these configuration files and enables you to go from a generic configuration to a tailor-made configuration.

Before you can actually deploy an application, you must be sure that the machines that run the application indeed have the .NET Framework runtime installed. When you use Windows Installer, it can take a lot of work out of your hand installing, and also uninstalling, of the application, especially because Windows Installer recognizes assemblies and knows how to handle them, such as installing them in the general assembly cache. CAB files are especially useful if you want to distribute the assemblies using the Internet Explorer 5.5 (and up). When you want to deploy .NET Controls, which you can compare with ActiveX Controls, issues such as licensing and signing are involved.

After deployment of an application, you do not have to reboot the system, and you can easily replace assemblies, just by adding a new version of the assembly to the assembly cache. You can modify configuration files, and the next time an assembly is loaded by the CLR, the modified configuration file is used.

Solutions Fast Track

Packaging Code

☑ An assembly has a Manifest that describes in detail what is wrapped in the assembly and with which other assemblies it will bind.

☑ An assembly is identified by its full name that consist of a name, string name, version number and culture (localization identifier).

☑ A single assembly can be packaged in a Cabinet (CAB) file, using the Cabinet Maker (**makecab.exe**).

☑ A complete .NET application can be packaged in a Windows Installer file.

Configuring the .NET Framework

☑ Application dependent settings can be set in the application configuration file. This file applies to all assemblies that are part of an application.

☑ Machine dependent settings can be set in the machine/administrator file. This file applies to all applications that run on this machine. Settings in the application configuration file can not override the settings in the machine configuration file.

☑ There are three security configuration files: enterprise, which applies to one or more systems and applications; user, which is related to the user that starts up an application (every user has his own security configuration file; and machine, which applies to a single machine that runs one or more applications (every machine has its own security configuration file).

Deploying the Application

☑ Before a .NET application can be deployed, the .NET Framework runtime system must be installed on all the machines that will run (a part of) the application.

☑ With Windows Installer 2.0, a complete .NET application can be installed and, if needed, be uninstalled later on. Windows Installer 2.0 is fully assembly-aware and can take care of placing and updating assemblies in the general assembly cache. Windows Installer 2.0 keeps an independent administration of assemblies it has installed and are referenced by other assemblies installed by Windows Installer 2.0; it will never remove a uninstalled assembly if it is still referenced by other assemblies.

☑ With CAB files, you not only compress an assembly significantly, but it can also be deployed using Internet Explorer 5.5 (or higher).

☑ The nonexecutable data files—such as Help files, audio files, and images—that are used by assemblies are converted in resource files (using the Resource Generator utility **resgen.exe**) that can be added to assembly files that hold executable code. They can also be compiled into an assembly without executable code, using the Assembly Generation utility (**al.exe**). This is called a satellite assembly, which are very useful in deploying localized applications.

Deploying Controls

☑ You can protect your controls by licensing them, using the License Compiler utility (**lc.exe**).

☑ You can authenticate your controls using a X.509 Authenticode Certificate, with the File Signing utility (**signcode.exe**), that comes with a Digital Signature Wizard.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: If I can XCOPY a .NET application to its destination, why should I use Installer instead?

A: You don't need to use Installer; XCOPY will work just fine. But when you need to deploy a more complex application, you may want to do a little more than just an XCOPY. The most obvious is that you need to install shared assemblies in the general assembly cache (GAC). Additionally, if you want to uninstall the application, removing the sub tree will not remove assemblies from the GAC. Besides how do you know if you can safely remove an assembly from the cache without breaking another application? If you install all your application with Windows Installer, Installer will take care of removing an assembly when it is safe to. That is, if you install all the applications using Installer 2.0. It is a good practice to be able to install an application with an as small as possible footprint *and* uninstall applications without leaving a footprint. Packaging your application with Windows Installer takes all that work away from you and the people who are responsible for the installation and maintenance of the application. By the way, it looks far more professional than copying files from a CD.

Another reason to use Windows Installer 2.0 is that you may have to integrate with applications that run outside the .NET Framework or even ship unmanaged components with the application, that still have all the installation issues attached to it. It will help you a lot to keep using just a single method of deployment and Windows Installer2.0 is not a bad choice to do so.

The XCOPY example is merely used as to imply that .NET applications are no longer plagued with the hassles of checking for the proper DLL version, using RegSvr32 to get your files registered, fixing CLSID problems and all these other annoyances. On the other hand, you're not out of the woods right now! All these problems still apply dealing with the .NET Framework runtime system and when there become more versions of this runtime system available you may have to deal with all the problems all over again, because application may need a specific runtime version to run in. If Microsoft is not able to make it possible to run different CLRs next to each other and independent from each other, we may have to deal with all the issues all over again.

Q: What is the best way to deploy a localized application?

A: The best part of localization and the .NET Framework is that it uses the systems "Regional Options" to determine the "Culture", in which the culture is a combination of language (two lowercase characters) and country/region (two uppercase characters). Compilation of .NET applications is language independent and no longer poses the same localization problems. Because culture is part of the assembly's full name, assemblies with a culture set always take precedent over assemblies that have no culture, called neutral. The Culture property at least has to hold the language, but you are advised always to be as specific as possible. This may help you to deploy your application in as many localization settings as possible, without having to replace files later on. Although the .NET Framework helps you a lot with easy localization implementation, it is the program's design and development that decides to what extent your application is able to handle localization. For example, address labels have another format in about every country. The .NET Framework will not help you to solve that problem—that's up to the program. But if you create your code so that it can format an address label based on a set of rules, you can save these rules as a resource in a localized assembly. Your code just has to read the rules, because the CLR takes care that the assembly with the proper localization is presented to the application.

Besides setting the culture of the assembly, put the culture in the name of the resource (for example, Graphic.en-US.resources) because the CLR will use this to determine the proper localized resource in case the Culture property is not set. The last thing you should do is place all resources of the same culture in a subdirectory that should be located directly under the application's root directory and must be named with the exact culture's name. The CLR will check this directory to locate the proper localized resource. If you adhere to these things, adding a new language to your application is as easy as copying a directory.

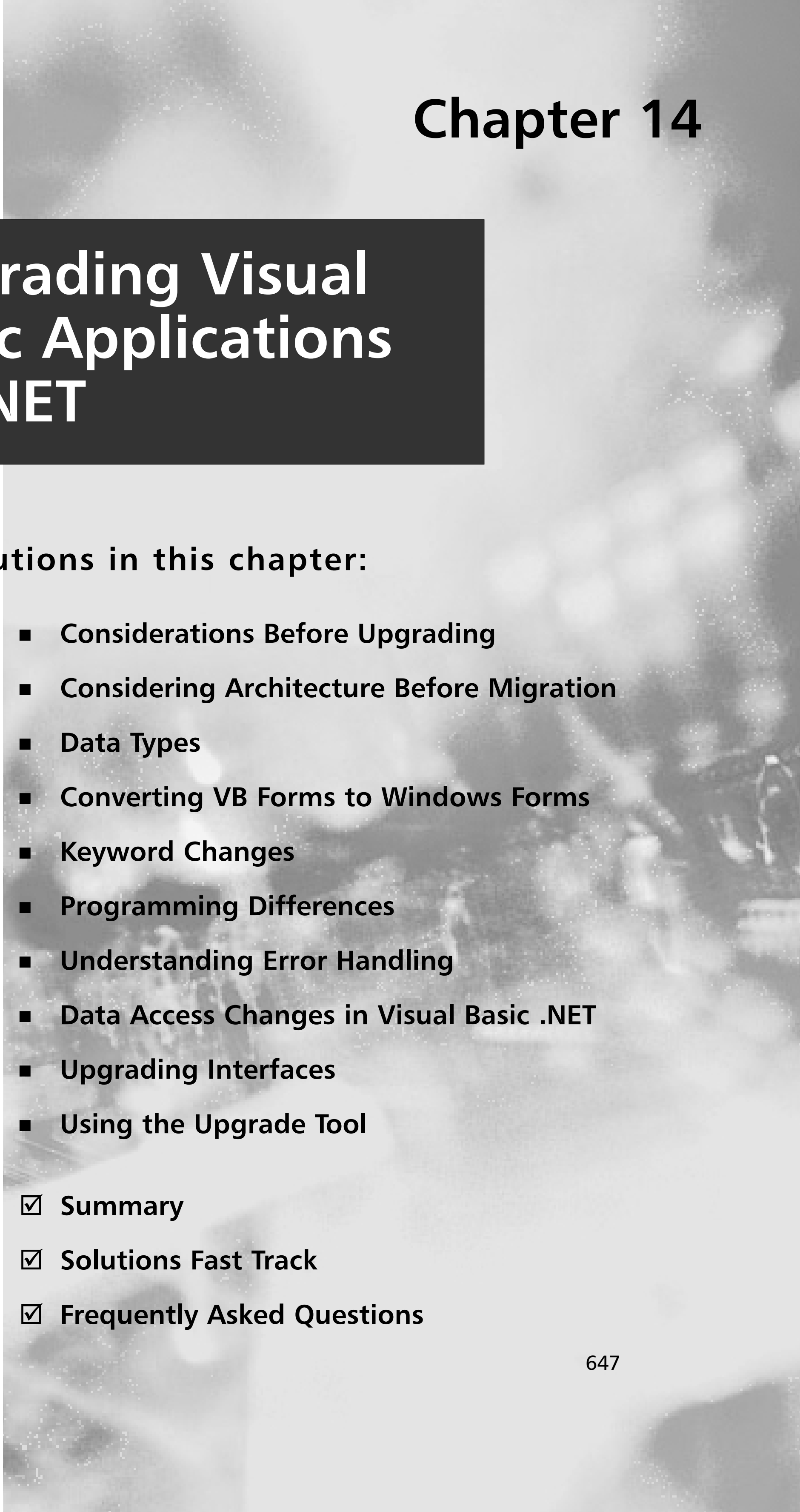
Q: How do I transfer an image file to a resource file? Using **resgen.exe** doesn't work.

A: This seems to be a odd problem:You need to use **resgen.exe** to create a resource file, only it excepts just two different formats— TXT files and RESX files.The latter is an XMLized version of a resource. In fact, it is advised to convert every resource into its XML-coded version.The reason is that the whole .NET Framework is optimized in using XML-based input. You can use streams to take care of all the formatting and stuff.That leads to the problem that **resgen** is not able to convert an image to a RESX file. Let's hope that Microsoft comes up with a more versatile **resgen** that can handle more types of input. Until then, you can use a program called ResXGen with the C source included that is able to do the trick.You can find it in the samples section, under tutorials\resourcesandlocalization, of the Framework SDK. You can use this .NET program or let the source help you to create your own conversion program.After you have run your image file through this program, you have the XMLized (RESX) version of your image that can be converted to a resource file using **Resgen.exe**.

Q: Is there a method to use version numbers for assemblies?

A: There is no standard method of keeping track of version numbers. But because the format of the version number is very clear, you have to go with that.As long as you go with that format, you are in the clear. Change version numbers only if it is really necessary.The reason is very simple:The CLR considers every version number as a valid assembly and will check it as it tries to find the correct version of an assembly.The more assemblies there are to check, the longer this will take. For example:You have built a reasonably extensive application, and you are now on version 2.0 (the major and minor part of the version number) and want to change it to 2.1. But because you've done a great deal in the development process, only half of the assemblies have changed. If you would change the version number of all assemblies, you double the number of assemblies while half of them are the same. Only by removing the obsolete ones can you control the number of active assemblies. This is no problem if the assemblies are used only by your own application, but if you share the assemblies with other programs, this is not so straightforward.There are ways around it.The best way is to edit the application configuration file and put a <assemblyredirect> in for all the unchanged assemblies and let the old versions point to the new one. Be careful with using publisher policies to take control of these issues. Applications can run in Safe Mode, thereby ignoring the publisher policy. As far as the build and revision goes, change these only on a per-assembly basis.

There is another way of dealing with a lot of assemblies with version changes by going for a select number of large assemblies, instead of with a lot of small ones. This makes good sense if you can group a lot of code based on a common functionality in relation to deploying such functionality. Be sure that it has a limited number of exposed interfaces, or you will run into version trouble again if the assembly is referenced all over the place.



Chapter 14

Upgrading Visual Basic Applications to .NET

Solutions in this chapter:

Introduction

Now that we have seen how to develop an application with Visual Basic .NET, what about your existing applications? Do you leave them as is, or should you upgrade them? There are different factors to consider when making this decision. This chapter will focus on when to upgrade your application, and what is involved in an upgrade when you decide to do it.

The .NET architecture is different from previous versions of Visual Basic and some applications will require significant changes. Visual Basic .NET has transitioned to making everything an object, prompting significant programming changes. You will have to modify data types to match the new Common Type System (CTS), which will impact applications using the Variant data type. There are other considerations. Some keywords have been changed or even removed, and your Visual Basic Forms will need to be upgraded to Windows Forms. Error handling will be completely different as well, and you will have to convert all existing error handling to the new exception-based format.

As you have seen, data access has changed drastically and is based on XML. This is a major paradigm shift, and addressing these issues can take considerable time and effort. Interfaces and events have changed from previous versions of Visual Basic. Not all applications use these features, but you need to understand how to convert them if they do. Fortunately, Microsoft has created an upgrade tool. This will attempt to upgrade your application from Visual Basic 6.0 to Visual Basic .NET. Of course, not all aspects of your application can be automatically upgraded. The portions of your application that cannot be upgraded will be commented out and will require manual conversion. Some applications will still necessitate a large development effort, so think long and hard before an upgrade and develop a plan for it.

Considerations Before Upgrading

Certain issues must be addressed before a legacy application can be upgraded to Visual Basic .NET. Changing your application could mean anything from changing the data type of a variable to rewriting an entire application so it uses a different data access mechanism. The following points are recommendations you should carefully look at before deciding on migration. We will cover these recommendations in detail in the sections that follow:

- Early binding of variables
- Avoiding Null Propagation

- Using ADO
- Using the Date data type to store dates
- Using constants instead of actual values

Early Binding of Variables

Visual Basic .NET, like Visual Basic 6.0, supports late-bound objects. *Late binding* an object is the practice of declaring a variable to be of the data type *Object* and assigning it to an instance of a class at runtime. Early binding also refers to the practice of declaring a variable a specific data type other than type *Object*. The advantage of early binding is that compiler errors, like using incorrect properties or calling non-existent methods, can be immediately detected. This is made possible by using type library. Whenever you set a reference to a type library, all the classes that are part of the component are available at design-time itself.

The usage of late binding objects has its disadvantages. The main drawback is the inability to enumerate the type library during design-time. During an upgrade process, late-bound objects can cause problems because of changes in property names and the removal of the default property feature from the controls. This is especially true if your legacy applications use code that late binds Windows controls, like labels and command buttons.

