CHAPTER 1

INTRODUCTION

1.0 OBJECTIVES

After studying this chapter you will be able to :

- Describe Microsoft .Net Framework
- State visual basic .Net introduction
- Explain examples of visual basic .Net

1.1 INTRODUCTION

Previously the language was know as Visual Basic 6.0,then it is merged in to .Net framework when Microsoft announce to release .Net, with its release for the .NET platform, the Visual Basic language has undergone dramatic changes.

For example:

❖ The language itself is now fully object-oriented.

❖ Applications and components written in Visual Basic .NET have full access to the .NET Framework, an extensive class library that provides system and application services.

❖ All applications developed using Visual Basic .NET run within a managed runtime environment, the .NET common language runtime. In this introduction, I briefly discuss these changes and other changes before showing you three very simple, but complete, Visual Basic .NET applications.

1.2 MICROSOFT .NET FRAMEWORK

The *.NET Framework* encompasses the following:

❖ *A new way to expose operating system and other APIs.* For years, the set of Windows functionality that was available to developers and the way that functionality was invoked were dependent on the language environment being used. For example, the Windows operating system provides the ability to create windows (obviously). Yet, the way this feature was invoked from a C++ program was dramatically different from the way it was invoked from a Visual Basic program. With .NET, the way that operating system services are invoked is uniform across all languages (including code embedded in ASP.NET pages). This portion of .NET is commonly referred to as the *.NET Framework class library*.

❖ *A new infrastructure for managing application execution.* To provide a number of sophisticated new operating-system services—including code-level security, cross-language class inheritance, cross-language type compatibility, and hardware and operating-system independence, among others—Microsoft developed a new runtime environment known as the Common Language Runtime (CLR). The CLR includes the Common Type System (CTS) for cross-language type compatibility and the Common Language Specification (CLS) for ensuring that third-party libraries can be used from all .NET-enabled languages. To support hardware and operating-system independence, Microsoft developed the Microsoft Intermediate Language (MSIL, or just IL). IL is a CPU-independent machine language-style instruction set into which .NET Framework programs are compiled. IL programs are compiled to the actual machine language on the target platform prior to execution (known as *just-in-time*, or JIT, compiling). IL is never interpreted.

- *A new web server paradigm.* To support high-capacity web sites, Microsoft has replaced its Active Server Pages (ASP) technology with ASP.NET. While developers who are used to classic ASP will find ASP.NET familiar on the surface, the underlying engine is different, and far more features are supported. One difference, already mentioned in this chapter, is that ASP.NET web page code is now compiled rather than interpreted, greatly increasing execution speed.

- *A new focus on distributed-application architecture.*Visual Studio .NET provides top-notch tools for creating and consuming *web services* -- vendor-independent software services that can be invoked over the Internet. The .NET Framework is designed top to bottom with the Internet in mind. For example,

- ADO.NET, the next step in the evolution of Microsoft's vision of "universal data access," assumes that applications will work with disconnected data by default. In addition, the ADO.NET classes provide sophisticated XML capabilities, further increasing their usefulness in a distributed environment. An understanding of the .NET Framework is essential to developing professional Visual Basic .NET applications. The .NET Framework is explained in detail in Chapter 3.

1.1 & 1.2 Check Your Progress

Fill in the blanks

1. The CLR includes the ……………………. for cross language type compatibility.
2. The net framework is designed ………………….. with the internet.
3. Visual Basic NET has a …………………..….. complier.
4. The browser-based application requires a computer running ……………………… server.
5 CTS is the part of ………………………..

1.3 VISUAL BASIC .NET

Visual Basic .NET is the next generation of Visual Basic, but it is also a significant departure from previous generations. Experienced Visual Basic 6 developers will feel comfortable with Visual Basic .NET code and will recognize most of its constructs. However, Microsoft has made some changes to make Visual Basic .NET a better language and an equal player in the .NET world. These include such additions as a Class keyword for defining classes and an Inherits keyword for object inheritance, among others. Visual Basic 6 code can't be compiled by the Visual Basic .NET compiler without significant modification. The good news is that Microsoft has provided a migration tool to handle the task. The Visual Basic .NET language itself is detailed in Chapter 2. Over the last several months I have spent almost all of my time playing with .NET and writing Visual Basic .NET programs. As a user of Visual Basic since Version 4, I can tell you that I am pleased with this new technology and with the changes that have been made to Visual Basic. In my opinion, Microsoft has done it right.

1.3.1 An Example Visual Basic .NET Program

The first program to write is the same for all languages: Print the words hello, world
It has become a tradition for programming books to begin with a *hello, world* example. The idea is that entering and running a program—any program—may be the biggest hurdle faced by experienced programmers approaching a new platform or

language. Without overcoming this hurdle, nothing else can follow. This chapter contains three such examples: one that creates a console application, one that creates a GUI application, and one that creates a browser-based application. Each example stands alone and can be run as is. The console and GUI applications can both be compiled from the command line (yes, Visual Basic .NET has a command-line compiler!). The browser-based application requires a computer running Internet Information Server (IIS).

1.3.2 hello, world

This is the world's favorite programming example, translated to Visual Basic .NET:
Imports System

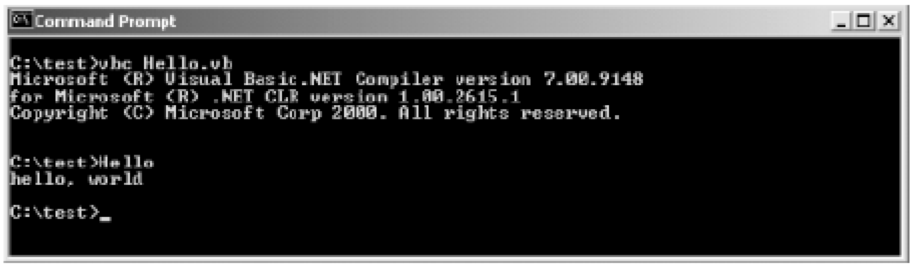
```
Public Module Hello
    Public Sub Main( )
        Console.WriteLine("hello, world")
    End Sub
End Module
```

The above version of *hello, world* is under *console application* -- it displays its output in a Windows commandprompt window. To compile this program, enter it using any text editor, such as Windows's Notepad, save it in a file whose name ends with *.vb*, such as *Hello.vb*, and compile it from the Windows command line with this command:-

```
vbc Hello.vb
```

The command vbc invokes the Visual Basic .NET command-line compiler, which ships with the .NET Framework SDK, and instructs it to compile the file named in the command-line argument. Compiling *Hello.vb* generates the file *Hello.exe*. After compiling, type Hello at the command line to run your program. Figure 1-1 shows the results of compiling and running this program.

Figure 1-1. Compiling and running hello, world



```
C:\test>vbc Hello.vb
Microsoft (R) Visual Basic.NET Compiler version 7.00.9148
for Microsoft (R) .NET CLR version 1.00.2615.1
Copyright (C) Microsoft Corp 2000. All rights reserved.


C:\test>Hello
hello, world

C:\test>_
```

If you're accustomed to programming in Visual Basic 6, you can see even from this little program that Visual Basic has changed dramatically. Here's a breakdown of what's happening in this code.

The first line:

```
Imports System
```

indicates that the program may use one or more types defined in the System *namespace*. (Types are grouped into namespaces to help avoid name collisions and to group related types together.) Specifically, the *hello, world* program uses the Console class, which is defined in the System namespace. The Imports statement is merely a convenience. It is not needed if the developer is willing to qualify type names with their namespace names. For example, the *hello, world* program could have been written this way:

```
Public Module Hello
    Public Sub Main( )
            System.Console.WriteLine("hello, world")
        End Sub
End Module
```

it is customary to use the “Imports” statement to reduce keystrokes and visual clutter. An important namespace for Visual Basic developers is Microsoft VisualBasic. The types in this namespace expose members that form Visual Basic's intrinsic functions and subroutines. For example: the Visual Basic *Trim* function is a member of the Microsoft.VisualBasic.Strings class, while the *MsgBox* function is a member of the Microsoft.VisualBasic.Interaction class. In addition, Visual Basic's intrinsic constants come from enumerations within this namespace. Much of the functionality available in this namespace, however, is also duplicated within the .NET ramework's Base Class Library. Developers who are not familiar with Visual Basic 6 will likely choose to ignore this namespace, favoring the functionality provided by the .NET Framework. The .NET Framework is introduced later in this chapter and is explained in detail in **Chapter 3**. Now, consider the following line:

```
Public Module Hello
```

This line begins the declaration of a standard module named Hello. The standard-module declaration ends with the following line:

```
End Module
```

In VB 6, various program objects were defined by placing source code in files having various filename extensions. **For example**, code that define classes was placed in *.cls* files, code that defined standard module was placed in *.bas* files, and so on. In Visual Basic .NET, all source files have *.vb* filename extensions, and program objects are defined with explicit syntax. For example, classes are defined with the keyword-

```
Class

End Class construct,
```

and standard modules are defined with the *Module…...End Module* construct. Any particular *.vb* file can contain as many of these declarations as desired. The purpose of standard modules in Visual Basic 6 was to hold code that was outside of any class definition. For example, global constants, global variables, and procedure libraries were often placed in standard modules. Standard modules in Visual Basic .NET serve a similar purpose and can be used in much the same way. However, in Visual Basic .NET they define datatypes that cannot be instantiated and whose members are all static. This will be discussed in more detail in next Chapter. The next line in the example begins the definition of a subroutine named
Main:

```
Public Sub Main( )
    ‘And It ends with
End Sub
```

This above syntax is similar to Visual Basic 6.Sub statement begins the definition of a *subroutine* – a method that has no return value. The *Main* subroutine is the entry point for the application. When the Visual Basic .NET compiler is invoked, it looks for a subroutine named *Main* in one of the classes or standard modules exposed by the application. If *Main* is declared in a class rather than in a standard module, the subroutine must be declared with the Shared modifier. This modifier indicates that the class does not need to be instantiated for the subroutine to be invoked. In either case, the *Main* subroutine must be Public. An example of enclosing the *Main* subroutine in a class rather than in a standard module is given at the end of this section. If no *Main* subroutine is found or if more than one is found, a compiler error is generated. The command-line compiler has a switch (/main:*location*) that allows you to specify which class or standard module contains the *Main* subroutine that is to be used, in the case that there is more than one. Lastly, there's the line that does the work:

```
Console.WriteLine("hello, world")
```

This code invokes the Console class's WriteLine method, which outputs the argument to the console. The WriteLine method is defined as a *shared* (also known as a *static*) method. Shared methods don't require an object instance in order to be invoked; nonshared methods do. Shared methods are invoked by qualifying them with their class name (in this case, Console). Here is a program that uses a class instead of a standard module to house its *Main* subroutine. Note that *Main* is declared with the Shared modifier. It is compiled and run in the same way as the standard module example, and it produces the same output. There is no technical reason to choose one implementation over the other.Code is as follows-

```
Imports System

Public Class Hello

    Public Shared Sub Main( )
        Console.WriteLine("hello, world")
    End Sub
End Class
```

1.3.3 Hello, Windows

Here's the GUI version of *hello, world*:

```
Imports System
Imports System.Drawing
Imports System.Windows.Forms
    Public Class HelloWindows
        Inherits Form
        Private lblHelloWindows As Label
            Public Shared Sub Main( )
                Application.Run(New HelloWindows( ))
            End Sub

            Public Sub New( )
                lblHelloWindows = New Label( )
                With lblHelloWindows
                    .Location = New Point(37, 31)
                    .Size = New Size(392, 64)
                    .Font = New Font("Arial", 36)
                    .Text = "Hello, Windows!"
                    .TabIndex = 0
                    .TextAlign = ContentAlignment.TopCenter
    End With
        Me.Text = "Programming Visual Basic .NET"
        AutoScaleBaseSize = New Size(5, 13)
        FormBorderStyle = FormBorderStyle.FixedSingle
        ClientSize = New Size(466, 127)
        Controls.Add(lblHelloWindows)
    End Sub
End Class
```

This is similar to the *hello, world* console application, but with extra stuff required since this is a GUI application. Two additional Imports statements are needed for drawing the application's window:

```
Imports System.Drawing
Imports System.Windows.Forms
```

The HelloWindows class has something that Visual Basic programs have never seen before, the Inherits statement:Inherits Form The Visual Basic .NET language has class inheritance. The HelloWindows class inherits from the Form class, which is defined in the System.Windows.Forms namespace. The next line declares a label control that will be used for displaying the text Hello, Windows:

```
Private lblHelloWindows As Label
```

The Label class is defined in the System.Windows.Forms namespace. As is the case with console applications, GUI applications must have a shared subroutine called *Main*:

```
Public Shared Sub Main( )
    Application.Run(New HelloWindows( ))
End Sub
```

This Main method creates an instance of the HelloWindows class and passes it to the Run method of the Application class (defined in the ystem.Windows.Forms namespace). The Run method takes care of the housekeeping of setting up a Windows *message loop* and hooking the HelloWindows form into it.
Next is another special method:

```
Public Sub New( )
```

Like *Main*, *New* has special meaning to the Visual Basic .NET compiler. Subroutines named *New* are compiled into *constructors*.

A constructor is a method that has no return value (but can have arguments) and is automatically called whenever a new object of the given type is instantiated.. The constructor in the HelloWindows class instantiates a Label object, sets some of its properties, sets some properties of the form, and then adds the Label object to the form's Controls collection. The interesting thing to note is how different this is from how Visual Basic 6 represented form design. In Visual Basic 6, form layout was represented by data in *.frm* files. This data was not code, but rather a listing of the properties and values of the various elements on the form. In Visual Basic .NET, this approach is gone. Instead, Visual Basic .NET statements must explicitly instantiate visual objects and set their properties. When forms are designed in Visual Studio .NET using its drag-and-drop designer, Visual Studio .NET creates this code on your behalf.
The command line to compile the *Hello, Windows* program is:

```
vbcHelloWindows.vb
/reference:System.dll,System.Drawing.dll,System.Windows.Forms.dll
/target:winexe
```

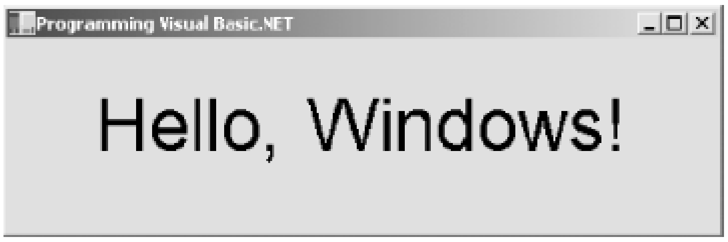
(Note that there is no break in the above line.)