The reason is that some of the property names have changed in Visual Basic .NET, the most important being the name change for the Caption property. It is now called Text. Suppose you had a form with the CommandButton control on it. The following Visual Basic 6.0 code declares a variable of data type *Object*, assigns a command button object to it, and sets a value to its *Caption* property:

```
Dim obj as Object
Set obj = Me.Command1
obj.Caption = "Ok"
```

The upgrade tool normally converts all references to the Caption property to Text property. Since the preceding code uses late binding, the upgrade tool cannot determine what type of object is being referenced here. Therefore, it won't have a clue as to how the properties must be translated. As a result, the upgrade tool marks these statements with an upgrade error. In such cases, you will have to change the code yourself.

In order to ensure a successful migration, make sure all variables are declared a specific type other than Object. The following code will migrate successfully:

```
Dim objCmd as CommandButton
Set objCmd = Me.Command1
objCmd.Caption = "Ok"
```

The practice of using late binding also affects Visual Basic 6.0 components that implement classes and interfaces. In Visual Basic 6.0, it is possible to assign an interface reference to a variable of type *Object* to access the properties and methods of the interface. The following code shows this implementation:

```
Dim IMyInteface as Interface1
Dim clsMyClass as Class1      'Assuming Class1 implements Interface1
Dim obj as Object
…………
…………
Set IMyInterface = clsMyClass 'Gets a reference to the ImyInterface
Set obj = ImyInterface        'Assigns the ImyInterface reference to obj
obj.property1 = Value          'Accesing a property
```

Unfortunately, in Visual Basic .NET, you can only late bind to public members of a class, not to interface members. Therefore, the previous code must be edited so the *object* variable directly references the class, not the interface, as shown next:

```
Set obj = clsMyClass        'Assigns the ImyInterface reference to obj
obj.property1 = Value       'Accesing a property
```

Avoiding Null Propagation

Null propagation means that if *Null* is used in an expression, the resulting expression is always Null. In previous versions of Visual Basic, the *Null* value disseminated throughout the expression.

Null propagation is commonly used in database applications where you need to check for a Null in a specific field or fields. The following expressions show you how a Null is propagated in an expression:

```
Dim var
var = 100 + Null
var = "hello" & Null
```

Null propagation is not supported in Visual Basic .NET. Consequently, the statement var = 100 + Null will result in a type mismatch error. Moreover, the **Null** keyword has been replaced with System.DBNull.Value. So, for the purpose of successful migration, your code should always test for Null instead of for Null propagation. Visual Basic uses the IsDBNull() function to determine if an expression contains a valid value or Null.

Using ADO

Visual Basic .NET supports DAO, RDO, and ADO code, but with some slight modifications. Visual Basic .NET, however, does *not* support DAO and RDO data binding to controls, data controls, and the RDO user connection. So, if a data access application has DAO or RDO data binding, it is better to upgrade it to ADO before migrating to Visual Basic .NET. This section aims to give you an overview only. More detailed discussion will come later in the chapter.

It is possible to run existing data access applications that utilize ADO by using Visual Basic .NET with very minor modifications. In order to accomplish this, right-click the **Reference** node in the Solution Explorer and choose **Add Reference**. From the **References** window, choose **ADO library** from the supplied list of registered COM components.

What occurs next—behind the scenes—is quite elaborate. There is a tool called TLBIMP.EXE, which is shipped with the .NET Framework, that generates an assembly containing regular .NET metadata based on the content of the specified COM-type library. The following command imports the ADO object model into .NET:

```
Tlbimp.exe msado15.dll
```

On executing this command, the tool creates a file called adodb.dll in the current folder. The name of the output file can also be specified using the /out: option.

Once the library is imported, all the ADO classes are available to the .NET code as native classes. Using Visual Basic .NET, a typical data access application that fetches rows from a table can be coded as follows:

```
String strSQLConn = "Provider=SQLOLEDB;Initial" +
   "Catalog=pubs;Server=localhost;UID=sa;PWD=;"
      Dim objCn As New ADODB.Connection()
      Dim objRs As New ADODB.Recordset()
```

```
        objCn.ConnectionString = strSQLConn
        objCn.Open()

        objRs.Open("Select au_lname from authors", objCn,
    ADODB.CursorTypeEnum.adOpenForwardOnly,
    ADODB.LockTypeEnum.adLockReadOnly)

        While Not objRs.EOF
            Debug.WriteLine(objRs(0).Value)
            objRs.MoveNext()
        End While
        objRs.Close()
        objCn.Close()
```

Using Date Data Type

In Visual Basic 6.0, you could use the Double data type to store and manipulate dates. This is not supported in Visual Basic .NET, however, because dates are not stored as doubles. Therefore, the following code is invalid in Visual Basic .NET:

```
Dim dblVal as Double
Dim dtVal as Date

dtVal = now
dblVal = dtVal    'Invalid in Visual Basic .NET
```

The .NET Framework provides two methods that do the conversion between dates and doubles. The functions are FromOADate and ToOADate. The ToOADate function converts a Date type value to a double and the FromOADate converts a double value to Date.

During an upgrade operation, it becomes very difficult to determine what the code is trying to do when it uses double data type to store dates. In order to do away with unwanted changes to your code, use the Date data type to store dates.

Using Constants

It is a good programming practice to use constants rather than the actual values that represent the constants or variables that store these values. In Visual Basic

.NET, the value of True has been changed from –1 to 1. The usage of constants ensures that the correct values are replaced when your project is upgraded. However, if an actual value is used, it is quite possible your project will be upgraded properly.

Considering Architecture Before Migration

This section discusses what changes are to be made to your existing applications before moving to the .NET platform. The migration to the .NET platform has its own advantages. It is a quantum leap from the previous architectures and provides extended support for scaling applications. The key highlights are disconnected data access and resolution of the DLL hell problem by implementing a file-copy-based deployment of components. To take advantage of these benefits, your existing applications must be modified. Microsoft has provided a migration tool that will make your applications .NET compatible. Not all applications can be readily modified. Certain projects that will remain the same because of lack of support in Visual Studio .NET.

The existing applications can be broadly classified into the following categories, which we will discuss in detail in the sections that follow:

- Internet/Intranet Applications
- Client/Server Applications
- Single-tier Applications
- Data Access Applications

Intranet/Internet Applications

Visual Basic 6.0 provided the following project types to build Intranet/Internet applications:

- Internet Information Server (IIS) Applications
- DHTML Applications
- ActiveX Documents

Each of these application types was unique in their way and helped developers build solutions best suited to the scenarios for which they were built. But, over a period of time, DHTML applications and ActiveX documents were used less and less because none of these application types were extensible. The introduction to Web forms and Web classes in the .NET architecture aims to increase application compatibility and enhance the functionality of Internet or intranet applications. Web forms are used to build feature-rich Web applications. Though the functional aspect of Web forms remains the same, it offers a variety of benefits. For example, the Web forms framework captures and stores information input on a form and makes it available as object properties. Web application services like these make Web forms unique. A separate section is devoted in this chapter to Web forms.

Internet Information Server (IIS) Applications

A *WebClass* is the building block of an IIS application. It is a Visual Basic component, and as such, resides on a Web server, responding to input from the browser. WebClasses, however, do not exist in Visual Basic .NET. The migration tool upgrades all WebClass applications to Web forms instead. As a result, migrated applications have to undergo some modifications before they are ready to run. It is also possible to navigate from a Visual Basic .NET Web form to a Visual Basic 6.0 WebClass.

Developing & Deploying…

Web Forms in ASP.NET

Visual Basic .NET introduces an enhanced version of ASP (Active Server Pages) called ASP.NET, ushering in a new programming model to build powerful Web applications.

Web Forms is an ASP.NET technology useful in creating programmable Web pages. Listed next are some of the highlights of Web Forms:

- Web Forms can be programmed using any of the Visual Studio.NET languages, like C#, Visual Basic, and so on.
- Web Forms can run on any browser and render browser-compliant HTML.
- Web Forms support user-created and third-party controls.
- Web Forms support managed execution environments, type safety, inheritance, and dynamic compilation.

DHTML Applications

DHTML applications typically house DHTML pages and client-side ActiveX DLLs that contain the business logic. DHTML applications cannot be upgraded to Visual Basic .NET.

ActiveX Documents

ActiveX documents, like DHTML applications, cannot be upgraded to Visual Basic .NET. The only recommended option is to replace ActiveX documents with user controls. Despite this, both ActiveX documents and DHTML applications can interoperate with Visual Basic .NET Web forms.

NOTE

Microsoft recommends implementing multi-tier architecture when building applications, so migration is easier. The architecture involves building the user-interface through ASP, and the business logic using Visual Basic 6.0 or Visual C++ 6.0 component. ASP is fully supported in Visual Basic .NET and the business components can either be upgraded to Visual Studio .NET or used as is.

Client/Server and Multi-Tier Applications

Multi-tier projects typically house Visual Basic Forms, containing embedded user controls, and middle-tier business components. The middle-tier components can either be a Microsoft Transaction Server (MTS) component or a COM+ component. Client/Server applications contain just two layers: the user-interface layer (also called the client), and the database layer (called the server). The business logic is embedded in either the client-side or server-side.

Visual Basic Forms has been replaced with Windows Forms in Visual Basic .NET. The object model of Windows Forms is different from Visual Basic 6.0 Forms. The good news is that the object models are compatible. The Upgrade Wizard converts Visual Basic Forms to Windows Forms during an upgrade operation. User controls are then upgraded to Windows controls, however, custom property tags and accelerator key settings are not upgraded.

Middle-tier components can remain the same, but it's advisable to upgrade them to .NET as well. This begs the question of debugging, however. How can I debug a component written in Visual Basic 6.0 while in a Visual Studio.NET environment? Visual Studio.NET provides a single integrated debugger for all Visual Studio languages. The unified debugger goes beyond supporting components written for the .NET Common Language Runtime (CLR). It also supports debugging Win32 native applications and this includes MTS/COM+ components written in Visual Basic 6.0. The only caveat is that they must be compiled to native code, with symbolic debug information, and should not include any optimizations.

Visual Basic .NET also introduces a new middle-tier component called Web Services. A Web Service is a component that contains business logic and is hosted by ASP.NET. They use HTTP methods as their transport mechanism and pass and return data using XML. The use of XML allows heterogeneous systems to interact with the Web Service, the only limitation being that Web Services does not support distributed transactions.

Single-Tier Applications

Single-tier applications can be classified as:

- Add-ins
- Miscellaneous utility programs

Visual Basic .NET is now an integrated part of the Visual Studio .NET Integrated Development Environment (IDE). This has necessitated a change in the extensibility model used in Visual Studio .NET. Applications employing the Visual Basic 6.0 IDE model cannot be migrated to exploit the advantages of the Visual Studio .NET extensibility model. The new IDE extensibility model is generic for all project types supported in Visual Studio .NET. The add-ins that can be created using the new model can be shared by any of the Visual Studio .NET supported languages.

Miscellaneous utility programs like those which perform file operations or manipulate registry functions upgrade without any problems. The migrated applications can then take advantage of many of the new features available in Visual Basic .NET, like structured exception handling, free threading, and so on.

Data Access Applications

Data Access applications typically use the following methods to perform data manipulation:

- ActiveX Data Objects
- Remote Data Objects
- Data Access Objects

Visual Basic .NET introduces ADO.NET, an enhanced version of ADO. ADO.NET provides performance enhancements over ADO and aims at disconnected data. A separate section at the end of this chapter is devoted to highlighting the differences between the two.

DAO, RDO, and ADO applications can still be used in Visual Basic .NET after making some minor modifications. Visual Basic .NET doesn't support data binding to DAO or RDO controls. So if any of your applications use data binding, it is best to leave them to Visual Basic 6.0, or port the code to ADO before migrating to Visual Studio .NET. Data binding is still available in VB.NET and is implemented with the help of the *Binding* class.

Data Types

Visual Basic .NET has not only brought in language enhancements but also changed the way we work with data types. It is imperative the programmer have adequate information on the changes made to ensure a successful and smooth migration to Visual Basic .NET. This section is devoted to dealing with changes that have been effected in Visual Basic .NET.

Variants

Variant is a special data type. What makes the Variant data type so unique is that it can be assigned to any primitive data type such as Empty, Nothing, Error, and Null. A primitive data type is one that is supported by the compiler natively. The only limitation with the Variant data type is that it cannot be assigned to fixed-length strings.

Visual Basic .NET uses the *Object* data type that effectively replaces the Visual Basic 6.0 Variant data type. In fact, the functionality of both *Object* and *Variant* data types has been combined into the new *Object* data type. The *Object* data type can be assigned to any primitive data type, *Empty*, *Nothing*, *Error*, *Null* and as a pointer to an object. The default data type in Visual Basic .NET is *Object*.

When a project is migrated to Visual Basic .NET, all variables of type *Variant* are converted to *Object*. It is a better programming practice to declare variables a specific data type before beginning the upgrade process. Not only does this help identify the type of data these variables will store, but it will also result in less ambiguous code after the upgrade has been completed.

Integers

In Visual Basic 6.0, the Long data type is used to represent signed 32-bit numbers, while the Integer data type is used to store 16-bit numbers. This has been changed in Visual Basic .NET. In Visual Basic .NET, the Long data type is used to store signed 64-bit numbers, the Integer to store 32-bit numbers, and the Short data type to store 16-bit numbers.

The Short data type can store numbers between –32768 to 32767. You must use the Short data type in case the possible values for a variable fall between the specified upper and lower limits. It is possible to convert a Short data type to Integer, Long, and Decimal without an overflow.

The Integer data type can store numbers between –2,147,483,648 to 2,147,483,647. In the earlier versions of Visual Basic, the variable needed to be declared as Long in case you wanted to store large numbers. Now, in order to enhance performance of your applications running on a 32-bit processor, it is advised you use the Integer data type.

The Long data type, meanwhile, stores numbers between –9,223,372,036,854,775,808 and 9,223,372,036,854,775,807, each stored as an 8-byte number. Refer to the following Visual Basic 6.0 code, where:

```
Dim Ctr as Integer
Dim Total as Long
```

is upgraded to:

```
Dim Ctr as Short
Dim Total as Integer
```

Dates

Visual Basic 6.0 and earlier versions used the Double data type to store dates. The Double data type uses four bytes to store the value. Visual Basic .NET, however, uses the Datetime data type, which is an 8-byte integer value. Since the representations are quite different in each of the versions, there is no implicit conversion between the Datetime and Double data types in Visual Basic .NET. The ToOADate and FromOADate can be used to convert between the Double and Visual Basic 6.0 representation of Date value. The upgrade tool inserts the

ToOADate or *FromOADate* method where a Double is assigned to a Date, as shown in the following code, where:

```
Dim dblVal as Double
Dim dtVal as Date
DblVal = dtVal
```

is changed to:

```
Dim dblVal as Double
Dim dtVal as Date
DblVal = dtVal.ToOADate
```

You can avoid calls to the ToOADate and FromOADate functions by using the Date data type to declare variables that store dates instead of using the Double data type. The OA in both the functions stands for OLE Automation-compatible date format.

Boolean

Boolean variables are stored as 32-bit numbers and can hold one of two values: True or False. True evaluates to 1, False to 0. The actual value translation is different that what it is in Visual Basic 6.0. In previous versions of Visual Basic, a Boolean value of True evaluated to –1 and a False evaluated to 0. Before upgrading, check your code to ensure you are not comparing Boolean variables to the hard-coded value –1 or 0, but to True and False.

Arrays

In Visual Basic 6.0, it is possible to declare arrays with any lower or upper bound numbers. The **Option Base** statement is used to determine the lower bound number if a range was not specified in the declaration. Visual Basic 6.0 also allows use of the **ReDim** statement to reassign a variant to an array.

In order to maintain interoperability with other languages, arrays defined in Visual Basic .NET have a default lower bound of zero. This has made the **Option Base** statement obsolete. In Visual Basic .NET, a **ReDim** statement cannot be used unless the variable has been declared as an array. To illustrate, the following code is valid in Visual Basic 6.0, but invalid in Visual Basic .NET:

```
Dim x
ReDim x(20) 'Cannot use ReDim since x has not been declared as an array
```

Consider the following declaration:

```
Dim x(10) as Integer
```

In Visual Basic 6.0, the preceding code declares an array of 11 integers. Since arrays in Visual Studio .NET are zero-based, the previous declaration pronounces an array of 10 integers, from 0 to 9.

During the upgrade process, all **Option Base** statements are removed. All arrays that have their lower bound as zero are left as is while those that are non-zero-based are upgraded to an array wrapper class. For example, the following Visual Basic 6.0 code

```
Dim arr(1 to 10) as double
```

is converted to:

```
Dim arr as object = new VB6.GetArray(GetType(Double), 1, 10)
```

There are a host of functions available in the wrapper class. This particular class is called *Microsoft.VisualBasic.Compatibility* and needs to be imported.

Fixed-Length Strings

In Visual Basic 6.0, variables can be declared with fixed-length strings except for public variables in class modules. Fixed-length strings are not supported in Visual Basic .NET. If the application contains fixed-length strings, then the Upgrade Wizard uses a wrapper function to implement the functionality. This is shown in the following Visual Basic 6.0 code, where:

```
Dim fxString as String * 50
```

is converted to:

```
Dim fxString as new VB6.FixedLengthString(50)
```

This lack of support also means that changes have to be made if your applications employ User-Defined Types (UDT). User-defined types are called structures in VB.NET. Structures do not support primitive types. So, if your VB6 application contains the following UDT declaration:

```
Type EmployeeRecord
        EmpID as Integer
        EmpFirstName as String * 20
        EmpMiddleName as String * 10
        EmpLastName as String * 20
EndType
```

The upgrade tool will mark the statements containing fixed-length strings, telling you to initialize each fixed-length string. However, you can modify the fixed-length strings declarations to strings in the following manner:

```
Type EmployeeRecord
        EmpID as Integer
        EmpFirstName as String
        EmpMiddleName as String
        EmpLastName as String
EndType
```

This declaration will ensure that the UDT is migrated to Visual Basic .NET without any changes. The same holds true for fixed size arrays. So, if you have the following array declaration:

```
Dim arrSample(12) as String
```

you can change it to:

```
Dim arrSample() as string
```

Windows API Data Types

A majority of the Windows API functions can be used as they are in Visual Basic .NET. The only modification you will have to make is to change the data types accordingly. The upgrade tool does the following to all your API declarations:

- Changes all occurrences of Integer to Short. (i.e., Visual Basic 6.0 Integer data type is now Short.)
- Changes all occurrences of Long to Integer. (i.e., Visual Basic 6.0 Long data type is now Integer.)
- Upgrades fixed-length string data types to a fixed-length string wrapper class.

The following Visual Basic 6.0 code displays the name of the logged in user on a Windows 2000 Server:

```
Public Declare Function GetUserName Lib "advapi32.dll" Alias
    "GetUserNameA" (ByVal lpBuffer As String, nSize As Long) As Long
```

```
Sub DisplayUserName()
    Dim strUserName As String
    strUserName = String(20, " ")
    Module1.GetUserName strUserName, 20
    MsgBox strUserName
End Sub
```

After the upgrade, the code is transformed into:

```
Public Declare Function GetUserName Lib "advapi32.dll"  Alias
    "GetUserNameA"(ByVal lpBuffer As String, ByRef nSize As Integer) As
    Integer
Sub DisplayUserName()
    Dim strUserName As String
    strUserName = New String(CChar(" "), 20)
    Module1.GetUserName(strUserName, 20)
    MsgBox(strUserName)
End Sub
```

Note the differences between the code written in Visual Basic 6.0 and that written in Visual Basic .NET:

- The nSize parameter in the Visual Basic 6.0 code is Long, whereas it is Integer in Visual Basic .NET.
- The data type of the return value has been changed from Long to Integer.
- The initialization of strings has been prompted.

Converting VB Forms to Windows Forms

Visual Basic .NET has a new Forms package called Windows Forms. While Windows Forms is largely compatible with Visual Basic 6.0 Forms, there are some minor changes that need to be done. The following differences exist between Visual Basic 6.0 Forms and Windows Forms:

- Windows Forms does not support the *Form.PrintForm* method. Therefore, you cannot print an image of the Windows Forms on a printer.

- Graphics commands like **Circle**, **Pset**, **Line**, **Point**, and **Cls** are not supported in Windows Forms. The Windows Forms package is built on a more feature-rich layer called GDI+.
- Windows Forms supports only twips as the unit of measurement in the *ScaleMode* property. If your Visual Basic 6.0 Forms used twips as the measurement, then it is upgraded correctly. There will be sizing issues if pixels were used as a unit of measurement.
- Visual Basic 6.0 supported any font type for forms and controls. But Visual Basic .NET supports only TrueType or OpenType fonts. If your existing application uses a non-TrueType font, they are changed to the default Windows Form font. All formatting is lost during this change, however. If you have formatted text, Microsoft recommends you use Arial instead of Visual Basic's default MS Sans Serif.
- In Windows Forms, the *MousePointer* property of the *Screen* object can be used only for forms inside the application.
- In Visual Basic 6.0, assigning a value of zero to the *Interval* property of the *Timer* object disables the Timer. Visual Basic .NET resets the value to one when a value of zero is assigned. To disable the timer, you should set the *Enabled* property to False. During an upgrade operation, if the tool detects a statement that assigns a value of zero to the *Interval* property, then the statement is commented with an upgrade error.
- Windows Forms does not support the *Name* property for forms and controls at runtime. Any Visual Basic 6.0 code that iterates through the Control Collection looking for a *Name* property will not be upgraded correctly.
- Windows Forms has two menu controls. They are MainMenu and ContextMenu. Visual Basic 6.0 has only one menu control called Menu that can be opened as a MainMenu or a ContextMenu. During an upgrade operation, Menu controls are upgraded to MainMenu controls. The ContextMenu controls, however, have to be explicitly re-created.
- Windows Forms does not support the OLE Container control. If your application has to use an OLE Container control, you can use the WebBrowser control as an alternative. During an upgrade process, an error is added to the upgrade report and an unsupported-control placeholder is inserted into the form.

- Image controls are not supported in Visual Basic .NET. As a result, all Image and PictureBox controls are upgraded solely to PictureBox controls.
- The *clipboard* object in Visual Basic .NET has more functionality than the Visual Basic 6.0 *clipboard* object. The new *clipboard* object supports more formats than the previous *clipboard* object. As a result, any Visual Basic 6.0 clipboard code cannot be upgraded to Visual Basic .NET. All clipboard statements will be marked with an upgrade error.
- Windows Forms has no built-in Dynamic Data Exchange (DDE) support. DDE is a form of interprocess communication that uses a concept called shared memory to exchange data between applications. Since Windows Forms does not support DDE, you cannot use the *LinkMode* property available in Visual Basic 6.0 forms. During an upgrade operation, all DDE properties and methods are commented with an upgrade warning.

Control Anchoring

If you are designing a form that the user might resize at runtime, you might want to make your controls resize and reposition correctly on the form. In order to resize controls correctly with the form, you should use the *Anchor* property. The *Anchor* property defines an anchor position for the controls on the forms. When a control is anchored on the form and the form is resized, the relative positions will be maintained after the resize. So, if a control's anchor position is set to bottom-right, irrespective of how the form is resized, horizontally or vertically, the control will always be placed in the bottom-right corner. Without the *Anchor* property, the control's position is fixed and it will not retain its original position after it is resized.

The anchor position value can be chosen from one of the AnchorStyles enumeration values. The following statement anchors a Button Control to the top-left corner of the form:

```
Button1.Anchor = AnchorStyles.TopLeft
```

Table 14.1 lists various AnchorStyles enumeration values.

Table 14.1 AnchorStyles Enumeration Values

Member Name	Description
All	Each edge of the control anchors to the corresponding edge of its container.
Bottom	The control is anchored to the bottom edge of its container.
BottomLeft	The control is anchored to the bottom and left edges of its container.
BottomLeftRight	The control is anchored to the bottom, left, and right edges of its container.
BottomRight	The control is anchored to the bottom and right edges of its container.
Left	The control is anchored to the left edge of its container.
LeftRight	The control is anchored to the left and right edges of its container.
None	The control is not anchored to any of the edges.
Right	The control is anchored to the right edge of its container.
Top	The control is anchored to the top edge of its container.
TopBottom	The control is anchored to the top and bottom edges of its container.
TopBottomLeft	The control is anchored to the top, bottom, and left edges of its container.
TopBottomRight	The control is anchored to the top, bottom, and right edges of its container.
TopLeft	The control is anchored to the top and left edges of its container.
TopLeftRight	The control is anchored to the top, left, and right edges of its container.
TopRight	The control is anchored to the top and right edges of its container.

Keyword Changes

The following keywords have either been removed from Visual Basic or replaced with a Visual Basic .NET specific version.

Goto

The **Goto** statement is present in Visual Basic .NET only for the purposes of error-handling and can be used only with **On Error...Goto**. The **Goto** statement cannot be used to perform multiple-branching. Instead, use the **Select...Case** statement to perform multiple-branching.

GoSub

The **GoSub** statement calls a procedure within a subprocedure. There is no support for the **GoSub** statement in Visual Basic .NET. Instead, method calls can be made using the **Call**, **Sub**, and **Function** statements.

Option Base

Arrays in Visual Basic .NET always have a lower bound of zero. This has rendered the Option Base obsolete in Visual Basic .NET. The **Option Base** statement was used in Visual Basic 6.0 to determine the lower bound value of those arrays that were not declared with an explicit lower bound value.

AND/OR

In Visual Basic 6.0, the AND, OR, XOR, and NOT operators perform both logical and bitwise operations depending on the expressions. In Visual Basic .NET, AND, OR, XOR, and NOT operators apply only to type *Boolean*. The AND and OR operator short-circuits evaluation if the value of the first operand is enough to determine the result of the operation.

Visual Basic .NET uses the new bitwise operators. They are BitOr, BitAnd, and BitXor. The new bitwise operators do not short-circuit.

The Upgrade Wizard upgrades an **AND/OR** statement which is non-Boolean or contains functions, methods, or properties that use a compatibility function with the same behavior as that in Visual Basic 6.0. For Boolean statements, it is upgraded to use the native Visual Basic .NET statement.

Lset

The **Lset** statement can be used in two ways:

- It can assign a variable of one user-defined data type to another variable of a different user-defined data type. This is not supported in Visual Basic .NET.

- It can pad a string with spaces to make it a specified length. The *PadRight* method of the *string* class can be used to achieve this functionality.

VarPtr

VarPtr is used to acquire the address of a variable or array element. It takes the variable name or an array element as the parameter and returns the address. Since Visual Basic .NET does not support this function, the upgrade tool does not upgrade this statement and therefore marks it with an upgrade error.

StrPtr

Strings in Visual Basic are stored as BSTRs. If you pass a string variable to the VarPtr function, you will get the address of the BSTR which acts as a pointer to the string. To get the address of the string buffer, you need to use the StrPtr function. This function returns the address of the first character in the string. Because Visual Basic.NET doesn't support StrPtr, the upgrade tool reports an upgrade error when it encounters this statement.

Def

The purpose of the **DefBool**, **DefByte**, **DefInt**, **DefLng**, **DefCur**, **DefSng**, **DefDbl**, **DefDec**, **DefDate**, **DefStr**, **DefObj** and **DefVar** statements is to set the default data type for those variables, parameters, and procedure variables whose names start with the specified character. To improve the readability and robustness of the code, Visual Basic .NET does not support these statements. Sample Visual Basic 6.0 code is illustrated in the following:

```
DefStr x-z

Sub Test

     x = "hello world"

End Sub
```

is upgraded to:

```
Sub Test

     Dim x as String

     x = "hello world"

End Sub
```

Programming Differences

The programming differences between the previous versions of Visual Basic and Visual Basic .NET are many. They include changes in the way methods and properties are implemented, and the fact that a host of old features, like GoSub, are not supported. This section discusses the following differences:

- Method implementation
- Dealing with unmanaged code
- Properties
- Property name changes for some controls
- Default properties
- Null usage

Method Implementation

Procedures and functions in Visual Basic .NET have undergone some changes, ranging from treatment of optional arguments to the latest feature of function overloading. Modifications regarding how methods are implemented can be discussed under the following headings:

- Optional Parameters
- Static Modifier
- Return Statement
- Procedure Calls
- External Procedure Declaration
- Passing Parameters
- ParamArray
- Overloading

Optional Parameters

In Visual Basic 6.0, you can declare a procedure parameter as optional without specifying a default value. If the optional parameter was of type Variant, the IsMissing function can be used to determine whether the optional parameter is present. This is something Visual Basic .NET can not do, since the IsMissing function is not supported there. But, In Visual Basic .NET, an optional procedure parameter must be declared with a default value. This value is then passed to the procedure if the calling program does not supply the optional parameter. The following declaration shows you how to code a procedure that accepts an optional parameter:

```
Function CheckBalance(ByVal strACNUM as string, Optional ByVal
    strACName as String = "")
```

You can easily check if the optional parameter was passed to a function or procedure with the help of the default value. In the procedure, check to see if the optional parameter contains the default value. If it has the same value, then the optional parameter has not been passed. If the optional parameter contains a different value than that of the default, then the parameter was passed. To successfully use this method, you must make sure the default value you assign is unique.