The command line for compiling the *Hello, Windows* program has more stuff in it than the one for the console-based *hello, world* program. In addition to specifying the name of the *.vb* file, this command line uses the /references switch to specify three *.dll*s that contain the implementations of library classes used in the program (Form, Label, Point, etc.). The *hello, world* console application didn't require references when being compiled because all it used was the Console class, defined in the System namespace. The Visual Basic .NET command-line compiler includes two references implicitly:

mscorlib.dll (which contains the System namespace) and *Microsoft.VisualBasic.dll* (which contains helper classes used for implementing some of the features of Visual Basic .NET). Besides the /references switch, the command line for compiling the *Hello, Windows* program includes the /target switch. The /target switch controls what kind of executable code file is produced. The possible values of the /target switch are: exe Creates a console application. The generated file has an extension of *.exe*. This is the default. Winexe

Creates a GUI application. The generated file has an extension of *.exe*. library Creates a class library. The generated file has an extension of *.dll*. The output of *Hello, Windows* is shown in Figure 1-2.

Figure 1-2. Hello, Windows!



1.3.1-1.3.3 Check Your Progress

TRUE OR FALSE

1. The command vbe invokes the visual basic NET command-line compiler.
2. Imports System indicates that the program may use one or more types defined in the System namespace.
3. The Visual Basic Trim function is not member of the Microsoft. VisualBasic. Strings class.
4. The main subroutine is the entry point for the application.
5. Console application is the only programming technic in .Net.

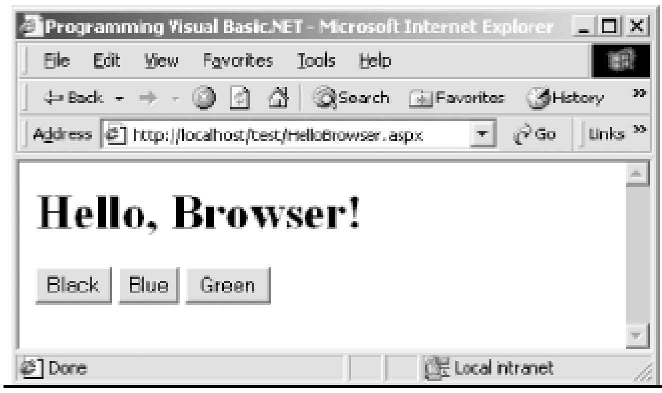
1.3.4 Hello, Browser

Here is a browser-based version of the *hello, world* application. Because the simplest version of such an application could be accomplished with only HTML, I've added a little spice. This web page includes three buttons that allow the end user to change the color of the text.

```
<script language="VB" runat="server">

    Sub Page_Load(Sender As Object, E As EventArgs)
        lblMsg.Text = "Hello, Browser!"
    End Sub
    Sub btnBlack_Click(Sender As Object, E As EventArgs)
        lblMsg.ForeColor = System.Drawing.Color.Black
    End Sub

    Sub btnGreen_Click(Sender As Object, E As EventArgs)
        lblMsg.ForeColor = System.Drawing.Color.Green
    End Sub
    Sub btnBlue_Click(Sender As Object, E As EventArgs)
        lblMsg.ForeColor = System.Drawing.Color.Blue
    End Sub
</script>

    <html>
    <head>
    <title>Programming Visual Basic .NET</title>
    </head>
    <body>
        <form action="HelloBrowser.aspx" method="post"
        runat="server">
        <h1><asp:label id="lblMsg" runat="server"/></h1>
        <p>
            <asp:button type="submit" id="btnBlack" text="Black"
                OnClick="btnBlack_Click" runat="server"/>
            <asp:button id="btnBlue" text="Blue"
                OnClick="btnBlue_Click" runat="server"/>
            <asp:button id="btnGreen" text="Green"
                OnClick="btnGreen_Click" runat="server"/>
        </p>
        </form>
    </body>
    </html>
```

To run this program, enter it using a text editor and save it in a file named *HelloBrowser.aspx*. Because the application is a web page that is meant to be delivered by a web server, it must be saved onto a machine that is running IIS and has the .NET Framework installed. Set up a virtual folder in IIS to point to the folder containing *HelloBrowser.aspx*. Finally, point a web browser to *HelloBrowser.aspx*. The output of the *Hello, Browser* application is shown in Figure 1-3.

Figure 1-3. Hello, Browser!


Programming Visual Basic.NET - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost/test/HelloBrowser.aspx
Go Links
Hello, Browser!
Black Blue Green
Done
Local intranet

Be sure to reference the file through the web server machine name or *localhost* (if the web server is on your local machine), so that the web server is invoked. For example, if the file is in a virtual directory called *Test* on your local machine, point your browser to h*ttp://localhost/Test/HelloBrowser.aspx*. If you point your browser directly to the file using a filesystem path, the web server will not be invoked. Going into detail on the *Hello, Browser* code would be too much for an introduction. However, I'd like to draw your attention to the <asp:label> and <asp:button> tags. These tags represent *serverside controls*. A server-side control is a class that is instantiated on the web server and generates appropriate output to represent itself on the browser. These classes have rich, consistent sets of properties and methods and can be referenced in code like controls on forms are referenced in GUI applications. ASP.NET has many other nifty features, some of which are:

- Web pages are compiled, resulting in far better performance over classic ASP.
- Code can be pulled out of web pages entirely and placed in *.vb* files (called *code-behind files*) that are referenced by the web pages. This separation of web page layout from code results in pages that are easier to develop and maintain.

1.4 SUMMARY

In this chapter we studied the overview of Microsoft .net framework. General introduction of VB .net, its similerties with old B-6.0

1.5 CHECK YOUR PROGRESS-*ANSWERS*

1.1 & 1.2

1. Common Type System
2. Top to Bottom
3. Command Line
4. Internet
5. CLR

1.3.1 & 1.3.3

1. True
2. True
3. False
4. True
5. False

1.6 QUESTIONS FOR SELF-STUDY

1. Explain Microsoft. NET Framework?
2. What is Visual basic. NET?

1.7 SUGGESTED READINGS

1. Programming Visual Basic .NET
   by Dave Grundgeiger
   Publisher: O'Reilly

References

Programming Visual Basic .NET by Dave Grundgeiger

◈◈◈

NOTES

CHAPTER 2

THE VISUAL BASIC .NET LANGUAGE

2.0 OBJECTIVES

After you study this chapter you will be able to explain :

- Source Files,Identifiers,Keywords, Literals,
- Types,Namespaces,Symbolic Constants
- Variables, Scope, Access Modifiers
- Assignment,
- Operators and Expressions
- Statements,
- Classes,Interfaces
- Structures,Enumerations, Exceptions
- Delegates,Events,Standers Modules
- Attributes,Conditional Compilation

2.1 INTRODUCTION

This chapter discusses the syntax of the Visual Basic .NET language, including basic concepts such as variables, operators, statements, classes, etc. Some material that you'd expect to find in this chapter will seem to be missing. For example, mathematical functions, file I/O, and form declarations are all very much a part of developing Visual Basic .NET applications, yet they are not introduced in this chapter because they are not intrinsic to the Visual Basic .NET language. They are provided by the .NET Framework and will be discussed in subsequent chapters. Additionally, Visual Basic

.NET functions that exist merely for backward compatibility with Visual Basic 6 are not documented in this chapter.

2.2 SOURCE FILES

Visual Basic .NET source code is saved in files with a *.vb* extension. The exception to this rule is when Visual Basic .NET code is embedded in ASP.NET web page files. Such files have an *.aspx* extension. Source files are plain-text files that can be created and edited with any text editor,like Notepad. Source code can be broken into as many or as few files as desired. When you use Visual Studio .NET, source files are listed in the Solution Explorer window, and all source is included from these files when the solution is built. When you are compiling from the command line, all source files must appear as command-line arguments to the compile command. The location of declarations within source files is unimportant. As long as all referenced declarations appear somewhere in a source file being compiled, they will be found. Unlike previous versions of Visual Basic, no special file extensions are used to indicate various language constructs (e.g., *.cls* for classes, *.frm* for forms, etc.). Syntax has been added to the language to differentiate various constructs. In addition, the pseudolanguage for specifying the graphical layout of forms has been removed. Form layout is specified by setting properties of form
objects explicitly within code. Either this code can be written manually, or the WYSIWYG form designer in Visual Studio .NET can write it.

2.3 IDENTIFIERS

Identifiers are names given to namespaces (discussed later in this chapter), types (enumerations, structures, classes, standard modules, interfaces, and delegates), type members (methods, constructors, events, constants, fields, and properties), and variables. Identifiers must begin with either an alphabetic or underscore character (_), may be of any length, and after the first character must consist of only alphanumeric and underscore characters. Namespace declarations may be declared either with identifiers or *qualified identifiers*. Qualified identifiers consist of two or more identifiers connected with the dot character (.). Only namespace declarations may use qualified identifiers. Consider this code fragment:

```
Imports System
Namespace ORelly.ProgVBNet
Public Class Hello
    Public Shared Sub SayHello( )
        Console.WriteLine("hello, world")
    End Sub
End Class
End Namespace
```

This code fragment declares three identifiers: OReilly.ProgVBNet (a namespace name), Hello (a class name), and SayHello (a method name). In addition to these, the code fragment uses three identifiers declared elsewhere: System (a namespace name), Console (a class name), and WriteLine (a method name). Although Visual Basic .NET is not case sensitive, the case of identifiers is preserved when applications are compiled. When using Visual Basic .NET components from case-sensitive languages, the caller must use the appropriate case.Ordinarily, identifiers may not match Visual Basic .NET keywords. If it is necessary to declare or use an identifier that matches a keyword, the identifier must be enclosed in square brackets ([]). Consider this code fragment:

```
Public Class [Public]
    Public Shared Sub SayHello( )
        Console.WriteLine("hello, world")
    End Sub
End Class
Public Class SomeOtherClass
    Public Shared Sub SomeOtherMethod( )
        [Public].SayHello( )
    End Sub
End Class
```

This code declares a class named Public and then declares a class and method that use the Public class. Public is a keyword in Visual Basic .NET. Escaping it with square brackets lets it be used as an identifier, in this case the name of a class. As a matter of style, using keywords as identifiers should be avoided, unless there is a compelling need. This facility allows Visual Basic .NET applications to use external components that declare identifiers matching Visual Basic .NET keywords.

2.4 KEYWORDS

Keywords are words with special meaning in a programming language. In Visual Basic .NET, keywords are reserved word; that is, they cannot be used as tokens for such purposes as naming variables and subroutines. The keywords in Visual Basic .NET are shown in Table 2-1.

Table 2-1. Visual Basic .NET keywords **Keyword Description**

AddHandler Visual Basic .NET Statement	AddressOf Visual Basic .NET Statement
Alias Used in the Declare statement	And Boolean operator
AndAlso Boolean operator	Ansi Used in the Declare statement
Append Used as a symbolic constant in the *FileOpen* function	As Used in variable declaration (Dim, Friend, etc.)
Assembly Assembly-level attribute specifier	Auto Used in the Declare statement
Binary Used in the Option Compare statement	Boolean Used in variable declaration (intrinsic data type)
ByRef Used in argument lists	Byte Used in variable declaration (intrinsic data type)
ByVal Used in argument lists	Programming Visual Basic .NET
25	Call Visual Basic .NET statement
Case Used in the Select Case construct	Catch Visual Basic .NET statement
CBool Data-conversion function	CByte Data-conversion function
CChar Data-conversion function	CDate Data-conversion function
CDec Data-conversion function	CDbl Data-conversion function
Char Used in variable declaration (intrinsic data type)	CInt Data-conversion function
Class Visual Basic .NET statement	CLng Data-conversion function
CObj Data-conversion function	Compare Used in the Option Compare statement
CShort Data-conversion function	CSng Data-conversion function
CStr Data-conversion function	CType Data-conversion function
Date Used in variable declaration (intrinsic data type)	Decimal Used in variable declaration (intrinsic data type)
Declare Visual Basic .NET statement	Default Used in the Property statement
Delegate Visual Basic .NET statement	Dim Variable declaration statement
Do Visual Basic .NET statement	Double Used in variable declaration (intrinsic data type)
Each Used in the For Each...Next construct	Else Used in the If...Else...ElseIf...End If construct
ElseIf Used in the If...Else...ElseIf...End If construct	End Used to terminate a variety of statements
EndIf Used in the If...Else...ElseIf...End If construct	Enum Visual Basic .NET statement
Erase Visual Basic .NET statement	Error Used in the Error and On Error compatibility statements
Event Visual Basic .NET statement	Explicit Used in the Option Explicit statement
False Boolean literal	For Used in the For...Next and For Each...Next constructs
Finally Visual Basic .NET statement	For Visual Basic .NET statement
Friend Statement and access modifier	Function Visual Basic .NET statement
Get Used in the Property construct	GetType Visual Basic .NET operator
GoTo Visual Basic .NET statement, used with the On Error statement	Handles Defines an event handler in a procedure declaration
	If Visual Basic .NET statement