Static Modifier

You can declare a procedure in Visual Basic 6.0 with the Static modifier. All variables inside the procedure are then treated as static variables. Static variables retain their value even after a function has finished execution. But in Visual Basic .NET, the Static modifier cannot be used when declaring procedures or functions. If you want to declare a variable as a static variable, you need to declare it explicitly.

Return Statement

The functionality of the **Return** statement has changed in Visual Basic .NET. In Visual Basic 6.0, the **Return** statement can be used only to branch to a particular statement in the calling code. This is typically the next statement following the **GoSub** statement. In Visual Basic .NET, you use the **Return** statement to give back control to the calling program, which is done by coding the **Return** statement in a Function or Sub procedure. (It is important to note that the **GoSub** statement is not supported in Visual Basic .NET.)

The following Visual Basic 6.0 code illustrates how the **Return** statement is used to return control to the line following the **GoSub** statement. First, the **GoSub** statement transfers control to the Add subroutine. The Add subroutine then adds the two integers and transfers control to the line following the **GoSub** statement with the help of the **Return** statement. After the control is transferred, the result is displayed and the **Exit Sub** statement terminates the operation.

(The **Exit Sub** statement is required to prevent the control from executing the add subroutine.)

```
Function AddInt() as Integer
    Dim x As Integer
    Dim y As Integer
    Dim result As Integer

    x = 10
    y = 20
    MsgBox result

    Return
End Sub
```

The following subroutine demonstrates implementation of the **Return** statement in giving back control to the calling program. First, the subroutine checks the value of the denominator. If the value is zero, then the control is returned to the calling program after displaying an error message. This is done with the help of the **Return** statement. If the denominator has a value greater than zero, the resulting division is displayed:

```
Sub Divide(Byval x as Short, ByVal y as short)
        If y = 0 then
             Msgbox "Cannot Divide by zero"
             Return 0
        Else

             Return x / y
        End if

End Sub
```

Procedure Calls

In Visual Basic 6.0, all function calls require parentheses around the parameter list. If you are invoking a Sub procedure, the parentheses are required if you use the **Call** statement. You cannot use parentheses when you invoke a Sub procedure without a **Call** statement. The following code fragment shows you the variations when a subroutine is called in Visual Basic 6.0:

```
Result = Add(10, 20)
Call FindDetails(aulname, aufname)
FindDetails aulname, aufname
```

In Visual Basic .NET, parentheses are required for any invocation of a Function or Sub procedure that contains parameters. The usage of a **Call** statement is optional. If a Sub or Function procedure does not contain any parameters, you can either choose to use parentheses, or leave them out altogether. The following code fragments illustrate how function calls are made:

```
Result = Add(10, 20)
FindDetails(aulname, aufname)
```

External Procedure Declaration

In Visual Basic 6.0, you can reference an external procedure using a **Declare** statement. External procedures are typically API calls. In some cases, when a data type for a parameter is unknown, or a return value is unknown, you can use the **Any** keyword. The **Any** keyword allows you to pass any data type to parameters that have been declared of this type. This does not, however, involve type safety. Visual Basic .NET, on the other hand, does not support the **Any** keyword. So, if an external procedure contains a parameter that has been declared as *Any*, the Upgrade Wizard converts it to the *Object* data type. This was done to increase type safety and ensure consistency and interoperability between applications. You will have to declare parameters to be of a specific type before you can use them. Declaring external procedures in Visual Basic .NET is done in much the same way as Visual Basic 6.0. The following is a Visual Basic 6.0 external procedure declaration:

```
Public Declare Function GetComputerNameW Lib "kernel32" (lpBuffer As
    Any, nSize As Long) As Long
```

In Visual Basic .NET, it will be changed to:

```
Public Declare Function GetComputerNameW Lib "kernel32" (ByRef lpBuffer
    As Object, ByRef nSize As Integer) As Integer
```

Passing Parameters

The default parameter passing mechanism in Visual Basic 6.0 is ByRef. This means that any change made to the parameter in the called program is reflected in the calling program. Of course, such passing mechanisms have their pros and cons. In Visual Basic .NET, on the other hand, the default parameter passing mechanism is ByVal. When parameters are passed ByVal, any changes made to the parameter values are effective only in the called function. The original values present in the calling function are not affected.

When an argument is passed as ByVal, a copy of the variables is passed to the called function. The advantage is that original values are retained. Arguments passed using ByRef, however, run the risk of being modified the called function or subroutine.

ParamArray

In Visual Basic 6.0, you can designate the ParamArray parameter to be the last parameter to accept an array of parameters. This can be helpful when you don't know the number of parameters being passed to a procedure. In addition, you cannot explicitly specify the passing mechanism as ByVal or ByRef for parameters declared to be ParamArray. They are always passed as ByRef. This, unfortunately, cannot be changed. Likewise, the data type for ParamArray parameters can only be Variant. In Visual Basic .NET, on the other hand, the ParamArray parameters are always passed as ByVal, and need to be declared as the *Object* data type.

The following Visual Basic 6.0 code uses the subroutine called AddToArray. This subroutine receives the array, as well as the elements added to the array, and appends the list of values to it. Since the number of individual array elements cannot be determined in advance, the ParamArray parameter is used to pass multiple parameters:

```
Private Sub Command1_Click()
    Dim x() As Integer
    Dim ctr As Integer

   Call AddToArray(x, 1, 2, 3, 4)

   For ctr = 0 To UBound(x)
    Debug.Print x(ctr)
   Next
```

```
End Sub
Sub AddToArray(x() As Integer, ParamArray Values() As Variant)
    Dim ctr As Integer

    For ctr = 0 To UBound(Values)
        ReDim Preserve x(ctr)
        x(ctr) = Values(ctr)
    Next
End Sub
```

The following code illustrates the same program written in Visual Basic .NET:

```
Protected Sub Button1_Click(ByVal sender As Object, ByVal e As

     System.EventArgs)
        Dim x() As Integer
        Dim ctr As Integer

        Call AddToArray(x, 1, 2, 3, 4)

        For ctr = 0 To UBound(x)
            system.Diagnostics.Debug.WriteLine(x(ctr))
        Next

    End Sub

    Sub AddToArray(ByRef x() As Integer, ParamArray ByVal Values() As
        Object)
        Dim ctr As Integer
        ctr = 0
        ReDim x((ubound(Values)) + 1)
        For ctr = 0 To UBound(Values)
            x(ctr) = values(ctr)
        Next

    End Sub
```

Note the change to the ParamArray parameter data type from Variant to Object, as well as the change to the **ReDim** statement. A value of one is added to the Ubound function when the array is re-dimensioned. This is necessary because arrays in Visual Basic .NET are zero-based.

The sample program demonstrates the use of ParamArray. The AddToArray function accepts an array as the first argument, as well as a series of values to be inserted into the array. The second parameter is declared to be a ParamArray since the number of values vary depending on the situation, while the subroutine re-dimensions the array to accommodate the actual number of elements that can be stored there. When re-dimensioning the array, it is necessary to add one to the size since arrays in VB.NET are zero-based.

Overloading

Overloading is a concept that has been in vogue for a number of years now. The C++ language introduced this feature as a means to achieve the same functionality, differing only in terms of the number or types of parameters. A new paradigm called function signature is closely related to function overloading. A function signature is nothing but a template used by a compiler to check when a call is made to the function or procedure. The signature consists of the following:

- Name of the procedure
- Number of parameters
- Order of parameters
- Data types of parameters

The following are *not* a part of a function signature:

- Procedure modifiers like Private, Public, and so on
- Parameter names
- Parameter modifiers like ByRef and ByVal
- Return values and types

Overloading a function involves changing at least one of the elements that form part of the signature. Overload procedures have the following features:

- Overloaded functions must differ only in their signatures, not on anything else.

- Functions differentiated by virtue of their signatures can have any procedure modifier or return type.
- It is possible to overload a Function procedure with a Sub procedure or vice-versa, provided their parameters are different. It is also possible to overload a property or method in a class.

Earlier versions of Visual Basic do not support the concept of overloading. As a result, you cannot have two or more procedures or functions with the same name that differ only in their arguments. Each procedure, even if they offer the same functionality, must be given a unique name. Thus, if you were writing a procedure to divide two values, and you want to offer variations in the arguments passed to these procedures, you have to write the procedures with the idea that each procedure should be different from the other, not only in name but in the arguments passed to it. The following code fragment illustrates this:

```
Function DivideInteger(i1 As Integer, i2 As Integer) As Integer
    DivideInteger = i1 / i2
End Function
Function DivideLong(l1 As Long, l2 As Long) As Long
    DivideLong = l1 / l2
End Function
```

The two functions, DivideInteger and DivideLong, offer variations on dividing two integers or two longs. This puts added pressure on the programmer to remember the correct name of the procedure when making a call to one or both. Visual Basic .NET, however, overcomes this problem with the help of function overloading. The same code fragment rewritten in Visual Basic .NET is shown next:

```
Overloads Function Divide(i1 As Integer, i2 As Integer) As Integer
    If i2 = 0 Then
        Return 0
    Else

        Return i1 / i2
    End If
End Function
Overloads Function Divide(l1 As Long, l2 As Long) As Long
```

```
        If i2 = 0 Then
        Return 0
    Else

        Return i1 / i2
    End If

End Function
```

Now, both functions share the same name, differing only in the data type passed as arguments. The runtime chooses the appropriate method depending on the parameters used when the call is made.

Since overloading is not supported in Visual Basic 6.0, the functions written in Visual Basic 6.0 undergo the same modifications that relate to data type changes, keyword changes, and so on, used in the procedure. The following points summarize changes affected by Visual Basic .NET:

- Optional parameters must be declared with a default value.
- Visual Basic .NET does not support the IsMissing function.
- Static modifiers cannot be used in Visual Basic .NET when declaring procedures or functions.
- The **Return** statement is now used to transfer control to the calling program.
- Visual Basic .NET requires parentheses to invoke any procedure or function containing parameters.
- Use of the **Any** keyword is not supported in Visual Basic .NET. The **Any** keyword is used in external procedure declarations to indicate a particular parameter can contain data of any type.
- The default parameter passing mechanism in Visual Basic .NET is ByVal.
- ParamArray parameters are always passed as ByVal and the data type for the parameters must be of type *Object*.
- Implementation of overloading—the concept of using the same procedure name with varying function signatures.

References to Unmanaged Libraries

The introduction of the .NET Framework has certainly given programming a new dimension. Likely the biggest worry you have, concerns all those COM components you created over the years. The good news is that all of them can still be used with applications from the .NET Framework. The .NET Framework provides you with various techniques to help leverage the functionality of existing components.

The following example illustrates a very simple implementation of a .NET component, and the unmanaged client accessing that component. The code for the .NET component is shown next:

```
Namespace SimpleComponent
Public Class Simple1

Public Sub New()

End Sub

Public Function SimpleMethod() As String
    Return "From SimpleComponent in .NET"
End Function

End Class
End Namespace
```

The following is the code for the unmanaged client:

```
Sub AccessNETComponent()
    Dim obj as Object
    Set obj = CreateObject("SimpleComponent.Simple1")
    Msgbox obj.SimpleMethod
End Sub
```

Though the method of coding has not changed much, the background work necessary to allow the unmanaged code to access the .NET component is quite substantial. The following steps detail the process:

1. Once the .NET component is coded, it must be compiled. This is done using the Visual Basic .NET compiler, which can be invoked from the command-line by typing: **vbc.exe**. The .NET component then has to be compiled into a DLL, the syntax for which is:

   ```
   Vbc.exe /target:library <sourcefile>
   ```

   The /target:library option instructs the compiler to produce a DLL as the output. In this example, the code is contained in SimpleComponent.vb, making the command:

   ```
   Vbc.exe /target:library SimpleComponent.vb
   ```

2. Once the .NET component is compiled, the next step is to register with COM. Dynamic Link Libraries compiled in the .NET environment are different from COM DLLs. Since the old regsvr32.exe cannot be used to register the DLLs, the Register Assembly tool (regasm.exe) should be employed instead. Available as part of the .NET SDK, this tool reads the component's metadata and makes appropriate entries in the registry. These entries include the programmatic identifier (ProgID) and the class ID (CLSID) for the co-classes, which register the appropriate subkeys. You can also use this tool to register the type library when you register the component. So, in this example, you can issue the following command to register the .NET component:

   ```
   Regasm.exe /tlb:simplecomponent.tlb simplecomponent.dll
   ```

3. Once the component is registered, the unmanaged client application can be copied to the same folder as the .NET component, and then run.

The Common Language Runtime (CLR), which is the core of the .NET Framework, manages all code that runs inside the .NET Framework. This code is called the managed code, whereas any code that runs outside the confines of the .NET Framework is called unmanaged code. All COM, COM+ components, ActiveX controls, and Win32 API functions fall under this category. The managed and unmanaged object models vary regarding data types, error handling, and so on. The main function of the CLR is to simplify the interoperation between these components. This is achieved in various ways. This section introduces some of the concepts that aid in this transition:

- Metadata
- Runtime Callable Wrapper
- COM Callable Wrapper

Debugging…

Attaching a Debugger

Visual Studio .NET introduces a feature called Debugger that allows you to attach to a process running outside the context of Visual Studio. This feature of attaching to a process allows you to:

- Debug a program that runs in a different process on the same machine, or on a different machine. Debugging a process in a different machine is called remote debugging. It is important to note that remote debugging is not supported in this beta version.
- Debug multiple programs at the same time.
- Invoke the debugger automatically whenever the debugged process crashes. This is also referred to as Just-In-Time (JIT) debugging.

Attaching a debugger to a process is very helpful when attempting to debug an application that interoperates with several other components. You can also debug unmanaged code from within managed code. This can be done by selecting Processes from the Debug menu. The list of currently running processes is listed in the available processes pane. The next step is to select the process you want to debug and click the **Attach** button. This brings up a dialog box displaying the list of available program types. Select the program type related to the application you wish to debug. Click the **OK** button to close this window, followed by the **Close** button on the Processes window.

Metadata

The word Metadata means data about data. Metadata can be defined as a collection of binary code that describes a .NET component. It is either stored in memory or in a .NET Framework portable executable file. It helps interoperate with pre-existing COM components, so if your .NET component wants to work with a COM component, the COM component needs to provide information about itself so the .NET component can identify the methods and properties contained in the *COM* object. The runtime uses the metadata definition to bind the component during compile time and generate the relevant wrapper. The metadata about the COM component is available just like any other managed namespace the CLR provides.

Various tools can be used to generate metadata of a COM component. They are:

- Type Library Importer
- TypeLibConverter Class

The Type Library Importer, tlbimp.exe, is a command-line utility that converts classes and interfaces contained in the COM type library to .NET metadata. The metadata is then used by the .NET clients to instantiate a *COM* object. Unfortunately, the Type Library Importer utility converts the entire COM type library, not just a portion of it. It also cannot convert an in-memory type library to metadata. The syntax of the tlbimp.exe tool is:

```
tlbimp.exe <TypeLibrary file> [/out: outputfilename]
```

For example, the following command will allow you to convert the ADO type library to .NET metadata:

```
Tlbimp.exe msado25.tlb /out:adonet.dll
```

You can use the Intermediate Language Disassembler (ILDASM) tool to view the contents of the file generated by the tlbimp.exe tool.

The TypeLibConverter class, meanwhile, is part of the *System.Runtime .InteropServices* namespace and can convert the classes and interfaces contained in the COM type library to .NET metadata. The class contains two methods that aid in this conversion. They are:

- ConvertAssemblyToTypeLib
- ConvertTypeLibToAssembly

Both methods output the same metadata.

Runtime Callable Wrapper

The Microsoft Component Object Model (COM) differs from the .NET Framework in a number of ways. There is also another CLR component, apart from using metadata, that helps .NET clients talk to COM components. It's called the Runtime Callable Wrapper (RCW). The RCW is a proxy created by the CLR when a .NET client generates an instance of the *COM* object. From the client's point of view, the RCW is seen as an instance of the managed object. The primary function of a RCW is to marshal calls between a .NET client and a *COM* object.

The CLR creates one RCW for each *COM* object. Even though a client may hold multiple references to the *COM* object, only one RCW will be created. The runtime is responsible for creating both the *COM* object and that for the Runtime Callable Wrapper. The *Runtime Callable Wrapper* object contains a repository, which holds the interface pointers to the *COM* object. It releases the reference to the *COM* object when the number of references is zero.

The RCW marshals data between managed and unmanaged code. It also reconciles the data representation differences that exist between the client and the *COM* object in terms of arguments passed to methods and return values. *Marshalling* is a mechanism that refers to the method by which a client in one process makes a call to functions in another process running on the same machine or on a remote machine. This is achieved in two steps. First, the client must be aware of the existence of the server process. This is done by taking a reference to the interface and passing it over to the client process. The second step is to pass the parameters from the client to the server. The underlying architecture creates a server proxy in the client process and a stub in the server process. The client sees the proxy as the server and makes method calls as if it is the actual server. Once a method call is made, the proxy accepts the call and passes on the stub located in the server process. The actual transportation is handled by some form of remote process communication like shared memory, named pipes, or others. After the stub receives the request, it parses the call and passes it onto the server. Marshalling is needed whenever the client and server are loaded in different processes either on the same or a different machine.

Garbage Collection is another issue to be contended with if you are working with objects. The .NET Framework does an excellent job of Garbage Collection and programmers can now concentrate more on building applications rather that worry about releasing objects or leaking memory. Visual Basic .NET controls the way objects are created and destroyed. The **New** keyword is used to create an object in Visual Basic .NET. When you set an object to Nothing, the object is destroyed and the memory referenced by the object is freed. But there is more to this than meets the eye. The *Sub New* procedure is a constructor that is called whenever you create an object, and can contain code that does common initialization tasks. It replaces the *Class_Initialize* method in Visual Basic 6.0. The *Sub New* constructor cannot be explicitly invoked from anywhere in the program except from another overloaded constructor in the same class or in a derived class. Just as a constructor is called when an object is created, the *Sub Finalize* method performs the role of a *de*structor, effectively replacing the *Class_Terminate* method and performing all cleanup activities.

The .NET Framework automatically calls the destructor when it determines that objects are not being used anymore. But it is important to note that the call to the destructor is not immediate. The .NET Framework does not invoke the destructor as soon as the object goes out of scope or is destroyed explicitly by setting it to Nothing. The framework instead calls the destructor sometime after the object has been destroyed. The main advantage with Garbage Collection in Visual Basic .NET is that it is automatic. Objects are released and memory is freed without any additional changes from the application. The disadvantage is that some objects might stay in memory longer than needed, causing the unnecessary locking of memory locations. Another disadvantage is that an application cannot directly make a call to the destructor.

You can also implement an additional destructor called Dispose if you want to take control of management of resources. The *Dispose* method can contain code to implement all cleanup activities just like the *Finalize* method. The *Dispose* method is not automatically invoked, so your application must summon it to perform finalization tasks.

COM Callable Wrapper

The COM Callable Wrapper does the same thing as the Runtime Callable Wrapper but from the COM client's point of view. When a COM client creates an instance of the managed class, the runtime creates a COM Callable Wrapper for the managed object. The runtime creates only one wrapper for the managed object irrespective of the number of COM clients requesting the reference to the managed object. The primary responsibility of a COM Callable Wrapper is to marshal calls between the managed object and the COM client. The following points summarize how references to unmanaged libraries are handled in Visual Basic .NET:

- Metadata is binary data that describes a .NET component.
- The Runtime Callable Wrapper (RCW) is a proxy that helps .NET clients talk to COM components.
- The COM Callable Wrapper (CCW) is a proxy that helps COM clients talk to .NET components.

Debugging…

Tracing Code

The *Debug* class present in the *System.Diagnostics* namespace provides you with various methods that allow you to trace code. Code tracing is important during development because it aids you in identifying a problem or in analyzing performance. The *Write* and the *WriteLine* methods allow you to print messages in the Output window. This way, you can place temporary messages to track the application flow. This is a very important factor to consider if you are building a client and server application and want to track the code-paths in both applications.

The .NET Framework also contains the *Trace class* which helps you trace the flow of the application. To embed tracing in your application, you should compile your application with a set of trace switches. These switches also allow you to specify where the trace information should be displayed and to what extent tracing should be done.

Since the *Trace* and *Debug* classes allow you to monitor an application's performance, as well as provide information about application flow, you may want to include code, when developing an application, that use the methods of the *Trace* and *Debug* class. The *Debug* class is normally used to display diagnostic or non-tracing information about your application. After the application has been developed and is ready to be deployed, you can compile the application by turning off Debug switches and turning on Trace switches.

To enable or disable Trace or Debug switches, open Solution explorer, right-click **Solution**, and choose **Properties**. In the **Property Page** dialog box, choose **Configuration Properties** from the left pane and select **Build**. In the right pane, select the **Define Debug Constant** and/or the **Define Trace Constant** checkboxes under conditional compilation constants, depending on whether you want debug and/or trace.

Properties

Property Procedures are implemented differently in Visual Basic .NET. With the **Set** statement no longer supported, both variable assignments and object assignments are treated the same. A property procedure consists of a set of Visual Basic statements that allow you to work with properties that are user-defined. These properties are defined in a class or a module. Visual Basic .NET provides two types of property procedures to work with properties. They are:

- *Get*: The Get procedure is used to return the property's value.
- *Set*: The Set procedure is used to assign a value to the property.

Working with Property Procedures

In Visual Basic 6.0, a property procedure is declared in the following manner:

```
Property Let CustName(strCustName as string)
        m_CustName = strCustName
End Property
Property Get CustName() as String
        CustName = m_CustName
End Property
```

In VB.NET, however, they are declared differently. Property procedure statements are contained within the **Property** and **End Property** statements. The Get and Set procedures are coded within this block. A property can be declared as a default property by prefixing the property procedure with the **Default** keyword, or you can define the scope of the property procedure using the **Public**, **Protected**, **Friend**, or **Private** keywords. Properties are public by default, unless otherwise specified. The following code shows you the implementation of a property procedure:

```
Public Property CustName() as String
Get
        Return m_CustName
End Get
Set
       m_CustName = Value
End Set
End Property
```

If you look closely at the procedure declaration, you will see that a variable with the name *Value* is being used. Visual Basic .NET uses this variable name as the default variable if you did not declare the Set procedure as receiving any arguments.

In a Get procedure, the return value is the value of the property returned to the calling expression. In a Set procedure, the new property value is passed in as the argument of the **Set** statement. If an argument is declared, then it must be of the same data type as the property. If an argument is not specified, then the implicit argument named *Value* is used to represent the new value.

The following code fragment shows you how to implement a property procedure with arguments:

```
Public Class Class1
    Private intSamp As Integer

    Property Sample(ByVal x As Integer)
        Get
            Sample = intSamp
        End Get
        Set
            intSamp = Value
        End Set
    End Property
End Class
```

The method of declaring arguments for property procedures is the same as declaring arguments for a Function or a Sub procedure. The only difference in the declarations is that all parameters are passed as ByVal. You can also declare optional arguments to property procedures. Arguments declared as optional must have a default value assigned to them. The new syntax is vastly different from the earlier versions of Visual Basic.

Control Property Name Changes

Visual Basic .NET has replaced many property names with new names. Besides this, all data binding properties have been implemented differently in VB.NET. The following section provides a summary of changes effected for property names.

Label Control

The Label control has undergone the following changes in Visual Basic .NET:

- The *Align* property has been changed to *TextAlign*.
- The *Appearance* property has no equivalent and has been combined with the *BorderStyle* property.
- The *Caption* property has been replaced with the new *Text* property.
- A new property called *Modifiers* has been introduced to fix the scope of the control. The possible values are Private, Public, and Protected.
- All data binding and OLE properties have been removed.

Button Control

The Visual Basic 6.0 CommandButton control has been renamed Button Control. Besides the change in name, the CommandButton control has also undergone the following changes:

- The *Caption* property has been changed to *Text* property.
- The Button Control can now have a ContextMenu associated with it through the *ContextMenu* property.
- A new property called the *DialogResult* property has been introduced. This property has the following valid values: Abort, Cancel, Ignore, No, None, OK, Retry, and Yes. If the value of this property is set to anything other than None, and if the parent form was displayed through the *ShowDialog* method, clicking the button closes the parent form without having to code for any events. The form's *DialogResult* property is then set to the same value as the *DialogResult* property of the *Button* object.
- The *Default* and *Cancel* properties have been removed.

Textbox Control

The Textbox control has undergone the following changes in Visual Basic .NET:

- Two new properties have been introduced to facilitate formatting the contents of the Textbox control. The properties are *AcceptsTab* and *AcceptsReturn*.

- A new property called *CharacterCase* has been introduced to set the case of text entered in the Textbox control.
- A new property called *Lines* has been introduced, allowing a user to enter multiple lines during design time.

In general, all controls have undergone changes with respect to the following properties:

- The *Index* property is no longer supported in any of the controls.
- The *MousePointer* property has been changed to *Cursor* property.
- A new property called *Modifiers* has been added. This can be used in selecting the access specifier for the control.
- The *Caption* property has been changed to *Text* property.
- A new property called *ImageAlign* has been added. This property can be used to set the alignment of the control in the form.
- A new property called *Dock* can be used to dock the controls to a specific location.

During an upgrade, the older properties supported will be automatically mapped to newer properties. This includes properties that have been retained as is or properties that have had their names changed. If your control used properties that are unsupported in Visual Basic .NET, then they are marked as UPGRADE_ISSUE with an appropriate description of the issue.

Default Property

A default property is a property that can be accessed by referencing the object directly. In reality, it is more of a programming shortcut. For example, the *Label* object has the *Caption* property as its default property. So, if you had a label named label1, instead of writing the following line of code to set the caption on the label:

```
label1.Caption = "Enter Name"
```

you can write:

```
label1 = "Enter Name"
```

The default property is resolved when the code is compiled. It is also possible to use late-bound objects with default properties. When using late-bound objects, the property is resolved at runtime, as shown in the following:

```
Dim objLbl as Object
Set objLbl = Form1.label1
ObjLbl = "Enter Name"
```

There are a lot of disadvantages in using default properties as implemented in Visual Basic 6.0:

- Default properties assume that the programmer knows what default property is associated with each object. This leads to uncertainty when debugging programs. In the preceding code fragment, it is difficult to determine whether the string value "Enter Name" is assigned to a variable called *label1* or whether the string value is assigned to the default property of the object called *label1*.
- It is not easy to determine if an object has a default property and if so, what property that should be.
- Default properties necessitate the usage of the **Set** statement. This is because we need to differentiate between working with an object and working with a default property of the object. With the **Set** statement becoming obsolete in Visual Basic .NET, the need for using parameterless default properties is also done away with. The following example illustrates the need for using the **Set** statement when assigning an object reference:

```
Dim Text1 as Textbox
Dim Text2 as Textbox
Text1 = "Some Text"    'Assigning a value to the text property
Set Text2 = Text1      'Assigning the Text1's object reference
                       'to Text2
Text2 = Text1          'Assigning the text property of Text1 to
                       'text property of Text2
```

Visual Basic .NET does not implement the concept of parameterless default properties. So, during an upgrade process, the Upgrade Wizard resolves the default properties to the appropriate property. But if you are using late-bound objects, then the Upgrade Wizard does not have much information about the type of object this object will be bound to. The preceding example can be rewritten in Visual Basic .NET as:

```
Dim Text1 as Textbox
Dim Text2 as Textbox
```

```
Text1.text = "Some Text"        'Assigning a value to the text property
Text2 = Text1                   'Assigning the Text1's object reference
                                'to Text2
Text2.text = Text1.text         'Assigning the text property of Text1 to
                                'text property of Text2
```

However, Visual Basic .NET does support default properties with parameters. The nomenclature of the two terms can be a little confusing. The following code aims to clear this up, however. The *System.Collections* namespace implements a *Dictionary* class that stores various key-value pairs. Consider the following code which adds two words to the dictionary collection and displays them:

```
Dim objCol As New System.Collections.Dictionary()
objcol.Add(1, "Amrita")
objcol.Add(2, "Aarthi")

Msgbox(objcol.Item(1).ToString) 'Explicitly referencing the
                                'Item property
Msgbox(objcol(2).ToString)       'Using the default property
                                 'with parameter
```

The *Item* property is the default property for the *Collection* object. Since this property can be accessed by specifying an index as a parameter, you can reference this as a default parameter so the statement becomes a valid reference to the default parameter (see the code fragment that follows):

```
Msgbox(objcol(2).ToString)
```

The following points summarize the changes that have been made to the process of working with properties in Visual Basic .NET:

- The syntax of property procedures has been changed.
- There is no longer a Let procedure.
- Property names for commonly-used controls have undergone changes in name. Also, some properties have been removed.
- A property can be considered a default property only if it can be parameterized.

Null Usage

The **Null** keyword and **Empty** keywords are not supported in Visual Basic .NET. The **Null** keyword was used in previous versions of Visual Basic to indicate that a variable does not contain any valid data. The **Empty** keyword, when assigned to a variable, indicates that the variable is uninitialized. Visual Basic .NET introduces the **Nothing** keyword, effectively replacing the **Null** and **Empty** keywords. Alternatively, you can use the *DBNull* class in the *System* namespace to represent a Null value. Note that the **Null** keyword is no longer in use.