Implements Visual Basic .NET statement	Imports Visual Basic .NET statement
In Used in the For Each...Next construct	Inherits Visual Basic .NET statement
Input Used in the *FileOpen* function	Integer Used in variable declaration (intrinsic data type)
Interface Visual Basic .NET statement	Is Object comparison operator
Let Reserved but unused in Visual Basic .NET	Lib Used in the Declare statement
Like Visual Basic .NET operator	Lock Function name
Long Used in variable declaration (intrinsic data type)	Loop Used in a Do loop
Me Statement referring to the current object instance	Mid String-manipulation statement and function
Mod Visual Basic .NET operator	Module Visual Basic .NET statement
MustInherit Used in the Class construct	MustOverride Used in the Sub and Function statements
MyBase Statement referring to an object's base class	MyClass Statement referring to the current object instance
Namespace Visual Basic .NET statement	New Object-creation keyword, constructor name
Next Used in the For...Next and For Each...Next constructs	Not Visual Basic .NET operator
Nothing Used to clear an object reference	NotInheritable Used in the Class construct
NotOverridable Used in the Sub, Property, and Function statements	Object Used in variable declaration (intrinsic data type)
Off Used in Option statements	On Used in Option statements
Option Used in Option statements	Optional Used in the Declare, Function, Property, and Sub statements
Or Boolean operator	OrElse Boolean operator
Output Used in the *FileOpen* function	Overloads Used in the Sub and Function statements
Overridable Used in the Sub and Function statements	Overrides Used in the Sub, Property, and Function statements
ParamArray Used in the Declare, Function, Property, and Sub statements	Preserve Used with the ReDim statement
Private Statement and access modifier	Property Visual Basic .NET statement
Protected Statement and access modifier	Public Statement and access modifier
Programming Visual Basic .NET	27
RaiseEvent Visual Basic .NET statement	Random Used in the *FileOpen* function
Read Used in the *FileOpen* function	ReadOnly Used in the Property statement
ReDim Visual Basic .NET statement	Rem Visual Basic .NET statement
RemoveHandler Visual Basic .NET statement	Resume Used in the On Error and Resume statements
Return Visual Basic .NET statement	Seek File-access statement and function
Select Used in the Select Case construct	Set Used in the Property statement
Shadows Visual Basic .NET statement	Shared Used in the Sub and Function statements
Short Used in variable declaration (intrinsic data type)	Single Used in variable declaration (intrinsic data type)
Static Variable declaration statement	Step Used in the For...Next construct
Stop Visual Basic .NET statement	String Used in variable declaration (intrinsic data type)
Structure Visual Basic .NET statement	Sub Visual Basic .NET statement
SyncLock Visual Basic .NET statement	Text Used in the Option Compare statement
Then Used in the If...Then...Else...EndIf construct	Throw Visual Basic .NET statement
To Used in the For...Next and Select Case constructs	True Boolean literal
Try Visual Basic .NET statement	TypeOf Used in variations of the If...Then...EndIf construct
Unicode Used in the Declare statement	Until Used in the For...Next construct
Variant Reserved but unused in Visual Basic .NET	When Used with the Try...Catch...Finally construct

While Used with the Do...Loop and While...End While constructs	With Visual Basic .NET statement
WithEvents Used in variable declaration (Dim, Public, etc.)	WriteOnly Used in the Property statement
XOr Visual Basic .NET operator	

2.5 LITERALS

Literals are representations of values within the text of a program. For example, in the following line of code, 10 is a literal, but x and y are not:

```
x = y * 10
```

Literals have data types just as variables do. The *10* in this code fragment is interpreted by the compiler as type Integer because it is an integer that falls within the range of the Integer type.

2.5.1 Numeric Literals

Any integer literal that is within the range of the Integer type (-2147483648 through 2147483647) is interpreted as type Integer, even if the value is small enough to be interpreted as type Byte or Short. Integer literals that are outside the Integer range but are within the range of the Long type (- 9223372036854775808 through 9223372036854775807) are interpreted as type Long. Integer literals outside the Long range cause a compile-time error. Numeric literals can also be of one of the floating point types—Single, Double, and Decimal. For example, in this line of code, 3.14 is a literal of type Double:

```
z = y * 3.14
```

In the absence of an explicit indication of type (discussed shortly), Visual Basic .NET interprets floating point literals as type Double. If the literal is outside the range of the Double type (-1.7976931348623157E308 through 1.7976931348623157E308), a compile-time error occurs.Visual Basic .NET allows programmers to explicitly specify the types of literals. Table 2-2 (shown later in this chapter) lists Visual Basic .NET's intrinsic data types, along with the method for explicitly defining a literal of each type. Note that for some intrinsic types, there is no way to write a literal.

2.5.2 String Literals

Literals of type String consist of characters enclosed within quotation-mark characters. For example, in the following line of code, "hello, world" is a literal of type String:Console.WriteLine("hello, world") String literals are not permitted to span multiple source lines. In other words, this is not permitted: ' Wrong Console.WriteLine("hello, world") To write a string literal containing quotation-mark characters, type the character twice for each time it should appear. For example:

```
Console.WriteLine("So then Dave said, ""hello, world"".")
```

This line produces the following output:So then Dave said, "hello, world".

2.5.3 Character Literals

Visual Basic .NET's Char type represents a single character. This is not the same as a one-character string; Strings and Chars are distinct types. Literals of type Char consist of a single character enclosed within quotation-mark characters, followed by the character c. For example, in the following code, "A"c is a literal of type Char:

```
Dim MyChar As Char
MyChar = "A"c
```

To emphasize that this literal is of a different data type than a single-character string, note that this code causes a compile-time error if Option Strict is On:

```
' Wrong.
Dim MyChar As Char
MyChar = "A"
```

The error is:

```
Option Strict On disallows implicit conversions from 'String' to 'Char'.
```

2.5.4 Date Literals

Literals of type Date are formed by enclosing a date/time string within number-sign characters. For example:

```
Dim MyDate As Date MyDate = #11/15/2001 3:00:00 PM#
```

Date literals in Visual Basic .NET code must be in the format m/d/yyyy, regardless of the regional settings of the computer on which the code is written.

2.5.5 Boolean Literals

The keywords True and False are the only Boolean literals. They represent the true and false Boolean states, respectively (of course!). For example:

```
Dim MyBoolean As Boolean
MyBoolean = True
```

2.5.6 Nothing

There is one literal that has no type: the keyword Nothing. Nothing is a special symbol that represents an uninitialized value of any type. It can be assigned to any variable and passed in any parameter. When used in place of a reference type, it represents a reference that does not reference any object. When used in place of a value type, it represents an empty value of that type. For numeric types, this is 0 or 0.0. For the String type, this is the empty string (""). For the Boolean type, this is False. For the Char type, this is the Unicode character that has a numeric code of 0. For programmer-defined value types, Nothing represents an instance of the type that has been created but has not been assigned a value.

2.5.7 Summary of Literal Formats

Table 2-2 shows all of Visual Basic .NET's intrinsic types, as well as the format for writing literals of those types in programs.

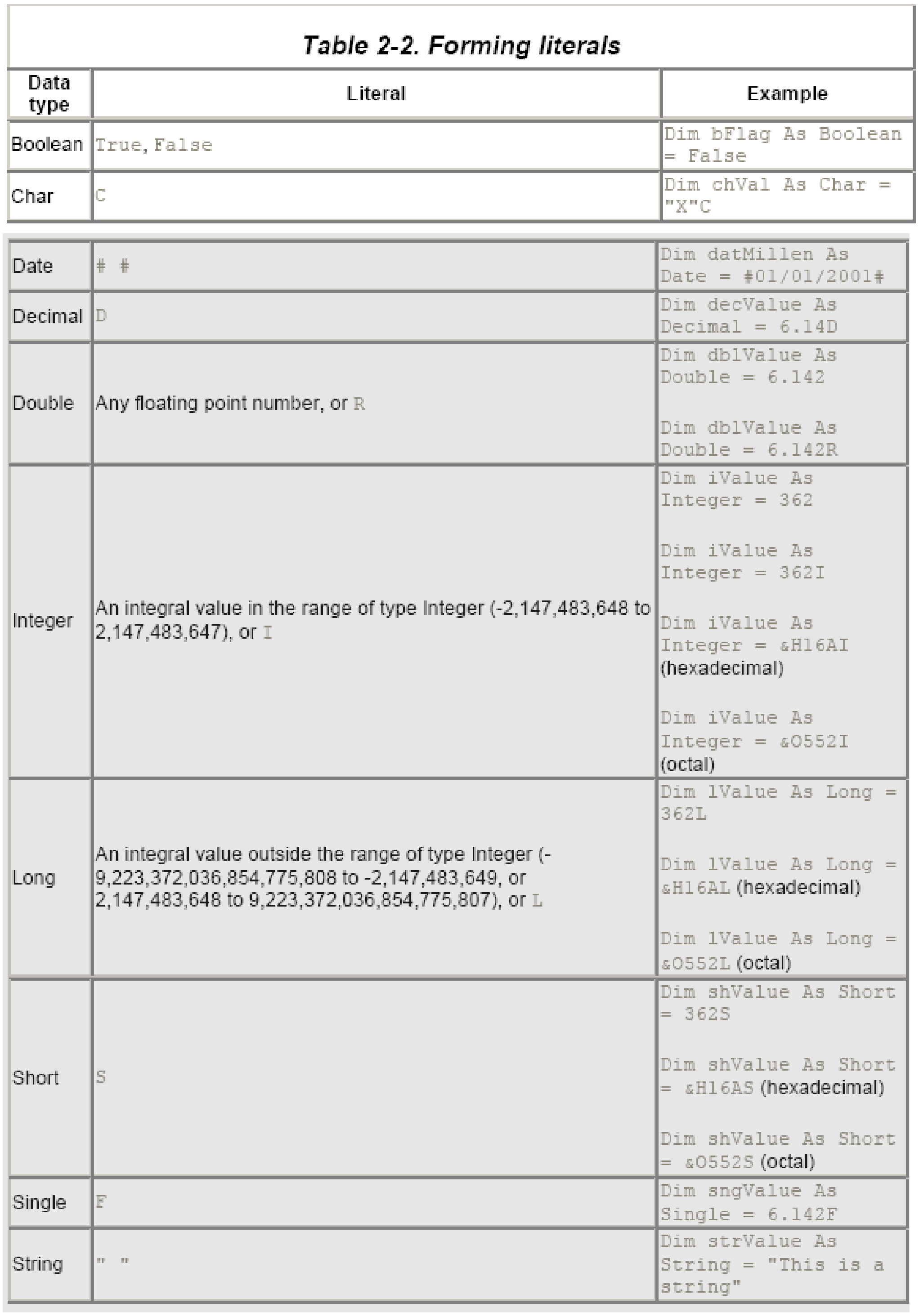


Table 2-2. Forming literals

Data type	Literal	Example
Boolean	`True`, `False`	`Dim bFlag As Boolean = False`
Char	`C`	`Dim chVal As Char = "X"C`
Date	`# #`	`Dim datMillen As Date = #01/01/2001#`
Decimal	`D`	`Dim decValue As Decimal = 6.14D`

Double	Any floating point number, or `R`	`Dim dblValue As Double = 6.142`
		
		`Dim dblValue As Double = 6.142R`

Integer	An integral value in the range of type Integer (-2,147,483,648 to 2,147,483,647), or `I`	`Dim iValue As Integer = 362`
		
		`Dim iValue As Integer = 362I`
		
		`Dim iValue As Integer = &H16AI` (hexadecimal)
		
		`Dim iValue As Integer = &O552I` (octal)

Long	An integral value outside the range of type Integer (-9,223,372,036,854,775,808 to -2,147,483,649, or 2,147,483,648 to 9,223,372,036,854,775,807), or `L`	`Dim lValue As Long = 362L` `Dim lValue As Long = &H16AL` (hexadecimal) `Dim lValue As Long = &O552L` (octal)
Short	`S`	`Dim shValue As Short = 362S` `Dim shValue As Short = &H16AS` (hexadecimal) `Dim shValue As Short = &O552S` (octal)
Single	`F`	`Dim sngValue As Single = 6.142F`
String	`" "`	`Dim strValue As String = "This is a string"`

Note the following facts about forming literals in Visual Basic .NET:

- There is no way to represent a literal of type Byte. However, this doesn't mean that literals cannot be used in situations where type Byte is expected. For example, the following code is fine:

  ```
  Dim MyByte As Byte = 100
  ```

- Even though the Visual Basic .NET compiler considers 100 to be of type Integer in this example, it recognizes that the number is small enough to fit into a variable of type Byte.

- Types not shown in Table 2-2 can't be expressed as literals.

2.1 - 2.4 Check Your Progress
Fill in the blanks

1. Visual Basic. NET files have an ……………………..…… extension.
2. ………………..…… declarations may be declared either with identifiers or qualified identifiers.
3. In Visual Basic. NET keywords are ……………………. which cannot be used as tokens.
4. ………….…………..….. are representations of values within the text of a program.

2.6 TYPES

Types in Visual Basic .NET are divided into two categories:

1. *value types* and
2. *reference types*.

Value types minimize memory overhead and maximize speed of access, but they lack some features of a fully object-oriented design (such as inheritance).

Reference types give full access to object-oriented features, but they impose some memory and speed overhead for managing and accessing objects. When a variable holds a value type, the data itself is stored in the variable. When a variable holds a reference type, a *reference* to the data (also known as a *pointer*) is stored in the variable, and the data itself is stored somewhere else.

Visual Basic .NET's primitive types include both value types and reference types (see "Fundamental Types" in this section). For extending the type system, Visual Basic .NET provides syntax for defining both new value types and new reference types (see "Custom Types" later in this section).

All reference types derive from the *Object type*. To unify the type system, value types can be treated as reference types when needed. This means that all types can derive from the Object type. Treating value types as reference types (a process known as *boxing*) is addressed later in this chapter.

2.6.1 Fundamental Types

Visual Basic .NET has several built-in types. Each of these types is an alias for a type supplied by the .NET architecture. Because Visual Basic .NET types are equivalent to the corresponding underlying .NET-supplied types, there are no type-compatibility issues when passing arguments to components developed in other languages. In code, it makes no difference to the compiler whether types are specified using the keyword name for the type or using the underlying .NET type name. For example, the test in this code fragment succeeds:

```
Dim x As Integer
Dim y As System.Int32
    If x.GetType() Is y.GetType( ) Then
        Console.WriteLine("They're the same type!")
    Else
        Console.WriteLine("They're not the same type.")
    End If
```

The fundamental Visual Basic .NET data types are:

Boolean

The Boolean type is limited to two values: True and False. Visual Basic .NET includes many logical operators that result in a Boolean type. For example:

```
Public Shared Sub MySub(ByVal x As Integer, ByVal y As Integer)
```

```
        Dim b As Boolean = x > y

        ' other code
    End Sub ' MySub
```

The result of the greater-than operator (>) is of type Boolean. The variable b is assigned the value True if the value in x is greater than the value in y and False if it is not. The underlying .NET type is System.Boolean.