The word Null is a reserved keyword in Visual Basic .NET, but does not have any implementation so far. With the **Empty** keyword phased out, the IsEmpty function is no longer supported in Visual Basic .NET. The IsNull function has been replaced by the IsDBNull function. The following code illustrates this. A variable of type *Object* is assigned a DBNull value and the IsDBNull function is used to check for the same:

```
Dim objSample As Object
objSample = System.DBNull.Value
If IsDBNull(objSample) Then
    Msgbox("Sample is null")
Else
    Msgbox("Sample is not null")
End If
```

It is important to note that some expressions which you might expect to evaluate to True under certain circumstances (such as If Var = DbNull and If Var <> DbNull) are always False. This is because any expression containing a DbNull is itself DbNull and, therefore, False.

Understanding Error Handling

Applications written using Visual Basic 6.0 used the **On Error** statement to handle errors. The *Err* object provided diagnostic information about the error. The *Number* and *Description* properties provided the error code and a description of the error. The main drawback of this kind of error handling is the inability to trap errors raised by Windows DLLs. System errors that arise during calls to Windows DLLs do not raise exceptions and cannot be trapped by this style of error handling.

Visual Basic .NET uses structured exception handling to handle errors and exceptions. Most of the object-oriented (OO) languages use this mechanism to handle errors. Structured exception handling enables you to create more robust and comprehensive error handlers. The Common Language Runtime (CLR) uses structured exception handling based on exception objects and protected blocks of code. When an exception or an error occurs, an object is created to represent the exception. The exception objects created are objects of exception classes derived from System.Exception. It is also possible to create custom exception classes. All languages that use the CLR handle exceptions in a similar manner. Structured exception handling consists of using the Try…Catch…Finally syntax. The *Try* block normally contains both the corresponding *Catch* and *Finally* blocks. The code that throws the exception is surrounded in a *Try* block. The *Catch* block consists of a series of statements beginning with the keyword **Catch**, followed by an optional filter that specifies the exception type. It is also possible to code multiple *Catch* blocks for a *Try* block. A *Catch* block that contains a filter for a specific exception type is invoked when that exception is thrown. The *Catch* block that contains no parameters, also called a general exception handler, is invoked for all other exceptions. The *Finally* block follows the *Catch* block, and contains the cleanup code.

It is important to note the order of the **Catch** statements if you are coding a general exception handler as well as specific exception handlers. The general handler should be the last if you are coding *Catch* blocks to handle specific exceptions. If the general handler is coded first followed by other specific handlers, the runtime invokes the general handler by default since the general handler handles all exceptions. The rule of thumb is to go from specific exception handlers to general exception handlers. See the code that follows:

```
Try
      <Statements to be executed>
     'Indicates the beginning of the exception handler
     'Contains code that might throw exceptions
Catch [<variable> As <ExceptionType>]
       <Exception processing statements>
      'This will be executed if the statements in the Try
      'block fails and the exception thrown matches the
      'exception specified as a parameter
[Additional Catch Blocks]
```

```
Finally
       <Cleanup Code>
      'Contains cleanup code
End Try
```

Consider the following code fragment that demonstrates how to implement a Try…Catch…Finally in a Visual Basic program.

```
Try
   result = intVar1 / intVar2
Catch ex as System.OverflowException
     Msgbox (ex.Message)
Finally
End Try
```

The statement that divides two integers is included in the *Try* block because there might be a situation when the denominator could be zero. The *Catch* block is defined with an object of the specific exception type. This is the exception we expect the code to throw. Once the exception is caught, you can display a custom error message to the user. In this case, the message displayed is the default message for this exception as defined in the CLR. Normally, the *Finally* block contains cleanup code. In this case, however, the *Finally* block is left empty.

It is also possible to extend exception handling by implementing custom interfaces. Custom exception handlers are implemented by creating a new exception that inherits from a *System.Exception* class or a class derived from the *System.Exception* class. Then you need to determine the situations under which this exception will be thrown, and finally, write appropriate code to throw that exception. The following exercise walks you through these steps.

Exercise 14.1: Using Error Handling

1. Add a class to your project. The first step is to inherit from the *System.Exception* class or a class derived from the *System.Exception* class. The following code shows you the implementation:

   ```
   Public Class CreditDebitException
           Inherits System.Exception
           'Call the base class constructor from the constructor
           'of this class.
   ```

```
        Public Sub New(ByVal strMessage as String)
              MyBase.New(strMessage)
        End Sub
End Class
```

2. The second step is to decide how and when to use this exception. This can be done by examining the code and determining where this exception could fit in.
3. After finalizing the usage, you can use the exception in the application. Use the **Throw** statement to explicitly raise an exception. This is equivalent to calling the *Raise* method of the *Err* object in Visual Basic 6.0. In this example, a middle-tier component throws the CreditDebitException when the Credit and Debit amounts are not equal. Once the exception is thrown by the component, the client receives the exception. The client would typically code the *CheckDebitCredit* method in the *Try* block and the *Catch* block can be coded to receive the *CreditDebitException*, as shown next:

```
Public Sub CheckDebitCredit(ByVal intDebitVal as Integer, ByVal _
    intCreditVal as Integer)
   If (intDebitVal != intCreditVal) Then
        Throw New CreditDebitException("Credit and Debit must
            be equal")
   End If
End Sub
```

Data Access Changes in Visual Basic .NET

ADO.NET is a vast improvement over its predecessor ADO. ADO.NET offers a variety of features that include disconnected data access, performance optimization, and better and richer data type support. There are a lot of differences between ADO and ADO.NET. This section attempts to cover most of them.. ADO.NET introduces the *Dataset* object which can represent multiple tables, store relationship information, and provide disconnected access to data.

Dataset and Recordset

ADO uses the *Recordset* object to represent the entire set of records from a single base table. Even though you cannot store multiple tables in a recordset, it is possible to store data from multiple tables using a JOIN clause in the SELECT query that builds the recordset. The ADO.NET *DataSet* object is a collection of one or more tables, and the tables contained in the dataset are called data tables. These can be accessed using the DataTable objects. The *TablesCollection* object contains all the *DataTable* objects in a DataSet, each *DataTable* object corresponding to a table in the actual database.

The columns in a DataSet are represented using a *DataColumn* object. It is also possible to relate the tables in the dataset using the *DataRelation* objects. The *DataRelation* object uses the *DataColumn* object to relate two or more tables by employing the concept of a foreign key. This feature allows the user to implement more complex operations than were hitherto possible. The DataSet, with the ability to store multiple tables and the relationships between those tables, offers a feature-rich implementation.

From the user's perspective, a dataset is a representation of an actual database residing on the client's machine. After an upgrade, you can still continue to use your existing applications but will not be able to leverage the benefits of ADO.NET. The Microsoft ActiveX Data Objects type library is automatically upgraded and the code is modified to reflect the syntax of VB.NET during the upgrade.

Application Interoperability

Marshalling ADO disconnected recordsets was achieved through Component Object Model (COM). The disadvantage with COM marshalling is the restricted availability of data types. The data types that are available are what COM provides. By comparison, ADO.NET transmits datasets as XML streams. Since XML has no restriction on data types, the component consuming the datasets is free to use whatever appropriate data types it normally uses.

Transmitting a large disconnected ADO recordset over the network places enormous stress on the network resources. The increase in stress is directly proportional to the size of the recordset being transmitted. Though ADO does offer its own performance optimization, it still suffers because of its dependency on COM. The data type conversions are required for COM marshalling between components. ADO.NET, on the other hand, does not need to enforce any data conversions and data is marshaled as XML.

Disconnected ADO recordsets that are marshaled across intranets or the Internet suffer from restrictions imposed by firewalls. A firewall allows only HTML text to pass, preventing any operations that access system resources. Here again, ADO.NET scores over ADO in that ADO.NET passes data around as XML streams. The advantage of XML streams is that they are just text data. This allows the data to be transmitted using the HTTP protocol which most firewalls allow. However, if you have to pass an ADO recordset, you also need to package and send interface methods and parameters from the client to the server or vice-versa.

Cursor Location

The ADO *Recordset* object can be created by the application in two places: within the application as a client-side cursor, or within the data store as a server-side cursor. Client-side cursors are supported in ADO.NET by the *DataSet* object, while server-side cursors are implemented using the *DataReader* object.

When you upgrade an ADO application from Visual Basic 6.0 to Visual Basic .NET, the Upgrade Wizard modifies the Microsoft ActiveX Data Objects library as well, prompting your existing code to be altered in order to suit the .NET Framework. Any reference to the *CursorLocation* constants is upgraded to reflect the change. So, the values of *adUseClient* and *adUseServer* are upgraded to *ADODB.CursorLocationEnum.adUseClient* and *ADODB.CursorLocation .adUseServer*, respectively.

Disconnected Access

While ADO is primarily designed for connected access to the database, ADO.NET provides disconnected access to data. Whereas ADO communicates with the database by making calls to the OLEDB provider, or through any of the APIs provided by the DBMS, ADO.NET communicates through the *data adapter* object. The *DataSetCommand* object in turn makes calls to the OLEDB provider. The main difference between disconnected access in ADO and ADO.NET is that the *DataSetCommand* object optimizes performance, performs validity checks, and controls the way the changes are written to the database.

Data Navigation

The data in an ADO recordset can be accessed by calling any one of the move methods supported by the *Recordset* object. In ADO.NET, rows are represented as collections. This allows the programmer to work with them just like objects. New rows can be added through the *Add* method, rows can be deleted using the

Remove method, and rows can be accessed through an ordinal or a primary key, such as an index. *DataRelation* objects allow the programmer to maintain a master-detail relationship between the tables in the database. This means that as you move from one record to another in a master table, the corresponding records in the transaction table are made available.

The object models for ADO and ADO.NET are radically different. ADO does not support any of the objects present in the new object model. The choice here is to either retain the application as it is, or re-write your application to take advantage of the new features.

Lock Implementation

ADO holds up database locks and database connections for long durations that result in performance bottlenecks and resource contentions. The disconnected data access implemented by ADO.NET ensures that database locks or database connections are not held for longer periods of time. When you upgrade an existing Visual Basic 6.0 ADO application, the Upgrade Wizard converts the existing ADO type library to .NET metadata. The existing classes and their associated methods and properties can be used as they are, without any modifications.

Upgrading Interfaces

Earlier versions of Visual Basic used interfaces, but could not create them directly. Visual Basic .NET introduces this feature with the **Interface** statement, which allows you to define true interfaces. The interfaces you define are distinctly different from classes, and it is now possible to actually implement them using the enhanced **Implements** keyword.

Interfaces define the properties, methods, and events that classes can implement. The basic purpose of defining an interface is to logically group properties, methods, and events that represent a logical entity. Besides, it is also possible to extend interfaces by creating new interfaces from the old, and adding more functionality without breaking existing clients.

The main purpose of interfaces in any language is to allow the objects and their interfaces (methods, parameters, properties, and so on) to be designed for all the objects in a system. This allows developers to work concurrently on different objects without having to wait on one or the other to be implemented since the interfaces between the objects are defined.

An interface, unlike a class, does not provide implementation, it defines a contract between itself and a class. This is a two-way relationship. The class implementing the interface must implement all the methods, and the interface guarantees no change will be made to the existing interface. In order to incorporate functionality changes, you can create a new interface that inherits from the original version.

Visual Basic .NET uses the **Interface** and **End Interface** statements to define an interface. The property, method, and event definitions are then embedded within these statements. The following rules apply to interfaces:

- The **Inherits** statement follows the **Interface** statement if the interface inherits from another interface.
- The **Inherits** statement must be the first statement after the **Interface** statement. Only comments, if any, can precede the **Inherits** statement.
- The interface can only contain **Event**, **Sub**, **Function**, or **Property** statements. Interfaces cannot contain any implementation code or even **End Sub** or **End Function** statements.
- **Interface** statements are Public by default, but they can also be declared as Friend, Protected, or Private.

While an Interface declaration can contain any of the four modifiers just mentioned, it is not possible to declare a Sub, Function, or Property definition with any other modifier than the **OverLoads** or **Default** keywords. Therefore, a function cannot be declared with Public, Private, Friend, Protected, Shared, Overrides, MustOverride, or Overridable. The reasons for this restriction are described in the following bullet points:

- The Public modifier indicates the entity has unrestricted access and can be accessed by any object, even those that do not implement the interface containing this entity. This disassociates the entity as a member of the interface.
- The Private modifier restricts the entity's access to only its declaration context, thereby rendering the entity totally inaccessible.
- The Friend modifier restricts access to only the program that contains the entity declaration.
- The Protected modifier restricts the entity's access to only those interfaces that derive from this class. As a result, those classes that implement this interface have no access to the entity.

- An entity declared with the Shared modifier does not operate on a specific instance of a type, meaning that the entity can be invoked directly from a class rather than from the instance.
- The Overrides modifier indicates that the entity will be overridden in the derived class. This goes against the contract between the Interface and the class, being that a class is required to implement all the entities in the interface without any changes.
- The MustOverride modifier means that the derived class must override the entity in order to be creatable.
- The Overridable modifier indicates that the entity can be overridden.

The **Implements** keyword signifies that a class member implements a specific interface. An **Implements** statement requires a comma-separated list of values, each value representing a single interface member that is implemented in a class. Normally, only a single interface member is specified, but it is also possible to implement multiple members. The interface member is specified in the following format:

```
<InterfaceName.InterfaceMember>
```

The method that implements the entity need not follow the Visual Basic 6.0 convention of *InterfaceName_MethodName*. The method name can be any legal identifier. The following code fragment illustrates a method that implements an interface:

```
Function Add(ByVal intOper1 as integer, ByVal intOper2 as integer) as_
    Integer Implements ICalculator.Add
    Add = intOper1 + intOper2
End Function
```

The implementing method's function signature should match the Interface method's function signature. In other words, the data type of the arguments, return value, and so on, must be exactly the same. You can declare the method that implements the interface member with any of the legal modifiers allowed on the instance method declarations. The legal modifiers are Overloads, Overrides, Overridable, Public, Private, Protected, Friend, Protected Friend, MustOverride, Default, and Static.

The **Implements** statement can also be used to declare a single method that implements multiple methods of multiple interfaces. This feature comes in handy when all the methods exhibit the same functionality. Consider the following code fragment:

```
Sub Method1 Implements InterfaceA.Method1, InterfaceB.Method2, _
InterfaceC.Method3, InterfaceD.Method4
'Visual Basic Code
End Sub
```

Upgrading Interfaces from Visual Basic 6.0

Visual Basic 6.0 allowed only components consume interfaces. There was no way an interface could actually be created. There might be situations when you decide to upgrade your component to Visual Basic .NET to take advantage of a lot of new features. When you do the upgrade, the Upgrade Wizard will make changes to your existing component to make it compatible with Visual Basic .NET. This section is devoted to demystifying the changes the Upgrade Wizard will make to your existing component.

Consider the following program, written in Visual Basic 6.0, which implements a simple calculator. The ICalculator interface implements four simple functions of Add, Subtract, Multiply, and Divide. Each of the functions accepts two integers and returns a third integer as a result. The *clsCalc* class implements all the interface methods. The client code, meanwhile, is implemented in a Form.

The code for the ICalculator interface is shown in the following:

CD File 14-1

```
Function Add(intOper1 As Integer, intOper2 As Integer) As Integer

End Function
Function Subtract(intOper1 As Integer, intOper2 As Integer) As Integer

End Function
Function Divide(intOper1 As Integer, intOper2 As Integer) As Integer

End Function
Function Multiply(intOper1 As Integer, intOper2 As Integer) As Integer

End Function
```

The code implementing the ICalculator interface is shown next.

CD File 14-2

```
Implements _ICalc
Private Function ICalc_Add(intOper1 As Integer, _
                             intOper2 As Integer) As Integer
    ICalc_Add = intOper1 + intOper2
End Function

Private Function ICalc_Divide(intOper1 As Integer, _
                                 intOper2 As Integer) As Integer
    ICalc_Divide = intOper1 \ intOper2
End Function

Private Function ICalc_Multiply(intOper1 As Integer, _
                                   intOper2 As_ Integer) As Integer
    ICalc_Multiply = intOper1 * intOper2
End Function

Private Function ICalc_Subtract(intOper1 As Integer, _
                                   intOper2 As Integer) As Integer
    ICalc_Subtract = intOper1 - intOper2
End Function
```

CD File 14-3

The client uses the methods of the ICalculator interface as illustrated in the following:

```
Private Sub TestInterface()
    Dim objCalc As ICalc
    Set objCalc = New clsCalc

    MsgBox objCalc.Add(10, 20)
End Sub
```

CD File 14-4

The succeeding code segment illustrates the changes made to the ICalculator interface after the project is upgraded to Visual Basic .NET:

```
Namespace Project1
    Interface _ICalc
        Function Add(ByRef intOper1 As Short, _
                       ByRef intOper2 As Short) As Short
        Function Subtract(ByRef intOper1 As Short, _
                          ByRef intOper2 As_ Short) As Short

        Function Divide(ByRef intOper1 As Short, _
                          ByRef intOper2 As_ Short) As Short

        Function Multiply(ByRef intOper1 As Short, _
                          ByRef intOper2 As_ Short) As Short
    End Interface

Public  Class ICalc
    Implements _ICalc
 Function Add(ByRef intOper1 As Short, ByRef intOper2
    As Short) As _
                          Short Implements _ICalc.Add

     End Function
Function Subtract(ByRef intOper1 As Short, ByRef intOper2
    As Short) As_
                          Short Implements  _ICalc.Subtract

     End Function
Function Divide(ByRef intOper1 As Short, ByRef intOper2 As Short) As _
          Short Implements _ICalc.Divide

     End Function
Function Multiply(ByRef intOper1 As Short, ByRef intOper2
    As Short) As _
                          Short Implements    _ICalc.Multiply

     End Function
End Class

End NameSpace
```

The changes that the Upgrade Wizard makes to the existing code are quite noteworthy:

1. The Upgrade Wizard creates a new interface called _ICalc. This name is the name of the Visual Basic 6.0 class that holds the interface definitions.
2. The interface definitions are coded within the Interface…End Interface. The End Functions are also removed so that definitions contain only the function names without any implementation code.
3. The integer data type is changed to short.
4. The Upgrade Wizard then creates a new class called *ICalc* that derives from _ICalc. This wrapper class contains interface member definitions with the partial implementation.
5. Then another derived class, which actually contains the full implementation of the four functions, is created. This class is called *clsCalc*, the contents of which are shown in the following:

CD File 14-5

```
Namespace Project1
    Public  Class clsCalc
    Implements _ICalc

     Private Function ICalc_Add(ByRef intOper1 As Short,
                            ByRef_ intOper2 As Short) As Short
                                         Implements _ICalc.Add
                     ICalc_Add = intOper1 + intOper2
     End Function

     Private Function ICalc_Divide(ByRef intOper1 As Short,
                              ByRef intOper2 As Short) As Short
                                      Implements _ICalc.Divide
          ICalc_Divide = intOper1 \ intOper2
     End Function

     Private Function ICalc_Multiply(ByRef intOper1 As Short,
                             ByRef intOper2 As Short) As Short
                                     Implements _ICalc.Multiply
          ICalc_Multiply = intOper1 * intOper2
     End Function

     Private Function ICalc_Subtract(ByRef intOper1 As Short,
                              ByRef intOper2 As Short) As Short
                                      Implements _ICalc.Subtract
           ICalc_Subtract = intOper1 - intOper2
     End Function

     End Class
End NameSpace
```

6. The *clsCalc* class implements the _ICalc interface and contains code that implements the four functions.
7. The client instantiates the _ICalc interface and assigns a reference to the *clsCalc* class. Afterward, the *Add* method is called with two short values. The code for the client is shown in the following:

CD File 14-6

```
Public Sub TestInterface()
    Dim objCalc As _ICalc
    objCalc = New clsCalc

    MsgBox(CStr(objCalc.Add(10, 20)))
End Sub
```

Using the Upgrade Tool

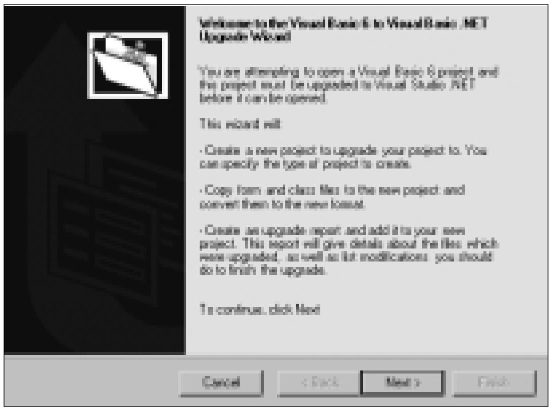
Visual Basic .NET is a paradigm shift from the previous versions of Visual Basic, and there are a lot of advantages in upgrading to it. Exercise 14.2 walks you through this process.

Exercise 14.2 Using the Upgrade Wizard

1. The upgrade tool is launched as soon as you open a Visual Basic 6.0 project in Visual Basic .NET. Figure 14.1 shows you the initial screen of the Upgrade Wizard. The first step in the wizard summarizes the actions that will be done throughout the Upgrade Wizard. It then creates a new Visual Basic .NET project in a separate folder you specify, leaving your

existing project unchanged. (It is important to note that a Visual Basic .NET project cannot be opened in Visual Basic 6.0.)

Figure 14.1 Step 1 of 5 of the Upgrade Wizard



Welcome to the Visual Basic 6 to Visual Basic .NET Upgrade Wizard

You are attempting to open a Visual Basic 6 project and the project must be upgraded to Visual Studio .NET before it can be opened.

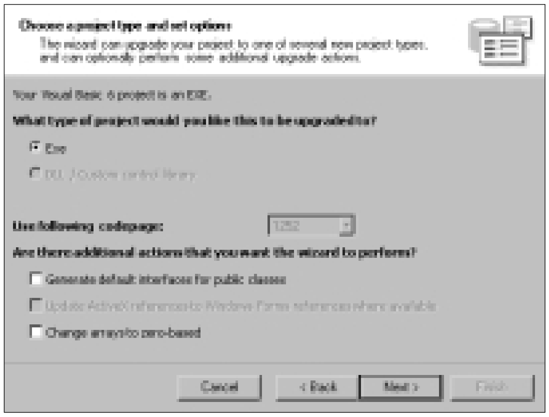
This wizard will:

- Create a new project to upgrade your project to. You can specify the type of project to create.
- Copy form and class files to the new project and convert them to the new format.
- Create an upgrade report and add it to your new project. This report will give details about the files which were upgraded, as well as list modifications you should do to finish the upgrade.

To continue, click Next

2. After the initial screen is displayed, the next step in the upgrade process is to choose what kind of project the existing project should be upgraded to, as well as configuring certain other options. The Upgrade Wizard determines the project type of the existing Visual Basic 6.0 project and selects the appropriate option. It also displays the existing project type in the first line. For internationalization projects, you can select the code page that translates the project. This step also allows you to configure other options. You can decide if you want to generate default interfaces for public classes, update ActiveX references to Windows Forms, and make arrays zero-based. Refer to Figure 14.2.

Figure 14.2 Step 2 of 5 of the Upgrade Wizard



Choose a project type and set options

The wizard can upgrade your project to one of several new project types, and can optionally perform some additional upgrade actions.

Your Visual Basic 6 project is an EXE.

What type of project would you like this to be upgraded to?

[✓ Exe]
DLL / Custom control library

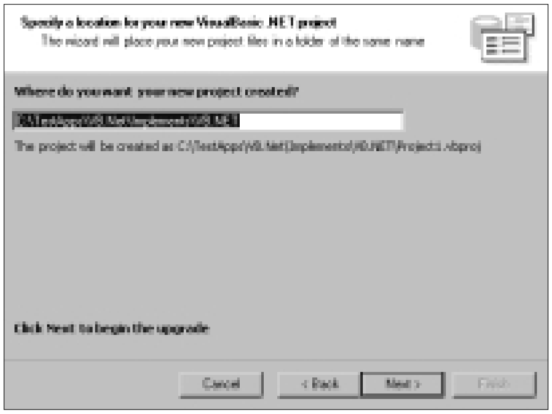
Use following codepage: 1252

Are there additional actions that you want the wizard to perform?

☐ Generate default interfaces for public classes
☐ Update ActiveX references to Windows Forms references where available
☐ Change arrays to zero-based

3. Step 3 in the upgrade process is to specify where the new upgraded project should be created. Note that your existing project will be left as is, and that the upgraded project won't be able to be opened in previous versions of Visual Basic. Figure 14.3 displays this process.

Figure 14.3 Step 3 of 5 of the Upgrade Wizard



Specify a location for your new Visual Basic .NET project

The wizard will place your new project files in a folder of the same name

Where do you want your new project created?

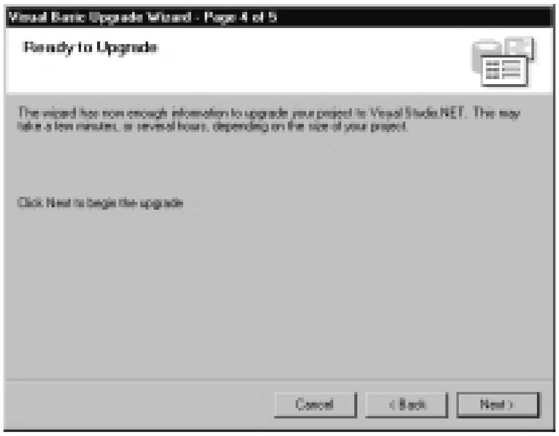
C:\TestApps\VB.Net\Implement\VB.NET

The project will be created as C:\TestApps\VB.Net\Implement\VB.NET\Project1.vbproj

Click Next to begin the upgrade

4. Figure 14.4 shows the screen informing the user that the project is ready to be upgraded. If you are set to proceed, click **Next**. When the project is upgraded, the language is modified for any syntax changes and Visual Basic Forms is converted to Windows Forms. More often than not, you will have to make changes to the code after it is upgraded. This is necessary because certain objects either have no equivalents, or the properties of some objects have been either erased or renamed.

Figure 14.4 Step 4 of 5 of the Upgrade Wizard



Visual Basic Upgrade Wizard - Page 4 of 5

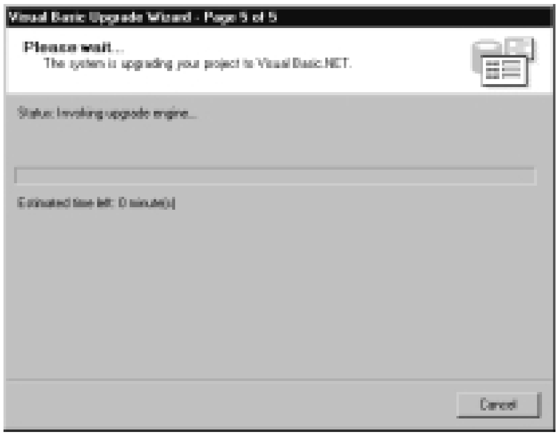
Ready to Upgrade

The wizard has now enough information to upgrade your project to Visual Studio.NET. This may take a few minutes, or several hours, depending on the size of your project.

Click Next to begin the upgrade

5. The last screen (Figure 14.5) shows the current status of the upgrade process. After the project is upgraded, the Upgrade Wizard creates an upgrade report that itemizes problems and inserts comments in the code, informing the programmer of what changes should be made. It is not difficult to find the parts of the code that need updating, because the Upgrade Wizard marks that code which needs changing, even including comments with the designation. The comments begin with the text TODO, and the IDE picks up these statements and lists them in the TaskList window. Navigating to the appropriate line is as easy as double-clicking the item in the TaskList window. Each item in the upgrade report is even associated with a related online help topic, which not only explains the need to change the code, but how to do it.

Figure 14.5 Step 5 of 5 of the Upgrade Wizard


Visual Basic Upgrade Wizard - Page 5 of 5

Please wait...
The system is upgrading your project to Visual Basic.NET.

Status: Invoking upgrade engine...

Estimated time left: 0 minute(s)

After the upgrade is completed, the Upgrade Wizard attaches various comments to the upgraded code. These can be categorized into the following four types, based on their severity:

- **UPGRADE_ISSUE** errors are items that will generate build errors and prevent the application from compiling. As a result, they are marked as compiler errors. It is necessary to correct them before running the project. These errors are logged in the upgrade report, as well as the Task List.
- **UPGRADE_TODO** errors are items that will not hinder the compilation process, but that do result in runtime errors. It is necessary to correct them before the solution will run successfully. These are reported in the upgrade report as both items in the TaskList and comments in the code.

- **UPGRADE_WARNING** errors are items that will not result in compiler errors but that might still cause an error when referencing the item during runtime. It is not absolutely necessary to rectify them, but doing so will result in a smoother execution of the project. These items are outlined in the upgrade report as both items in the TaskList window and comments in the code.
- **UPGRADE_NOTE** indicates serious changes in the code. Upgrade notes are added when there is a major structural change in code. Reported as comments in code, you should read them thoroughly before deciding on a course of action.

After the Visual Basic 6.0 project has been upgraded to Visual Basic .NET, an upgrade report is added to the project. This report contains the following details and is named _Upgradereport.htm:

- Project name
- Time of upgrade
- Upgrade settings consist of the following key-value pairs:
  - A Boolean value indicating whether ADO+ was used.
  - A Boolean value indicating whether the user requested the Upgrade Wizard to generate public interfaces for classes.
  - The name of the logfile.
  - The kind of project this project migrated from.
  - A Boolean value to indicate if the user preferred to change the arrays to zero-based.
  - The path to the output directory.
  - The name of the project that was created.
  - The actual path to the project that was created.
- A list of project files with information regarding the new filename, the old filename, file type, status, errors, warnings, as well as other issues.