Byte

The Byte type can hold a range of integers from 0 through 255. It represents the values that can be held in eight bits of data. The underlying .NET type is System.Byte.

Char

The Char type can hold any Unicode[1] character. The Char data type is new to Visual Basic .NET. The underlying .NET type is System.Char. [1] Unicode is a 16-bit character-encoding scheme that is standard across all platforms, programs, and languages (human and machine). See http://www.unicode.org for information on Unicode.

Date

The Date type holds values that specify dates and times. The range of values is from midnight on January 1, 0001 (0001-01-01T00:00:00) through 1 second before midnight on December 31, 9999 (9999-12-31T23:59:59). The Date type contains many members for accessing, comparing, and manipulating dates and times. The underlying .NET type is System.DateTime.

Decimal

The Decimal type holds decimal numbers with a precision of 28 significant decimal digits. Its purpose is to represent and manipulate decimal numbers without the rounding errors of the Single and Double types. The Decimal type replaces Visual Basic 6's Currency type. The underlying .NET type is System.Decimal.

Double

The Double type holds a 64-bit value that conforms to IEEE standard 754 for binary floating point arithmetic. The Double type holds floating point numbers in the range -1.7976931348623157E308 through 1.7976931348623157E308. The smallest nonnegative number (other than zero) that can be held in a Double is 4.94065645841247E-324. The underlying .NET type is System.Double.

Integer

The Integer type holds integers in the range -2147483648 through 2147483647. The Visual Basic .NET Integer data type corresponds to the VB 6 Long data type. The underlying .NET type is System.Int32.

Long

The Long type holds integers in the range -9223372036854775808 through 9223372036854775807. In Visual Basic .NET, Long is a 64-bit integer data type. The underlying .NET type is:

System.Int64.

Object

The Object type is the base type from which all other types are derived. The Visual Basic .NET Object data type replaces the Variant in VB 6 as the universal data type. The underlying .NET type is :

System.Object.

Short

The Short type holds integers in the range -32768 through 32767. The Short data type corresponds to the VB 6 Integer data type. The underlying .NET type is System.Int16.
Single

The Single type holds a 32-bit value that conforms to IEEE standard 754 for binary floating point arithmetic. The Single type holds floating point numbers in the range -3.40282347E38 through 3.40282347E38. The smallest nonnegative number (other than zero) that can be held in a Double is 1.401298E-45. The underlying .NET type is

System.Single.

String

The String type holds a sequence of Unicode characters. The underlying .NET type is System.String.Of the fundamental types, Boolean, Byte, Char, Date, Decimal, Double, Integer, Long, Short, and Single (that is, all of them except Object and String) are value types. Object and String are reference types.

2.6.2 Custom Types

Visual Basic .NET provides rich syntax for extending the type system. Programmers can define both new value types and new reference types. Types declared with Visual Basic .NET's Structure and Enum statements are value types, as are all .NET Framework types that derive from System.ValueType. Reference types include Object, String, all types declared with Visual Basic .NET's Class, Interface, and Delegate statements, and all .NET Framework types that don't derive from System.ValueType.

2.6.3 Arrays

Array declarations in Visual Basic .NET are similar to those in Visual Basic 6 and other languages. For example, here is a declaration of an Integer array that has five elements:

```
Dim a(4) As Integer
```

The literal 4 in this declaration specifies the upper bound of the array. All arrays in Visual Basic .NET have a lower bound of 0, so this is a declaration of an array with five elements, having indexes 0, 1, 2, 3, and 4. The previous declaration is of a variable named a, which is of type "array of Integer." Array types implicitly inherit from the .NET Framework's Array type (defined in the System namespace) and, therefore, have access to the methods defined in that type. For example, the following code displays the lower and upper bounds of an array by calling the Array class's GetLowerBound and GetUpperBound methods:

```
Dim a(4) As Integer
Console.WriteLine("LowerBound is"&a.GetLowerBound(0).ToString( ))
Console.WriteLine("UpperBound is"&a.GetUpperBound(0).ToString( ))
```

The output of above program is:

```
LowerBound is 0
UpperBound is 4
```

Note that the upper bound of the array is dynamic: it can be changed by methods available in the Array type. Array elements are initialized to the default value of the element type. A type's default value is determined as follows:

- ❖ For numeric types, the default value is 0.
- ❖ For the Boolean type, the default value is False.
- ❖ For the Char type, the default value is the character whose Unicode value is 0.
- ❖ For structure types (described later in this chapter), the default value is an instance of the structure type with all of its fields set to their default values.
- ❖ For enumeration types (described later in this chapter), the default value is an instance of the enumeration type with its internal representation set to 0, which may or may not correspond to a legal value in the enumeration.

- For reference types (including String), the default value is Nothing. You can access array elements by suffixing the array name with the index of the desired element enclosed in parentheses, as shown here:

```
For i = 0 To 4
        Console.WriteLine(a(i))
Next
```

Arrays can be multidimensional. Commas separate the dimensions of the array when used in declarations and when accessing elements. Here is the declaration of a three-dimensional array,where each dimension has a different size:

```
Dim a(5, 10, 15) As Integer
```

As with single-dimensional arrays, array elements are initialized to their default values.

2.6.3.1 Initializing arrays

Arrays of primitive types can be initialized by enclosing the initial values in curly brackets ({}). For
example:

```
Dim a( ) As String = {"First", "Second", "Third", "Fourth", "Fifth"}
```

Notice that when arrays are initialized in this manner, the array declaration is not permitted to specify an explicit size. The compiler infers the size from the number of elements in the initializer. To initialize multidimensional arrays, include the appropriate number of commas in the array-name declaration and use nested curly brackets in the initializer. Here is a declaration of a two-dimensional array having three rows and two columns:

```
Dim a(,) As Integer = {{1, 2}, {3, 4}, {5, 6}}
```

This declaration produces the following array:

```
a(0,0)=1, a(0,1)=2,
a(1,0)=3, a(1,1)=4,
a(2,0)=5, a(2,1)=6.
```

When initializing multidimensional arrays, the innermost curly brackets correspond to the rightmost dimension.

2.6.3.2 Dynamically allocating arrays

Use the New keyword to allocate arrays of any type. For example, this code creates an array of five Integers and initializes the elements as shown:

```
Dim a( ) As Integer
a = New Integer(4) {1, 2, 3, 4, 5}
```

If the array elements won't be initialized by the allocation, it is still necessary to include the curly brackets:

```
Dim a( ) As Integer
    ' allocates an uninitialized array of five Integers
a = New Integer(5)
    {
    }
```

Curly brackets are required so the compiler won't confuse the array syntax with constructor syntax. Note also the meaning of this declaration by itself:

```
Dim a( ) As Integer
```

This is the declaration of a reference that could point to a single-dimensional array of Integers, but doesn't yet. Its initial value is Nothing.

2.6.4 Collections

A *collection* is any type that exposes the ICollection interface (defined in the System.Collections namespace). In general, collections store multiple values and provide a way for iterating through those values.

Specialized collection types may also provide other means for adding and reading values. For example, the Stack type (defined in the System.Collections namespace) provides methods, such as Push and Pop, for performing operations that are appropriate for the stack data structure. The Visual Basic .NET runtime provides a type called Collection (defined in the Microsoft. VisualBasic namespace) that mimics the behavior of Visual Basic 6 collections and exposes the Icollection interface. Example 2-1 shows its use.

Example 2-1. Using the Collection type

```
 ' Create a new collection object.
     Dim col As New Collection( )
 ' Add some items to the collection.
     col.Add("Some value")
     col.Add("Some other value")
     col.Add("A third value")
 ' Iterate through the collection and output the strings.
     Dim obj As Object
     For Each obj In col
         Dim str As String = CType(obj, String)
         Console.WriteLine(str)
     Next
```

The Collection type's Add method adds items to the collection. Although strings are added to the collection in Example 2-2, the Add method is defined to take items of type Object, meaning that any type can be passed to the method. After items are added to the collection, they can be iterated using the For Each statement (discussed later in this chapter, under Section 2.13). Because the Collection class is defined to store items of type Object, the loop variable in the For Each statement must be of type Object. Because the items are actually strings, the code in Example 2-1 converts the Object references to String references using the *CType* function. Type conversions are discussed later in this section. The output of the code in Example 2-1 is:

```
Some value
Some other value
A third value
```

The items in a Collection object can also be iterated using a numerical index. The Collection object has a Count property, which indicates the number of items in the collection. Example 2-2 is precisely the same as Example 2-1, except that it iterates through the Collection object using a numerical index and a standard For loop.

Example 2-2. Using a numerical index on a collection object

```
 ' Create a new collection object.
     Dim col As New Collection( )
 ' Add some items to the collection.
     col.Add("Some value")
     col.Add("Some other value")
     col.Add("A third value")
 ' Iterate through the collection and output the strings.
     Dim i As Integer
     For i = 1 To col.Count
         Dim str As String = CType(col(i), String)
         Console.WriteLine(str)
     Next
```

Note that to access an item by index, the index number is placed within parentheses following the name of the Collection reference variable, as shown again here:

```
col(i)
```

The syntax of the Add method is:

```
Public Sub Add( _ByValItem As Object,
    _Optional ByVal Key As String = Nothing,
    _Optional ByValBefore As Object = Nothing,
    _Optional ByVal After As Object = Nothing )
```

The parameters are:

Item

The item to add to the collection.

Key

An optional string value that can be used as an index to retrieve the associated item. For example, the following code adds an item to a collection and then uses the key value to retrieve the item:

```
Dim col As New Collection( )
    col.Add("Some value", "Some key")
' ...
Dim str As String = CType(col("Some key"), String)
    Console.WriteLine(str)
```

The output is:Some value

Before

The item before which the new item should be added.

After

The item after which the new item should be added.

The .NET Framework class library provides several additional collection types.

2.6.5 Type Conversions

Visual Basic .NET provides a variety of ways for values of one type to be converted to values of another type. There are two main categories of conversions: *widening conversions* and *narrowing conversions*. Widening conversions are conversions in which there is no possibility for data loss or incorrect results. For example, converting a value of type Integer to a value of type Long is a widening conversion because the Long type can accommodate every possible value of the Integer type.Narrowing is the reverse operation—converting from a Long to an Integer—because some values of type Long can't be represented as values of type Integer. Visual Basic .NET performs widening conversions automatically whenever necessary. For example, a widening conversion occurs in the second line of the following code. The Integer value on the righthand side of the assignment is automatically converted to a Long value so it can be stored in the variable b:

```
Dim a As Integer = 5
Dim b As Long = a
```

A conversion that happens automatically is called an *implicit conversion*. Now consider the reverse situation:

```
Dim a As Long = 5
Dim b As Integer = a
```

The second line of code here attempts to perform an implicit narrowing conversion. Whether the compiler permits this line of code depends on the value set for the Option Strict compiler option. When Option Strict is On, attempts to perform an implicit widening conversion result in a compiler error. When Option Strict is Off, the compiler automatically adds code behind the scenes to perform the conversion. At runtime, if the actual value being converted is out of the range that can be represented by the target type, a runtime exception occurs. Option Strict can be set in either of two ways. First, it can be set in code at the top of a source file, like this:

```
Option Strict On
' ...
```

or:

```
Option Strict Off
' ...
```

The other way is to set a compiler switch, which affects all source files in the application. If you're compiling from the command line, specify /optionstrict+ on the command line to set Option Strict On. Specify

/optionstrict- to set Option Strict Off.

For example:

```
vbc MySource.vb /optionstrict+
```

To set Option Strict in Visual Studio .NET:

1. Right-click on the project name in the Solution Explorer window and choose Properties. This brings up the Project Property Pages dialog box. (If the Solution Explorer window is not visible, choose View Solution Explorer from the Visual Studio .NET main menu to make it appear.)
2. Within the Project Property Pages dialog box, choose the Common Properties folder. Within that folder, choose the Build property page. This causes the project-build options to appear on the right side of the dialog box.
3. Set the desired value for the Option Strict option.
   By default, Option Strict is Off, meaning that implicit narrowing conversions are allowed. This matches the default setting of Visual Basic 6. However, most experienced developers consider it
   beneficial to set Option Strict On so the compiler can help detect coding errors before they become runtime errors. Attempting to assign a Long to an Integer, for example, is usually a sign either that something was mistyped or that there is a problem with the design of the program. Setting Option Strict On helps the developer discover such errors at compile time. On the other hand, there may sometimes be a legitimate need to perform a narrowing conversion. Perhaps the application is interfacing to another application that passes a value as a Long, but it is guaranteed that the actual value passed will never be outside the range of the Integer type. Option Strict could be set to Off to allow implicit narrowing conversions, but a better alternative is to have Option Strict On (so it can protect the majority of the program) and to specify an *explicit* narrowing conversion. For example:

```
Dim a As Long = 5
Dim b As Integer = CInt(a)
```

This is known as an *explicit conversion* because the programmer is explicitly requesting a conversion to Integer. If at runtime a contains a value that is outside the Integer range, an exception is thrown.

Table 2-3 shows Visual Basic .NET's conversion functions.

Table 2-3. Conversion functions

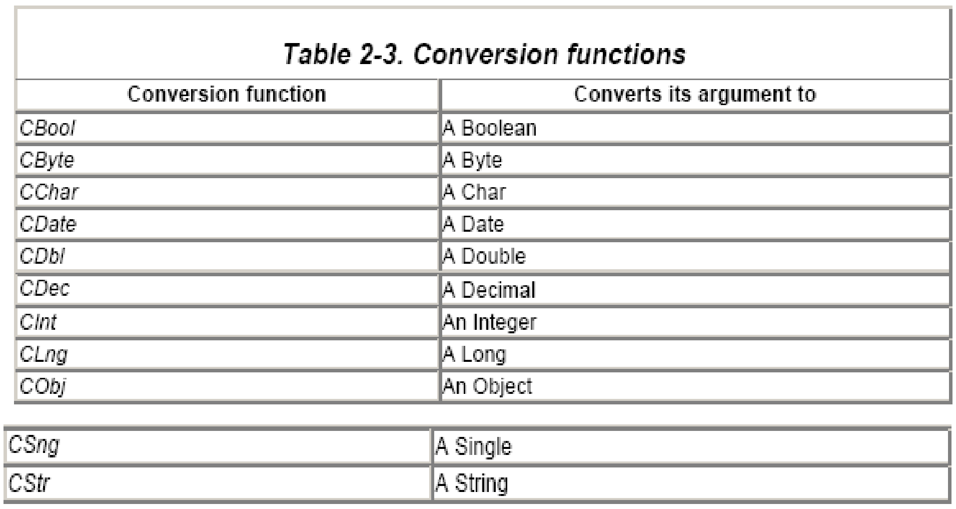


Table 2-3. Conversion functions

Conversion function	Converts its argument to
CBool	A Boolean
CByte	A Byte
CChar	A Char
CDate	A Date
CDbl	A Double
CDec	A Decimal
CInt	An Integer
CLng	A Long
CObj	An Object
CSng	A Single
CStr	A String

The functions shown in Table 2-3 all take a single argument. If the argument can't be converted to the given type, an exception is thrown. Consider the the following:

- ❖ When converting from any numeric value to Boolean, zero converts to False and nonzero converts to True.
- ❖ When converting from Boolean to a numeric value, False converts to 0 and True converts to -1.
- ❖ When converting from String to Boolean, the string must contain either the word "false", which converts to False, or the word "true", which converts to True. The case of the string is not important.
- ❖ When converting from Boolean to String, True converts to "True" and False converts to "False".
- ❖ Anything can be converted to type Object.

It's also possible to convert between reference types. Any object-reference conversion of a derived type to a base type is considered a widening conversion and can therefore be done implicitly.Conversely, conversion from a base type to a derived type is a narrowing conversion. As previously discussed, in order for narrowing conversions to compile, either Option Strict must be Off or an explicit conversion must be performed. Explicit conversions of reference types are done with the *CType* function. The *CType* function takes two arguments. The first is a reference to some object, and the second is the name of the type to which the reference will convert. At runtime, if a conversion is possible, the return value of the function is an object reference of the appropriate type. If no conversion is possible, an exception is thrown.

Here is an example of converting between base and derived classes:

```
' This is a base class.
Public Class Animal
' ...
End Class
' This is a derived class.
Public Class Cat
Inherits Animal
' ...
End Class
' This is another derived class.
Public Class Dog
Inherits Animal
' ...
End Class
' This is a test class.
Public Class AnimalTest
   Public Shared Sub SomeMethod( )
      Dim myCat As New Cat( )
      Dim myDog As New Dog( )
      Dim myDog2 As Dog
```

```
        Dim myAnimal As Animal = myCat ' Implicit conversion OK
        myAnimal = myDog ' Implicit conversion OK
        myDog2 = CType(myAnimal, Dog) ' Explicit conversion required
    End Sub
End Class
```

Object references can also be implicitly converted to any interface exposed by the object's class.

2.7 NAMESPACES

Thousands of types are defined in the .NET Framework. In addition, programmers can define new types for use in their programs. With so many types, name clashes are inevitable. To prevent name clashes, types are considered to reside inside of *namespaces*. Often, this fact can be ignored. For example, in Visual Basic .NET a class may be defined like this:

```
Public Class SomeClass
' ...
End Class
```

This class definition might be in a class library used by third-party customers, or it might be in the same file or the same project as the client code. The client code that uses this class might look something like this:

```
Dim x As New SomeClass( )
x.DoSomething( )
```

Now consider what happens if the third-party customer also purchases another vendor's class library, which also exposes a SomeClass class.

The Visual Basic .NET compiler can't know which definition of SomeClass will be used. The client must therefore use the *full name* of the type, also known as its *fully qualified name* .

Code that needs to use both types might look something like this:

```
' The namespace is "FooBarCorp.SuperFoo2100".
Dim x As New FooBarCorp.SuperFoo2100.SomeClass( )
x.DoSomething( )
' ...
' The namespace is "MegaBiz.ProductivityTools.WizardMaster".
Dim y As New MegaBiz.ProductivityTools.WizardMaster.SomeClass( )
y.DoSomethingElse( )
```

Note that a namespace name can itself contain periods (.). When looking at a fully qualified type name, everything prior to the final period is the namespace name. The name after the final period is the type name. Microsoft recommends that namespaces be named according to the format *CompanyName*.*TechnologyName*. For example,

"Microsoft.VisualBasic".

2.7.1 The Namespace Statement

So how does a component developer specify a type's namespace? In Visual Basic .NET, this can be done several ways. One is to use the Namespace keyword, like this:

```
Namespace MegaBiz.ProductivityTools.WizardMaster
Public Class SomeClass
' ...
End Class
End Namespace
```

Note that it is permissible for different types in the same source file to have different namespaces.A second way to provide a namespace is to use the

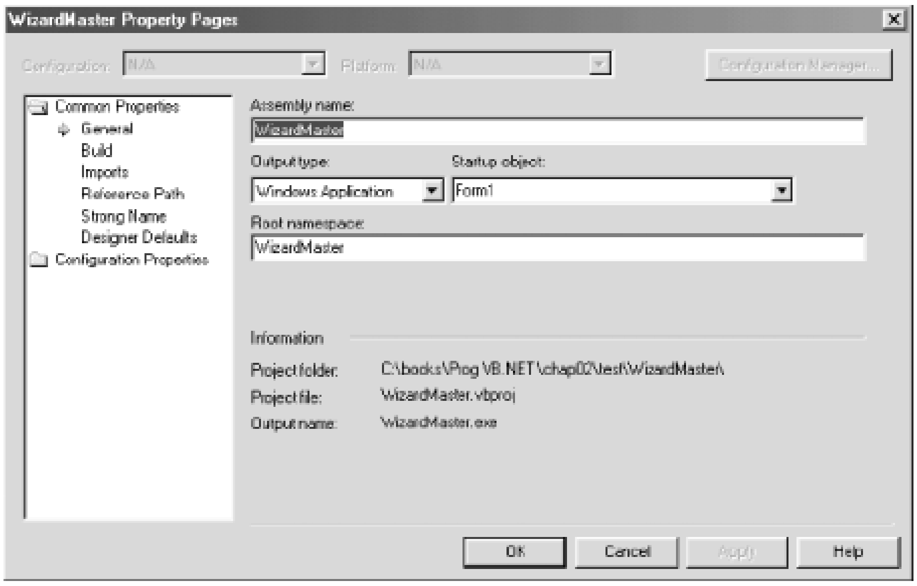
/rootnamespace switch on the Visual Basic .NET command-line compiler. For

example:

```
vbc src.vb /t:library /rootnamespace:MegaBiz.ProductivityTools.WizardMaster
```

All types defined within the compiled file(s) then have the given namespace.If you're compiling in the Visual Studio .NET IDE, the root namespace is specified in the Project Property Pages dialog box, which can be reached by right-clicking the project name in the Solution Explorer window of the IDE, then choosing Properties (see Figure 2-1 for the resulting WizardMaster Property Pages dialog). By default, Visual Studio .NET sets the root namespace equal to the name of the project.

Figure 2-1. Setting the root namespace in the Visual Studio .NET IDE


WizardMaster Property Pages

Configuration: N/A
Platform: N/A

Assembly name: WizardMaster
Output type: Windows Application
Startup object: Form1
Root namespace: WizardMaster

Information
Project folder: C:\books\Prog VB.NET\chap02\test\WizardMaster\
Project file: WizardMaster.vbproj
Output name: WizardMaster.exe

Note that regardless of which compiler is used (command line or Visual Studio .NET), if a root namespace is specified *and* the Namespace keyword is used, the resulting namespace will be the concatenation of the root namespace name and the name specified using the Namespace keyword.

2.7.2 The Imports Statement

So far, the discussion has implied that it's not necessary for the user of a type to specify the type's full name unless there is a name clash. This isn't exactly true. The CLR deals with types only in terms of their full names. However, because humans don't like to deal with long names, Visual Basic .NET offers a shortcut. As an example, the .NET Framework provides a drawing library, in which a type called Point is defined. This type's namespace is called System.Drawing, so the type's fully qualified name is:-

```
System.Drawing.Point
```

Code that uses this type might look like this:

```
Dim pt As System.Drawing.Point
pt.X = 10
pt.Y = 20
' ...
```

Typing the full name of every type whenever it is used would be too cumbersome, though, so Visual Basic .NET offers the Imports statement. This statement indicates to the compiler that the types from a given namespace will appear without qualification in the code. For example:

```
' At the top of the source code file:
Imports System.Drawing
' ...
' Somewhere within the source code file:
Dim pt As Point
pt.X = 10
pt.Y = 20
' ...
```

To import multiple namespaces, list each one in its own Imports statement. It's okay if multiple imported namespaces have some name clashes. For the types whose names clash, the full name must be specified wherever the type is used. The Imports statement is just a convenience for the developer. It does not set a reference to the assembly in which the types are defined. See the discussion of assemblies in Chapter 3 to learn how to reference assemblies that contain the types you need. Finally, note that namespaces, too, are just a convenience for the developer writing source code. To the runtime, a type is not "in" a namespace—a namespace is just another part of a type name. It Is perfectly acceptable for any given assembly to have types in different namespaces, and more than one assembly can define types in a single namespace.

2.8 SYMBOLIC CONSTANTS

Consider this function:

```
Public Shared Function RemainingCarbonMass(
_ByVal InitialMass As Double, _ByVal Years As Long _) As Double

Return InitialMass * ((0.5 ^ (Years / 5730)))

End Function
```

What's wrong with this code? One problem is readability. What does it mean to divide Years by 5730?
In this code, 5730 is referred to as a *magic number* -- one whose meaning is not readily evident from examining the code.

The following changes correct this problem:

```
Public Const CarbonHalfLifeInYears As Double = 5730
Public Shared Function RemainingCarbonMass(
_ByVal InitialMass As Double, _ByVal Years As Long _) As Double

Return InitialMass * ((0.5 ^ (Years / CarbonHalfLifeInYears)))

End Function
```

There is now no ambiguity about the meaning of the divisor.Another problem with the first code fragment is that a program filled with such code is hard to maintain.What if the programmer later discovers that the half-life of carbon is closer to 5730.1 years, and she wants to make the program more accurate? If this number is used in many places throughout the program, it must be changed in every case. The risk is high of missing a case or of changing a number that shouldn't be changed. With the second code fragment, the number needs to be changed in only one place. See also the discussion of read-only fields later in this chapter, under Section 2.14.

2.9 VARIABLES

A *variable* is an identifier that is declared in a method and that stands for a value within that method.Its value is allowed to change within the method. Each variable is of a particular type, and that type is indicated in the declaration of the variable. For example, this line declares a variable named **i** whose type is Integer:

```
Dim i As Integer
```

The keyword Dim indicates a variable declaration. *Dim* is short for *dimension* and dates back to the original days of the BASIC programming language in the late 1960s.

Variable identifiers may be suffixed with *type characters* that serve to indicate the variable's type. For example, this line declares a variable of type Integer:

```
    Dim x%
```

The effect is precisely the same as for this declaration:

```
Dim x As Integer
```

The set of type characters is shown in Table 2-4; note that not all data types have a type character.

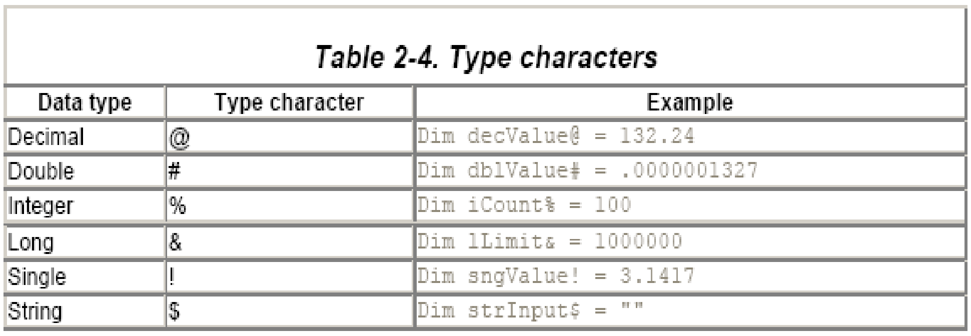


Table 2-4. Type characters

Data type	Type character	Example
Decimal	@	`Dim decValue@ = 132.24`
Double	#	`Dim dblValue# = .0000001327`
Integer	%	`Dim iCount% = 100`
Long	&	`Dim lLimit& = 1000000`
Single	!	`Dim sngValue! = 3.1417`
String	$	`Dim strInput$ = ""`

As a matter of style, type characters should be avoided in preference to spelling out type names and using descriptive variable names.

2.9.1 Variable Initializers

New to Visual Basic .NET is the ability to combine variable declaration and assignment. For example, this code declares an Integer i and gives it an initial value of 10:

```
Dim i As Integer = 10
```

This is equivalent to the following code:

```
Dim i As Integer
i = 10
```

2.5 - 2.9 Check Your Progress
True or False

1. The object type is the base type from which all other types are derived.
2. Byte type can hold a range of integers from 0 through 255.
3. A collection is any type hat exposes the ICollection interface.
4. A conversion that happens automatically is called an explicit conversion.
5. The keyword Dim indicates a variable declaration.

2.10 SCOPE

Scope refers to the so-called *visibility* of identifiers within source code. That is, given a particular identifier declaration, the *scope* of the identifier determines where it is legal to reference that identifier in code. For example, these two functions each declare a variable CoffeeBreaks. Each declaration is invisible to the code in the other method. The scope of each variable is the method in which it is declared.

```
    Public Sub MyFirstMethod( )
        Dim CoffeeBreaks As Integer
    ' ...
    End Sub
    Public Sub MySecondMethod( )
        Dim CoffeeBreaks As Long
    ' ...
    End Sub
```

Unlike previous versions of Visual Basic, Visual Basic .NET has *block scope*. Variables declared within a set of statements ending with End, Loop, or Next are local to that block. For example:

```
   Dim i As Integer
      For i = 1 To 100
         Dim j As Integer
         For j = 1 To 100
' ...
         Next
      Next
' j is not visible here
```

Visual Basic .NET doesn't permit the same variable name to be declared at both the method level and the block level. Further, the life of the block-level variable is equal to the life of the method. This means that if the block is re-entered, the variable may contain an old value (don't count on this behavior, as it is not guaranteed and is the kind of thing that might change in future versions of Visual Basic).