Summary

There are special considerations that must be taken into account when migrating your Visual Basic 6.0 applications to .NET. For instance, you should bind variables because of the changes to property names and data type changes. In addition, you should make sure to avoid null propagation, use ADO for all database applications, use the Date data type to store dates, avoid fixed-length strings, and use constants instead of underlying values. There are changes that have been made to the application architecture in .NET, so it is advisable to port all your applications to ADO before they are moved to .NET.

Data types have also undergone a significant number of changes. Visual Basic .NET, for example uses the Object data type instead of the Variant data type. It also introduces a new data type called Short to store 16-bit numbers. The Integer data type, meanwhile, has been modified to store 32-bit numbers, the Long data type now stores 64-bit numbers, and the underlying value of Boolean True has been changed to –1. Arrays, too, have undergone a change, in that the lower-bound value is now fixed as zero and is impossible to change.

A host of new features have been added to Windows Forms as well, effectively replacing the Visual Basic 6.0 forms. Keywords like **GoSub**, **Option Base**, **Lset**, **VarPtr**, **StrPtr**, **Set**, and **Def** are no longer supported. In addition, the functionality of the **GoTo** statement is restricted only to error-handlers now.

Several new modifiers to functions, meanwhile, have been introduced in Visual Basic .NET. The functionality of the **Return** statement has been extended to return a value to the calling function besides returning control. Visual Basic .NET has also introduced the new concept of function overloading, allowing multiple functions to have the same name with each function differing only in their signature. In other developments, the syntax of property procedures has changed, Visual Basic .NET no longer allows parameter-less default properties, and with no support for the **Set** statement, you cannot have a Set property procedure.

Visual Basic .NET allows interoperability with COM components using metadata, Runtime Callable Wrappers, and COM Callable Wrappers. This allows you to leverage the functionality of existing components. Visual Basic .NET also allows you to implement true interfaces, and introduces structured exception handling that uses the *Try…Catch…Finally* block to handle errors instead of just extending support for the **On error…Goto** statement. Database applications can now take advantage of the new data access mechanism called ADO.NET, as well, which is very different from the earlier connection-based ADO model.

Finally, the Upgrade Wizard is available to ensure the smooth and easy migration of your existing applications to .NET. Its various upgrade messages clearly outline the changes that must be made before your application is ready to run.

Solutions Fast Track

Considerations Before Upgrading

☑ Early binding of variables is necessary because Visual Basic .NET has introduced changes to property names and default properties.

☑ The **Null** keyword is not available in Visual Basic .NET. Instead, you can use the DBNull.Value available in the *System* namespace to specify a Null value.

☑ It is advisable to change the data type of date variables to the Date data type in your legacy applications to facilitate an easier migration.

☑ It is recommended that constants be used instead of the actual underlying value.

Considering Architecture Before Migrating

☑ DHTML and ActiveX Document applications cannot be upgraded to Visual Basic .NET.

☑ Visual Basic 6.0 Forms has been replaced with a new architecture called Windows Forms.

☑ DAO or RDO data-binding applications must be ported to ADO first before they can be moved to Visual Basic .NET.

Data Types

☑ All Variant data types will be converted to the Object data type during an upgrade.

☑ Visual Basic .NET introduces a new data type called Short. The Visual Basic 6.0 Integer data type is now represented by the Short data type (which stores 16-bit numbers), the Visual Basic 6.0 Long data type is now represented by the Integer data type (which stores 32-bit numbers), And the Long data-type stores 64-bit numbers.

- ☑ ToOADate and FromOADate are used to convert between the Double and Visual Basic 6.0 representation of the date value.
- ☑ The underlying value of the True has been changed from –1 to 1.
- ☑ Arrays in Visual Basic .NET are zero-based.
- ☑ Fixed-length strings are not supported in Visual Basic .NET.

Converting VB Forms to Windows Forms

- ☑ Windows Forms does not support the OLE Container control.
- ☑ Windows Forms contains two menu controls called MainMenu and ContextMenu.
- ☑ Image controls are not supported in Windows Forms.

Keyword Changes

- ☑ **GoSub**, **Option Base**, **VarPtr**, **StrPtr**, and **Def** keywords are not supported in Visual Basic .NET.
- ☑ **GoTo** can be used only for the purposes of error handling.
- ☑ Visual Basic .NET introduces three new operators to perform bitwise operations. They are BitOr, BitAnd, and BitXor.

Programming Differences

- ☑ Optional parameters must be supplied with a default value.
- ☑ The **Return** statement returns control to the calling program.
- ☑ The default parameter passing mechanism is ByVal.
- ☑ *ParamArray* parameters must be declared as Object.
- ☑ Overloading is implemented with the help of function signature.
- ☑ The Runtime Callable Wrapper (RCW) helps .NET clients talk to COM components.

- ☑ The COM Callable Wrapper (CCW) helps COM clients talk to .NET components.
- ☑ The Set property procedure is not supported in Visual Basic .NET.
- ☑ Visual Basic .NET supports default properties only if the properties can be parameterized.

Understanding Error Handling

- ☑ Visual Basic .NET introduces structured error handling with the help of the **Try**, **Catch**, and **Finally** statements.
- ☑ It is possible to have multiple **Catch** statements to handle multiple exceptions.
- ☑ You can also create custom exceptions by creating a class that derives from the *System.Exception* class.

Data Access Changes in Visual Basic .NET

- ☑ Visual Basic .NET introduces a new object model called ADO.NET.
- ☑ The *DataSet* object can hold multiple tables and store relationships between the tables.
- ☑ The *DataReader* object implements the server-side cursor.
- ☑ ADO.NET provides disconnected access to the database.

Upgrading Interfaces

- ☑ Interfaces are created using the **Interface** keyword.
- ☑ The **Implements** keyword is used to implement an interface.

Using the Upgrade Tool

- ☑ The Upgrade Wizard is automatically launched when you open a Visual Basic 6.0 project in Visual Basic .NET.
- ☑ The upgraded code is placed in a different folder than that containing the source.

☑ The Upgrade Wizard lists various upgrade messages indicating what changes must be made to the existing code to ensure a smooth run of the upgraded project.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: Can I manually invoke the Upgrade Wizard?

A: No, you cannot. Visual Basic .NET automatically invokes the wizard when you open an older version of a Visual Basic application.

Q: Should I only use structured exception handling to manage errors in my application?

A: Structured exception handling is a more comprehensive way to managing errors. However, you can still continue to use the **On Error** statements.

Q: Is the object model for ADO.NET frozen? Can I start port my ADO code to ADO.NET?

A: At present, it is unclear whether this will be the final draft of ADO.NET. Therefore, it is best to hold on to your existing ADO applications. You can still, however, use the current model to write new non-critical applications for use in your current environment.

Q: What is the purpose of the Microsoft Visual Basic 6.0 Compatibility library?

A: The Microsoft Visual Basic 6.0 compatibility library contains functions and methods that are a part of Visual Basic 6.0 but not Visual Basic .NET. The contents of this library are used so that your existing applications do not break down solely because the implementation of a concept differed between the two versions.

Index

Enter the Global Knowledge Chrysler PT Cruiser Sweepstakes

This sweepstakes is open only to legal residents of the United States who are Business to Business MIS/IT managers or staff and training decision makers, that are 18 years of age or older at time of entry. Void in Florida & Puerto Rico.

OFFICIAL RULES

No Purchase or Transaction Necessary To Enter or Win, purchasing will not increase your chances of winning.

1. How to Enter: Sweepstakes begins at 12:00:01 AM ET May 1, 2001 and ends 12:59:59 PM ET December 31, 2001 the ("Promotional Period"). There are four ways to enter to win the Global Knowledge PT Cruiser Sweepstakes: Online, at Trade shows, by mail or by purchasing a course or software. Entrants may enter via any of or all methods of entry.

[1] To be automatically entered online, visit our web at www.globalknowledge.com click on the link named Cruiser and complete the registration form in its entirety. All online entries must be received by 12:59:59 PM ET December 31, 2001. Only one online entry per person, per e-mail address. Entrants must be the registered subscriber of the e-mail account by which the entry is made.

[2] At the various trade shows, during the promotional period by scanning your admission badge at our Global Knowledge Booth. All entries must be made no later than the close of the trade shows. Only one admission badge entry per person.

[3] By mail or official entry blank available at participating book stores throughout the promotional period. Complete the official entry blank or hand print your complete name and address and day & evening telephone # on a 3"x5" card, and mail to: Global Knowledge PT Cruiser Sweepstakes, P.O. Box 4012 Grand Rapids, MN 55730-4012. Entries must be postmarked by 12/31/01 and received by 1/07/02. Mechanically reproduced entries will not be accepted. Only one mail in entry per person.

[4] By purchasing a training course or software during the promotional period: online at http://www.globalknowledge.com or by calling 1-800-COURSES, entrants will automatically receive an entry onto the sweepstakes. Only one purchase entry per person.

All entries become the property of the Sponsor and will not be returned. Sponsor is not responsible for stolen, lost, late, misdirected, damaged, incomplete, illegible entries or postage due mail.

2. Drawings: There will be five [5] bonus drawings and one [1] prize will be awarded in each bonus drawing. To be eligible for the bonus drawings, on-line entries, trade show entries and purchase entries must be received as of the dates listed on the entry chart below in order to be eligible for the corresponding bonus drawing. Mail in entries must be postmarked by the last day of the bonus period, except for the month ending 9/30/01 where mail in entries must be postmarked by 10/1/01 and received one day prior to the drawing date indicated on the entry chart below. Only one bonus prize per person or household for the entire promotion period. Entries eligible for one bonus drawing will not be included in subsequent bonus drawings.

Bonus Drawings	Month starting/ending 12:00:01 AM ET/11:59:59 PM ET	Drawing Date on or about
1	5/1/01-7/31/01	8/8/01
2	8/1/01-8/31/01	9/11/01
3	9/1/01-9/30/01	10/10/01
4	10/1/01-10/31/01	11/9/01
5	11/1/01-11/30/01	12/11/01

There will also be a grand prize drawing in this sweepstakes. The grand prize drawing will be conducted on January 8, 2002 from all entries received. Bonus winners are eligible to win the Grand prize.

All random sweepstakes drawings will be conducted by Marden-Kane, Inc. an independent judging organization whose decisions are final. All prizes will be awarded. The estimated odds of winning each bonus drawing are 1:60,000, for the first drawing and 1:20,000 for the second, third, fourth and fifth drawings, and the estimated odds of winning the grand prize drawing is 1:100,000. However the actual odds of winning will depend upon the total number of eligible entries received for each bonus drawing and grand prize drawings.

3. Prizes: Grand Prize: One (1) PT Cruiser 2002 model Approx. Retail Value (ARV) $18,000. Winner may elect to receive the cash equivalent in lieu of the car. Bonus Prizes: Five (5), awarded one (1) per bonus period. Up to $1,400.00 in self paced learning products ARV up to $1,400.00 each.

No substitutions, cash equivalents, except as noted, or transfers of the prize will be permitted except at the sole discretion of the Sponsor, who reserves the right to substitute a prize of equal or greater value in the event an offered prize is unavailable for any reason. Winner is responsible for payment of all taxes on the prize, license, registration, title fees, insurance, and for any other expense not specifically described herein. Winner must have and will be required to furnish proof of a valid driver's license. Manufacturers warranties and guarantees apply.

4. Eligibility: This sweepstakes is open only to legal residents of the United States, except Florida and Puerto Rico residents who are Business to Business MIS/IT managers or staff and training decision makers, that are 18 years of age or older at the time of entry. Employees of Global Knowledge Network, Inc and its subsidiaries, advertising and promotion agencies including Marden-Kane, Inc., and immediate families (spouse, parents, children, siblings and their respective spouses) living in the same household as employees of these organizations are ineligible. Sweepstakes is void in Florida and Puerto Rico and is subject to all applicable federal, state and local laws and regulations. By participating, entrants agree to be bound by the official rules and accept decisions of judges as final in all matters relating to this sweepstakes.

5. Notification: Winners will be notified by certified mail, return receipt requested, and may be required to complete and sign an Affidavit of Eligibility/Liability Release and, where legal, a Publicity Release, which must be returned, properly executed, within fourteen (14) days of issuance of prize notification. If these documents are not returned properly executed or are returned from the post office as undeliverable, the prize will be forfeited and awarded to an alternate winner. Entrants agree to the use of their name, voice and photograph/likeness for advertising and promotional purposes for this and similar promotions without additional compensation, except where prohibited by law.

6. Limitation of Liability: By participating in the Sweepstakes, entrants agree to indemnify and hold harmless the Sponsor, Marden-Kane, Inc. their affiliates, subsidiaries and their respective agents, representatives, officers, directors, shareholders and employees (collectively, "Releasees") from any injuries, losses, damages, claims and actions of any kind resulting from or arising from participation in the Sweepstakes or acceptance, possession, use, misuse or nonuse of any prize that may be awarded. Releasees are not responsible for printing or typographical errors in any instant win game related materials; for stolen, lost, late, misdirected, damaged, incomplete, illegible entries; or for transactions, or admissions badge scans that are lost, misdirected, fail to enter into the processing system, or are processed, reported, or transmitted late or incorrectly or are lost for any reason including computer, telephone, paper transfer, human, error; or for electronic, computer, scanning equipment or telephonic malfunction or error, including inability to access the Site. If in the Sponsor's opinion, there is any suspected or actual evidence of electronic or non-electronic tampering with any portion of the game, or if computer virus, bugs, unauthorized intervention, fraud, actions of entrants or technical difficulties or failures compromise or corrupt or affect the administration, integrity, security, fairness, or proper conduct of the sweepstakes the judges reserve the right at their sole discretion to disqualify any individual who tampers with the entry process and void any entries submitted fraudulently, to modify or suspend the Sweepstakes, or to terminate the Sweepstakes and conduct a random drawing to award the prizes using all non-suspect entries received as of the termination date. Should the game be terminated or modified prior to the stated expiration date, notice will be posted on http://www.globalknowledge.com. Any attempt by an entrant or any other individual to deliberately damage any web site or undermine the legitimate operation of the promotion is a violation of criminal and civil laws and should such an attempt be made, the sponsor reserves the right to seek damages and other remedies from any such person to the fullest extent permitted by law. Any attempts by an individual to access the web site via a bot script or other brute force attack or any other unauthorized means will result in the IP address becoming ineligible. Use of automated entry devices or programs is prohibited.

7. Winners List: For the name of the winner visit our web site www.globalknowledge.com on January 31, 2002.

8. Sponsor: Global Knowledge Network, Inc., 9000 Regency Parkway, Cary, NC 27512. Administrator: Marden-Kane, Inc. 36 Maple Place, Manhasset, NY 11030.