2.11 ACCESS MODIFIERS

Access modifiers control the accessibility of types (including enumerations, structures, classes, standard modules, and delegates) and type members (including methods, constructors, events, constants, fields [data members], and properties) to other program elements. They are part of the declarations of types and type members. In the following code fragment, for example, the keywords

```
   Public and Private are access modifiers:
   Public Class SomeClass
      Public Sub DoSomething( )
      ' ...
      End Sub
      Private Sub InternalHelperSub( )
      ' ...
      End Sub
   End Class
```

The complete list of access modifiers and their meanings is shown in Table 2-5.

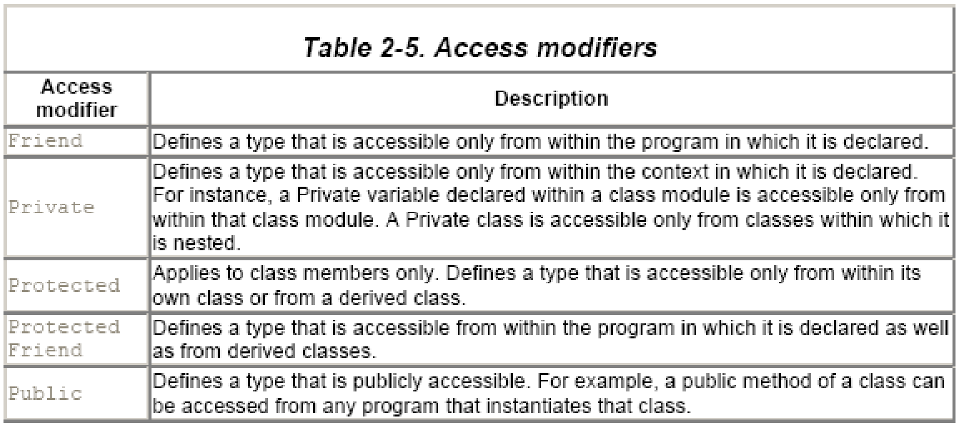


Table 2-5. Access modifiers

Access modifier	Description
`Friend`	Defines a type that is accessible only from within the program in which it is declared.
`Private`	Defines a type that is accessible only from within the context in which it is declared. For instance, a Private variable declared within a class module is accessible only from within that class module. A Private class is accessible only from classes within which it is nested.
`Protected`	Applies to class members only. Defines a type that is accessible only from within its own class or from a derived class.
`Protected Friend`	Defines a type that is accessible from within the program in which it is declared as well as from derived classes.
`Public`	Defines a type that is publicly accessible. For example, a public method of a class can be accessed from any program that instantiates that class.

2.12 ASSIGNMENT

In Visual Basic .NET, assignment statements are of the form:
variable, field, or property = expression Either the type of the expression must be the same as that of the item receiving the assignment, or there must exist an appropriate implicit or explicit conversion from the type of the expression to the type of the item receiving the assignment. For information on implicit and explicit conversions, see Section 2.5.5 earlier in this chapter. When an assignment is made to a value type, the value of the expression is copied to the target. In contrast, when an assignment is made to a reference type, a reference to the value is stored in the target. This is an

important distinction that is worth understanding well.Consider the code in Example 2-3.

Example 2-3. Value-type assignment versus reference-type assignment

```
Public Structure SomeStructure
    Public MyPublicMember As String
End Structure
Public Class SomeClass
    Public MyPublicMember As String
End Class
Public Class AssignmentTest
    Public Shared Sub TestValueAndReferenceAssignment( )
    Dim a, b As SomeStructure
    Dim c, d As SomeClass
    ' Test assignment to value type.
    a.MyPublicMember = "To be copied to 'b'"
    b = a
    a.MyPublicMember = "New value for 'a'"
    Console.WriteLine("The value of b.MyPublicMember is """ _
    & b.MyPublicMember & """")
    ' Test assignment to reference type.
    c = New SomeClass( )
    c.MyPublicMember = "To be copied to d'"
    d = c
    c.MyPublicMember = "New value for 'c'"
    Console.WriteLine("The value of d.MyPublicMember is """ _
    & d.MyPublicMember & """")
End Sub
End Class
```

The output of the TestValueAndReferenceAssignment method in Example 2-3 is:

```
The value of b.MyPublicMember is "To be copied to 'b'"
The value of d.MyPublicMember is "New value for 'c'"
```

In Example 2-3, the SomeStructure structure and the SomeClass class have identical definitions, except that one is a structure and the other is a class. This leads to very different behavior during assignment. When a value type is copied, the actual value is copied. When a reference type is copied, only the reference is copied, resulting in two references to the same value. If the value is subsequently changed through one of the references, the new value is also seen through the other reference.This difference is shown in the output from Example 2-3. The value type in variable a is copied to variable b. The value of a.MyPublicMember is then modified. Subsequently, the call to Console.WriteLine shows that this modification does not affect b.MyPublicMember. In contrast, the assignment of c to d copies only a reference, which means that after the assignment, both c and d reference the same object. The value of c.MyPublicMember is then modified. The subsequent call to Console.WriteLine shows that this modification *did* affect d.MyPublicMember. Indeed, d.MyPublicMember refers to the same memory as c.MyPublicMember.

2.13 OPERATORS AND EXPRESSIONS

Operators are symbols (characters or keywords) that specify operations to be performed on one or two *operands* (or *arguments*). Operators that take one operand are called *unary operators*. Operators that take two operands are called *binary operators*. Unary operators use *prefix notation*, meaning that the operator precedes the operand (e.g., -5). Binary operators (except for one case) use *infix notation*, meaning that the operator is between the operands (e.g., 1 + 2). The TypeOf...Is operator is a binary operator that uses a special form that is neither prefix nor infix notation.

2.13.1 Unary Operators

Visual Basic supports the following unary operators:
+ (unary plus)

The unary plus operator takes any numeric operand. It's not of much practical use because the value of the operation is equal to the value of the operand.
- (unary minus)
The unary minus operator takes any numeric operand (except as noted later). The value of the operation is the negative of the value of the operand. In other words, the result is calculated by subtracting the operand from zero. If the operand type is Short, Integer, or Long, and the value of the operand is the maximum negative value for that type, then applying the unary minus operator will cause a System.OverflowException error, as in the following code fragment:

```
Dim sh As Short = -32768
Dim i As Integer = -sh
```

Not (logical negation)
The logical negation operator takes a Boolean operand. The result is the logical negation of the operand. That is, if the operand is False, the result of the operation is True, and vice versa.

AddressOf
The AddressOf operator returns a reference to a method. Two different kinds of references can be obtained, depending on the context in which the operator is used:

- ❖ When the AddressOf operator is used within the argument list of a call to a method, which is made available via the Declare statement, it returns a function pointer that is suitable for such calls.
- ❖ When the AddressOf operator is used in any other context, a delegate object is returned. See Section 2.19 later in this chapter for information.

2.13.2 Arithmetic Operators

The arithmetic operators perform the standard arithmetic operations on numeric values. The arithmetic operators supported by Visual Basic .NET are:
* (multiplication)
The multiplication operator is defined for all numeric operands. The result is the product of the operands.
/ (regular division)
The regular division operator is defined for all numeric operands. The result is the value of the first operand divided by the second operand.
\ (integer division)
The integer division operator is defined for integer operands (Byte, Short, Integer, and Long).
The result is the value of the first operand divided by the second operand, then rounded to the integer nearest to zero.
Mod (modulo)
The modulo operator is defined for integer operands (Byte, Short, Integer, and Long). The result is the remainder after the integer division of the operands.
^ (exponentiation)
The exponentiation operator is defined for operands of type Double. Operands of other numeric types are converted to type Double before the result is calculated. The result is the value of the first operand raised to the power of the second operand.+ (addition)
The addition operator is defined for all numeric operands and operands of an enumerated type.The result is the sum of the operands. For enumerated types, the sum is calculated on the underlying type, but the return type is the enumerated type. See the discussion of enumerated types in the "Enumerations" section later in this chapter for more information on the types that can underlie an enumerated type. See also Section 2.12.4 later in this section.
- (subtraction)
The subtraction operator is defined for all numeric operands and operands of an enumerated type. The result is the value of the first operand minus the second operand. For enumerated types, the subtraction is calculated on the underlying type, but the return type is the enumerated type. See the discussion of enumerated types in Section 2.17 later in this chapter for more information on the types that can underlie an enumerated type.

2.13.3 Relational Operators

The relational operators all perform some comparison between two operands and return a Boolean value indicating whether the operands satisfy the comparison. The

relational operators supported by Visual Basic .NET are:= (equality) The equality operator is defined for all primitive value types and all reference types. For primitive value types and for the String type, the result is True if the values of the operands are equal; False if not. For reference types other than String, the result is True if the references refer to the same object; False if not. If the operands are of type Object and they reference primitive value types, value comparison is performed rather than reference comparison.
<> (inequality)
The inequality operator is defined for all primitive value types and for reference types. For primitive value types and for the String type, the result is True if the values of the operands are not equal; False if equal. For reference types other than String, the result is True if the references refer to different objects; False if they refer to the same object. If the operands are of type Object and they reference primitive value types, value comparison is performed rather than reference comparison.
< (less than)
The less-than operator is defined for all numeric operands and operands of an enumerated type. The result is True if the first operand is less than the second; False if not. For enumerated types, the comparison is performed on the underlying type.
> (greater than)
The greater-than operator is defined for all numeric operands and operands that are of an enumerated type. The result is True if the first operand is greater than the second; False if not. For enumerated types, the comparison is performed on the underlying type.
<= (less than or equal to)
The less-than-or-equal-to operator is defined for all numeric operands and operands of an enumerated type. The result is True if the first operand is less than or equal to the second operand; False if not.
>= (greater than or equal to)The greater-than-or-equal-to operator is defined for all numeric operands and operands of an enumerated type. The result is True if the first operand is greater than or equal to the second operand; False if not.
TypeOf...Is
The TypeOf...Is operator is defined to take a reference as its first parameter and the name of a type as its second parameter. The result is True if the reference refers to an object that is type-compatible with the given type-name; False if the reference is Nothing or if it refers to an object that is not type-compatible with the given type name. Use the TypeOf...Is operator to determine whether a given object:

- Is an instance of a given class
- Is an instance of a class that is derived from a given class
- Exposes a given interface

In any of these cases, the TypeOf expression returns True.
Is The Is operator is defined for all reference types. The result is True if the references refer to the same object; False if not. Like The Like operator is defined only for operands of type String. The result is True if the first operand matches the pattern given in the second operand; False if not.The rules for matching are:

- The ? (question mark) character matches any single character.

- The * (asterisk) character matches zero or more characters.

- The # (number sign) character matches any single digit.

- A sequence of characters within [] (square brackets) matches any single character inthe sequence. Within such a bracketed list, two characters separated by a - (hyphen) signify a range of Unicode characters, starting with the first character and ending with the second character. A - character itself can be matched by placing it at the beginning or end of the bracketed sequence. Preceding the sequence of characters with an ! (exclamation mark) character matches any single character that does not appear in the sequence.

- The ?, *, #, and [characters can be matched by placing them within [] in the pattern string. Consequently, they cannot be used in their wildcard sense within [].

- The] character does not need to be escaped to be explicitly matched. However, it can't be used within [].

2.13.4 String-Concatenation Operators

The & (ampersand) and + (plus) characters signify string concatenation. String concatenation is defined for operands of type String only. The result is a string that consists of the characters from the first operand followed by the characters from the second operand.

2.13.5 Bitwise Operators

It is sometimes necessary to manipulate the individual bits that make up a value of one of the integer types (Byte, Short, Integer, and Long). This is the purpose of the bitwise operators. They are defined for the four integer types and for enumerated types. When the bitwise operators are applied to enumerated types, the operation is done on the underlying type, but the result is of the enumerated type. The bitwise operators work by applying the given Boolean operation to each of the corresponding bits in the two operands. For example, consider this expression: 37 And 148 To calculate the value of this expression, consider the binary representation of each operand. It's
helpful to write one above the other so that the bit columns line up:

```
00100101 (37)
10010100 (148)
Next, apply the Boolean And operation to the bits in each column:
00100101 (37)
And 10010100 (148)
--------
00000100 (4)
37 And 148, therefore, equals 4.
```

The bitwise operators are:

And

Performs a Boolean And operation on the bits. (The result bit is 1 if and only if both of the source bits are 1.) AndAlso The result is True if and only if both the operands are True; otherwise, the result is False. AndAlso performs logical short-circuiting: if the first operand of the expression is False, the second operand is not evaluated.

Or

Performs a Boolean Or operation on the bits. (The result bit is 1 if either or both of the source bits are 1.)

OrElse

The result is True if either or both the operands is True; otherwise, the result is False.
OrElse performs logical short-circuiting: if the first operand of the expression is True, the second operand is not evaluated.

Xor

Performs a Boolean *exclusive or* operation on the bits. (The result bit is 1 if either of the source bits is 1, but not both.)

Not

Performs a Boolean Not operation on the bits in the operand. This is a unary operator. (The result is 1 if the source bit is 0 and 0 if the source bit is 1.)

2.13.6 Logical Operators

Logical operators are operators that require Boolean operands. They are: And The result is True if and only if both of the operands are True; otherwise, the result is False.

Or

The result is True if either or both of the operands is True; otherwise, the result is False.

Xor

The result is True if one and only one of the operands is True; otherwise, the result is False.

Not

This is a unary operator. The result is True if the operand is False; False if the operand is True.

2.13.7 Operator Precedence

Operator precedence defines the order in which operators are evaluated. For example, the expression 1 + 2 * 3 has the value 9 if the addition is performed first but has the value 7 if the multiplication is performed first. To avoid such ambiguity, languages must define the order in which operations are evaluated. Visual Basic .NET divides the operators into groups and defines each group's precedence relative to the others. Operators in higher-precedence groups are evaluated before operators in lowerprecedence groups. Operators within each group have the same precedence relative to each other.When an expression contains multiple operators from a single group, those operators are evaluated from left to right. Table 2-6 shows Visual Basic .NET's operators, grouped by precedence from highest to lowest order of evaluation.

Table 2-6. The precedence of Visual Basic .NET's operators **Category Operator**

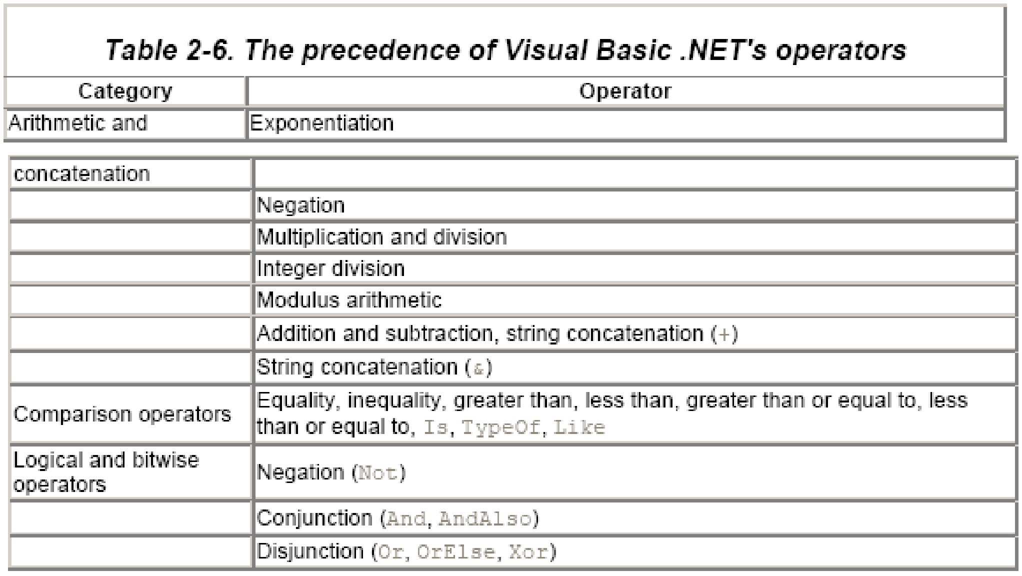


Table 2-6. The precedence of Visual Basic .NET's operators

Category	Operator
Arithmetic and	Exponentiation
concatenation	
	Negation
	Multiplication and division
	Integer division
	Modulus arithmetic
	Addition and subtraction, string concatenation (`+`)
	String concatenation (`&`)
Comparison operators	Equality, inequality, greater than, less than, greater than or equal to, less than or equal to, `Is`, `TypeOf`, `Like`
Logical and bitwise operators	Negation (`Not`)
	Conjunction (`And`, `AndAlso`)
	Disjunction (`Or`, `OrElse`, `Xor`)

Parentheses override the default order of evaluation. For example, in the expression 1 + 2 * 3, the multiplication is performed before the addition, yielding a value of 7. To perform the addition first, the expression can be rewritten as (1 + 2) * 3, yielding a result of 9.

2.13.8 Operator Overloading

Operator overloading is a feature that some languages (C#, for example) provide to allow developers to specify how the built-in operators (+, -, *, /, =, etc.) should behave when applied to programmerdefined types. For example, the developer of a type representing complex numbers could use operator overloading to specify appropriate functionality for the built-in arithmetic operators when applied to operands of the custom type. The .NET Framework supports operator overloading, but .NET languages are not required to do so. The current version of Visual Basic .NET doesn't support operator overloading, although there's no reason that Microsoft couldn't add it in the future. Components that are written in other languages may overload operators, but Visual Basic .NET will not be aware of the overloads. Well-designed components provide an alternative mechanism for accessing the functionality provided by the overloads. For example, if a component written in C# provides a class that overloads the + operator, it should also provide a method that takes two parameters and returns their sum. Thus, what would be written as:

```
c = a + b
```

in a language that supports overloading would be written as:

```
c = MyCustomType.Add(a, b)
```

in Visual Basic .NET. The name of the actual method would depend on the component's implementer.

2.10 - 2.13 Check Your Progress

Fill in the blanks

1. Scope refers to the ………….. within source code.
2. ……………. control the accessibility of types.
3. ………….. defines a types that is accessible only from within the program in Which it is declared.
4. The ………….. operator is to test the type compatibility of two object reference variables with various data types
5. ………………….. is operator is a binary operator that uses a special form that is neither prefix nor infix notation.

2.14 STATEMENTS

Visual Basic .NET is a line-oriented language, in which line breaks generally indicate the ends of statements. However, there are times when a programmer may wish to extend a statement over several lines or have more than one statement on a single line.To extend a statement over several lines, use the line-continuation character, an underscore (_). It must be the last character on its line, and it must be immediately preceded by a space character.
Lines connected in this way become a single logical line. Here is an example:

```
Dim strSql As String = "SELECT Customers.CompanyName," _
& " COUNT(Orders.OrderID) AS OrderCount" _
& " FROM Customers INNER JOIN Orders" _
& " ON Customers.CustomerID = Orders.CustomerID" _
& " GROUP BY Customers.CompanyName" _
& " ORDER BY OrderCount DESC"
```

A line break can occur only where whitespace is allowed.

To place two or more statements on a single line, use the colon (:) between the statements, like this:

```
i = 5 : j = 10
```

The remainder of this section discusses the statements in Visual Basic .NET.

2.14.1 Option Statements

There are three Option statements, which affect the behavior of the compiler. If used, they must appear before any declarations in the same source file. They control the compilation of the source code in the file in which they appear. They are:
Option Compare
The Option Compare statement controls the manner in which strings are compared to each other. The syntax is:

```
Option Compare [ Binary | Text ]
```

If Binary is specified, strings are compared based on their internal binary representation (i.e., string comparisons are case-sensitive). If Text is specified, strings are compared based on case-insensitive alphabetical order. The default is Binary.

Option Explicit

The Option Explicit statement determines whether the compiler requires all variables to be explicitly declared. The syntax is:

```
Option Explicit [ On | Off ]
```

If On is specified, the compiler requires all variables to be declared. If Off is specified, the compiler considers a variable's use to be an implicit declaration. It is considered good programming practice to require declaration of variables. The default is On.

Option Strict

The Option Strict statement controls the implicit type conversions that the compiler will allow. The syntax is:

```
Option Strict [ On | Off ]
```

If On is specified, the compiler only allows implicit widening conversions; narrowing conversions must be explicit. If Off is specified, the compiler allows implicit narrowing conversions as well. This could result in runtime exceptions not foreseen by the developer. It is considered good programming practice to require strict type checking. The default is Off. See Section 2.5.5 earlier in this chapter for the definitions of widening and narrowing conversions.

2.14.2 Branching Statements

Visual Basic .NET supports a number of branching statements that interrupt the sequential flow of program execution and instead allow it to jump from one portion of a program to another. These can be either conditional statements (such as If or Select Case) or unconditional (such as Call and Exit).

2.14.2.1 Call

The Call statement invokes a subroutine or function. For example:
Call *SomeMethod( )*When the invoked subroutine or function finishes, execution continues with the statement following the Call statement. If a function is invoked, the function's return value is discarded.
The Call statement is redundant because subroutines and functions can be invoked simply by naming them:

```
SomeMethod( )
```

2.14.2.2 Exit

The Exit statement causes execution to exit the block in which the Exit statement appears. It is generally used to prematurely break out of a loop or procedure when some unusual condition occurs. The Exit statement should be avoided when possible because it undermines the structure of the block in which it appears. For example, the exit conditions of a For loop should be immediately apparent simply by looking at the For statement. It should not be necessary to read through the entire loop to determine if there are additional circumstances under which the loop might exit. If a given For loop truly needs an Exit statement, investigate whether a different loop construct would be better suited to the task. If a given procedure truly needs an Exit statement, investigate whether the procedure is factored appropriately. The Exit statement has a different form for each type of block in which it can be used, as listed here:

Exit Do
Exits a Do loop. Execution continues with the first statement following the Loop statement.

Exit For

Exits a For loop. Execution continues with the first statement following the Next statement.

Exit Function
Exits a function. Execution continues with the first statement following the statement that called the function.

Exit Property
Exits a property get or property set procedure. Execution continues with the first statement following the statement that invoked the property get or property set procedure.

Exit Sub
Exits a subroutine. Execution continues with the first statement following the statement that called the subroutine.

Exit Try
Exits the Try clause of a Try block. If the Try block has a Finally clause, execution continues with the first statement in the Finally clause. If the Try block does not have a Finally clause, execution continues with the first statement following the Try block.

2.14.2.3 Goto

The Goto statement transfers execution to the first statement following the specified label. For example:

```
' ...
Goto MyLabel
' ...
MyLabel:
' ...
```

The label must be in the same procedure as the Goto statement.The Goto statement is generally avoided in structured programming because it often leads to code that is difficult to read and debug.

2.14.2.4 If

The If statement controls whether a block of code is executed based on some condition. The simplest form of the If statement is:

If *expression* Then *statements*

End If

expression is any expression that can be interpreted as a Boolean value. If *expression* is True, the statements within the If block are executed. If *expression* is False, those statements are skipped. To provide an alternative set of statements to execute when *expression* is False, add an Else clause, as shown here:

If *expression* Then
statements
Else
statements
End If

If *expression* is True, only the statements in the If clause are executed. If *expression* is False, only the statements in the Else clause are executed. Finally, a sequence of expressions can be evaluated by including one or more ElseIf clauses, as shown here:

If *expression* Then
    statements
ElseIf *expression* Then
    statements
ElseIf *expression* Then

```
    statements
Else
    statements
End If
```

The first If or ElseIf clause whose expression evaluates to True will have its statements executed.Statements in subsequent ElseIf clauses will not be executed, even if their corresponding expressions are also True. If none of the expressions evaluate to True, the statements in the Else clause will be executed. The Else clause can be omitted if desired.

2.14.2.5 RaiseEvent

The RaiseEvent statement fires the given event. After the event has been fired to all listeners, execution continues with the first statement following the RaiseEvent statement. See Section 2.20
later in this chapter for more information.

2.14.2.6 Return

The Return statement exits a function and provides a return value to the caller of the function.Execution continues with the first statement following the statement that called the function. Here is an example:

```
Public Shared Function MyFactorial(ByVal value As Integer) As Integer
    Dim retval As Integer = 1
    Dim i As Integer
    For i = 2 To value
        retval *= i
    Next
Return retval
End Function
```

Another way to return a value to the caller of the function is to assign the value to the function name and then simply drop out of the bottom of the function. This is how it was done in Visual Basic 6 (and can still be done in Visual Basic .NET). Here is an example:

```
Public Shared Function MyFactorial(ByVal value As Integer) As Integer
    Dim retval As Integer = 1
    Dim i As Integer
    For i = 2 To value
        retval *= i
    Next
MyFactorial = retval
End Function
```

In Visual Basic 6, the Return statement was used to return
execution to the statement following a GoSub statement. In Visual
Basic .NET, the GoSub statement no longer exists, and the Return statement is now used as described here.

2.14.2.7 Select Case

The Select Case statement chooses a block of statements to execute based on some value. For example:

```
Select Case strColor
Case "red"
' ...
Case "green"
' ...
Case "blue"
' ...
Case "yellow"
' ...
Case Else
' ...
End Select
```

If strColor in this example contains "blue", only the statements in the Case "blue" clause are executed. If none of the Case clauses matches the value in the Select Case statement, the statements in the Case Else clause are executed. If more than one Case clause matches the given value, only the statements in the first matching Case clause are executed. Case statements can include multiple values to be matched against the value given in the Select Case statement. For example:

```
Case "red", "green", "blue", strSomeColor
```

This case will be matched if the value in the Select Case statement is "red", "green", "blue", or the value contained in strSomeColor. The To keyword can be used to match a range of values, as shown here:

```
Case "apples" To "oranges"
```

This Case statement matches any string value that falls alphabetically within this range.The Is keyword can be used for matching an open-ended range:

```
Case Is > "oranges"
```

Don't confuse this use of the Is keyword with the Is comparison operator.

2.14.3 Iteration Statements

Iteration statements, also known as *looping* statements, allow a group of statements to be executed more than once. The group of statements is known as the *body* of the loop. Three statements fall under this category in Visual Basic .NET are:
Do, For, and For Each.

2.14.3.1 Do

The Do loop executes a block of statements either until a condition becomes true or while a condition remains true. The condition can be tested at the beginning or at the end of each iteration. If the test is performed at the end of each iteration, the block of statements is guaranteed to execute at least once. The Do loop can also be written without any conditions, in which case it executes repeatedly until and unless an Exit Do statement is executed within the body of the loop. Here are some examples of Do loops:

```
Do While i < 10
' ...
Loop
Do Until i >= 10
' ...
Loop
Do
' ...
Loop While i < 10
Do
' ...
Loop Until i >= 10
Do
' ...
Loop
```

2.14.3.2 For

The For loop executes a block of statements a specified number of times. The number of iterations is controlled by a loop variable, which is initialized to a certain value by the For statement, then is incremented for each iteration of the loop. The statements in the body of the loop are repeatedly executed until the loop variable exceeds a given upper bound.
The syntax of the For loop is:

For *variable* = *expression* To *expression* [Step *expression*]

```
statements
Next [ variable_list ]
```

The loop variable can be of any numeric type. The variable is set equal to the value of the first expression before entering the first iteration of the loop body. Prior to executing each iteration of the loop, the loop variable is compared with the value of the second expression. If the value of the loop variable is greater than the expression (or less than the expression if the step expression is negative), the loop exits and execution continues with the first statement following the Next statement. The step expression is a numeric value that is added to the loop variable between loop iterations. If the Step clause is omitted, the step expression is taken to be 1. The Next statement marks the end of the loop body. The Next keyword can either appear by itself in the statement or be followed by the name of the loop variable. If For statements are nested, a single Next statement can terminate the bodies of multiple loops. For example:

```
For i = 1 To 10
For j = 1 To 10
For k = 1 To 10
' ...
Next k, j, I
```

This code is equivalent to the following:

```
For i = 1 To 10
For j = 1 To 10
For k = 1 To 10
' ...
Next
Next
Next
```

I recommend the latter style, since it is considered more structured to terminate each block explicitly.It is interesting to note that the For loop is equivalent to the following Do loop construction (assuming that *step_expression* is nonnegative):

```
loop_variable = from_expression
Do While loop_variable <= to_expression
Statements
loop_variable += step_expression
Loop
```

If *step_expression* is negative, the For loop is equivalent to this (only the comparison in the Do statement is different):

```
loop_variable = from_expression
Do While loop_variable >= to_expression
statements
loop_variable += step_expression
Loop
```

2.14.3.3 For Each

The For Each statement is similar to the For statement, except that the loop variable need not be numeric, and successive iterations do not increment the loop variable. Instead, the loop variable takes successive values from a collection of values. Here is the syntax:

```
For Each variable In expression statements
Next [ variable ]
```

The loop variable can be of any type. The expression must be a reference to an object that exposes the IEnumerable interface (interfaces are discussed later in this chapter). Generally, types that are considered collections expose this interface. The .NET Framework class library provides several useful collection types, which are listed in Chapter 3. (See Section 2.5.4 earlier in this chapter for an explanation of what constitutes a collection type.) The type of the items in the collection must be compatible with the type of the loop variable. The statements in the body of the loop execute once for each item in the collection. During each iteration, the loop variable is set equal to each consecutive item in the collection. Because all Visual Basic .NET arrays expose the IEnumerable interface, the For Each statement can be used to iterate through the elements of an array. For example:

```
Dim a( ) As Integer = {1, 2, 3, 4, 5}
Dim b As Integer
For Each b In a
Console.WriteLine(b)
Next
This is equivalent to the following code:
Dim a( ) As Integer = {1, 2, 3, 4, 5}
Dim b As Integer
Dim i As Integer
For i = a.GetLowerBound(0) To a.GetUpperBound(0)
b = a(i)
Console.WriteLine(b)
Next
```

Because all arrays in Visual Basic .NET implicitly derive from the Array type (in the System namespace), the a array in this example has access to methods defined on the Array type (specifically GetLowerBound and GetUpperBound).In case you're interested, here is the equivalent code using a Do loop. This is essentially what the For Each statement is doing under the covers, although the For Each construct is likely to compile to faster code.

```
Dim a( ) As Integer = {1, 2, 3, 4, 5}
Dim b As Integer
Dim e As Object = a.GetEnumerator( )
Do While CType(e.GetType( ).InvokeMember("MoveNext", _
Reflection.BindingFlags.InvokeMethod, Nothing, e, Nothing), Boolean)
b = CType(e.GetType( ).InvokeMember("Current",
_Reflection.BindingFlags.GetProperty,  Nothing, e, Nothing), Integer)
Console.WriteLine(b)
Loop
```

2.14.4 Mathematical Functions

Mathematical functions are provided through the Math class (defined in the System namespace). The Math class constants and methods are listed in Appendix E.

2.14.5 Input/Output

File and Internet I/O features are provided by the .NET Framework class library and will be briefly touched on in Chapter 3. In addition, Visual Basic .NET provides its own class library that includes functions for opening, reading, and closing files. File access and network-protocol programming are not discussed in this book. Instead, preference is given to the much more common tasks of database access and web-service programming.

2.15 CLASSES

A *class* is one form of data type. As such, a class can be used in contexts where types are expected— in variable declarations, for example. In object-oriented design, classes are intended to represent the definition of real-world objects, such as *customer*, *order*, *product*, etc. The class is only the definition, not an object itself. An object would be *a* customer, *an* order, or *a* product. A class declaration defines the set of members—fields, properties, methods, and events—that each object of that class possesses. Together, these members define an object's state, as well as its functionality. An object is also referred to as an *instance* of a class. Creating an object of a certain class is called *instantiating* an object of the class.
Consider the class definition in Example 2-4.

Example 2-4. A class definition

```
Public Class Employee
Public EmployeeNumber As Integer
Public FamilyName As String
Public GivenName As String
Public DateOfBirth As Date
Public Salary As Decimal
Public Function Format( ) As String
Return GivenName & " " & FamilyName
End Function
End Class
```

The code in Example 2-4 defines a class called Employee. It has five public *fields* (also known as *data members*) for storing state, as well as one *member function*. The class could be used as shown in Example 2-5.

Example 2-5. Using a class

```
Dim emp As New Employee( )
emp.EmployeeNumber = 10
emp.FamilyName = "Rodriguez"
emp.GivenName = "Celia"
emp.DateOfBirth = #1/28/1965#
emp.Salary = 115000
Console.WriteLine("Employee Name: " & emp.Format( ))
Console.WriteLine("Employee Number: " & emp.EmployeeNumber)
Console.WriteLine("Date of Birth: " & emp.DateOfBirth.ToString("D",
Nothing))
Console.WriteLine("Salary: " & emp.Salary.ToString("C", Nothing))
```

The resulting output is:

```
Employee Name: Celia Rodriguez
Employee Number: 10
Date of Birth: Thursday, January 28, 1965
Salary: $115,000.00
```

2.15.1 Object Instantiation and New

Object instantiation is done using the New keyword. The New keyword is, in effect, a unary operator that takes a type identifier as its operand. The result of the operation is a reference to a newly created object of the given type. Consider the following:

```
Imports System.Collections
' ...
Dim ht As Hashtable
ht = New Hashtable( )
```

The Dim statement declares a variable that is capable of holding a reference to an object of type Hashtable, but it doesn't actually create the object. The code in the line following the Dim statement instantiates an object of type Hashtable and assigns to the variable a reference to the newly created object. As with any other variable declaration, the assignment can be done on the same line as the declaration, as shown here:

```
Imports System.Collections
' ...
    Dim ht As Hashtable = New Hashtable( )
```

Visual Basic .NET permits a typing shortcut that produces the same result:

```
    Imports System.Collections
    ' ...
    Dim ht As New Hashtable( )
```

2.15.2 Constructors

When a class is instantiated, some initialization often must be performed before the type can be used.To provide such initialization, a class may define a *constructor*. A constructor is a special kind of method. It is automatically run whenever an object of the class is instantiated. Constructor declarations use the same syntax as regular method declarations, except that in place of a method name, the constructor declaration uses the keyword New. For example:

```
Public Class SomeClass
Public Sub New( )
' Do any necessary initialization of the object here.
End Sub
End Class
```

To invoke the constructor, a new object must be instantiated:

```
Dim obj As New SomeClass( )
```

Note the parentheses (( )) following the name of the class. Until you get used to it, this method-style syntax following a class name may appear odd. However, the empty parentheses indicate that the class's constructor takes no arguments. Constructors can take arguments, if they are necessary for the initialization of the object:

```
Public Class SomeClass
Dim m_value As Integer
Public Sub New(ByVal InitialValue As Integer)
m_value = InitialValue
End Sub
End Class
```

When objects of this class are instantiated, a value must be provided for the constructor's argument:

```
Dim obj As New SomeClass(27)
```

Constructors can be overloaded, if desired. For example:

```
Public Class SomeClass
Dim m_value As Integer
Public Sub New( )
m_value = Date.Today.Day ' for example
End Sub
Public Sub New(ByVal InitialValue As Integer)
m_value = InitialValue
End Sub
End Class
```

The constructor that is called depends on the arguments that are provided when the class is instantiated, as shown here:

```
Dim obj1 As New SomeClass( ) ' calls parameterless constructor
Dim obj2 As New SomeClass(100) ' calls parameterized constructor
```

Constructors are usually marked Public. However, there are times when it may be desirable to mark a constructor as Protected or Private. Protected access prohibits the class from being instantiated by any class other than a class derived from this class. Private access prohibits the class from being instantiated by any code other than its own. For example, a particular class design might require that the class itself be in control of whether and when instances are created. Example 2-6 shows a class that implements a crude form of object pooling.

Example 2-6. Using a private constructor

```
Imports System.Collections
' ...
Public Class MyPooledClass
' This shared field keeps track of instances that can be handed out.
Private Shared m_pool As New Stack( )
' This shared method hands out instances.
Public Shared Function GetInstance( ) As MyPooledClass
If m_pool.Count > 0 Then
' We have one or more objects in the pool. Remove one from the
' pool and give it to the caller.
Return CType(m_pool.Pop( ), MyPooledClass)
Else
```

```
' We don't have any objects in the pool. Create a new one.
Return New MyPooledClass( )
End If
End Function
' This method must be called to signify that the client is finished
' with the object.
Public Sub ImDone( )
' Put the object in the pool.
m_pool.Push(Me)
End Sub
' Declaring a private constructor means that the only way to
' instantiate this class is through the GetInstance method.
Private Sub New( )
End Sub
End Class
```

The class in Example 2-6 would be used like this:

```
Dim obj As MyPooledClass = MyPooledClass.GetInstance( )
' ...
obj.ImDone( )
```

Sometimes when constructors are overloaded, it makes sense to implement one constructor in terms of another. For example, here is a class that has a constructor that takes a SqlConnection object as a parameter. However, it also has a parameterless constructor that creates a SqlConnection object and passes it to the class's parameterized constructor. Note the use of the MyClass keyword to access members of the type:

```
Imports System.Data.SqlClient
' ...
Public Class SomeClass
Public Sub New( )
MyClass.New(New SqlConnection( ))
End Sub
Public Sub New(ByVal cn As SqlConnection)
' Do something with the connection object.
End Sub
End Class
```

Similarly, MyBase.New can call a base-class constructor. If this is done, it must be done as the first statement in the derived class's constructor. Note that if no explicit call is made, the compiler creates a call to the base-class constructor's parameterless constructor. Even if the base class exposes a parameterized constructor having the same signature (i.e., the same number and types of parameters) as the derived class's constructor, by default the compiler generates code that calls the base class's *parameterless* constructor. If a class has shared fields that must be initialized before access, and that initialization can't be performed by initializers in the fields' declarations, a shared constructor may be written to initialize the fields, as shown here:

```
Public Class SomeClass
Public Shared SomeStaticField As Integer
Shared Sub New( )
SomeStaticField = Date.Today.Day
End Sub
End Class
```

The shared constructor is guaranteed to run sometime before any members of the type are referenced. If any shared fields have initializers in their declarations, the initializers are assigned to the fields before the shared constructor is run. Shared constructors may not be overloaded, nor may they have access modifiers (Public, Private, etc.). Neither feature is meaningful in the context of shared constructors.

2.15.3 Fields

Fields, also known as *data members*, hold the internal state of an object. Their declarations appear only within class and structure declarations. Field declarations include an *access modifier*, which determines how visible the field is from code outside the containing class definition. Access modifiers were discussed earlier in this chapter, under Section 2.10.The value stored in a field is specific to a particular object instance. Two instances can have different values in their corresponding fields. For example:

```
Dim emp1 As New Employee( )
Dim emp2 As New Employee( )
emp1.EmployeeNumber = 10
emp2.EmployeeNumber = 20 ' Doesn't affect emp1.
```

Sometimes it is desirable to share a single value among all instances of a particular class. Declaring a field using the Shared keyword does this, as shown here:

```
Public Class X
Public Shared a As Integer
End Class
```

Changing the field value through one instance affects what all other instances see. For example:

```
Dim q As New X( )
Dim r As New X( )
q.a = 10
r.a = 20
Console.WriteLine(q.a)
' Writes 20, not 10.
```

Shared fields are also accessible through the class name:

```
Console.WriteLine(X.a)
```

2.15.3.1 Read-only fields

Fields can be declared with the ReadOnly modifier, which signifies that the field's value can be set only in a constructor for the enclosing class. This gives the benefits of a constant when the value of the constant isn't known at compile time or can't be expressed in a constant initializer. Here's an example of a class that has a read-only field initialized in the class's constructor:

```
Public Class MyDataTier
Public ReadOnly ActiveConnection As System.Data.SqlClient.SqlConnection
Public Sub New(ByVal ConnectionString As String)
ActiveConnection = _
New System.Data.SqlClient.SqlConnection(ConnectionString)
End Sub
End Class
```

The ReadOnly modifier applies only to the field itself—not to members of any object referenced by the field. For example, given the previous declaration of the MyDataTier class, the following code is legal:

```
Dim mydata As New MyDataTier(strConnection)
mydata.ActiveConnection.ConnectionString=strSomeOtherConnection
```

2.15.4 Handling Events

When a field is of an object type that exposes events, the field's enclosing class may define methods for handling those events. For an explanation of events, see Section 2.20 later in this chapter. Here is an example:

```
Imports System.Data.SqlClient
Public Class EventHandlingTest
Private WithEvents m_cn As SqlConnection
Public Sub MySqlInfoMessageEventHandler( _ByVal sender As Object, _ByVal e
As SqlInfoMessageEventArgs _)
Handles m_cn.InfoMessage
Dim sqle As SqlError
For Each sqle In e.Errors
Debug.WriteLine(sqle.Message)
Next
End Sub
' ...
```

```
End Class
```

This class has a field, m_cn, that holds a database connection. The field is declared with the WithEvents keyword, so the class is capable of receiving and handling events raised by the Connection object. In order to handle the Connection object's InfoMessage event, the class defines a method having the appropriate parameter list and a Handles clause:

```
Public Sub MySqlInfoMessageEventHandler( _ByVal sender As Object, _ByVal e
As SqlInfoMessageEventArgs _)
Handles m_cn.InfoMessage
```

This declaration signifies that when the InfoMessage event is raised by the object referenced in m_cn, the ySQLInfoMessageEventHandler method should be called to handle it. The body of the event handler in this case simply outputs the messages received from SQL Server.

2.15.5 Inheritance

Inheritance is one way to reuse and extend previously written code. A program's design often requires several classes as variations of a common theme. Consider a drawing program that deals with many shapes. Such a program would probably define a class for each kind of shape. However, there would be much in common among such classes, including many of their fields, methods, and events. Inheritance allows these common features to be extracted into a *base class* from which the various specific shape classes are *derived*. Example 2-7 shows a base class called Shape, two utility classes used by Shape (Point and Extent), and two classes derived from Shape (Circle and Square)
.

Example 2-7. Class inheritance

```
' This structure represents a point on a plane.
Public Structure Point
Public X As Integer
Public Y As Integer
End Structure

' This structure represents a size or offset.
Public Structure Extent
Public XExtent As Integer
Public YExtent As Integer
End Structure

' This class represents the functionality that is common for
' all shapes. This class can't itself be instantiated, because
' of the "MustInherit" modifier.
Public MustInherit Class Shape
' The upper-left corner of the shape.
Public Origin As Point
' The width and height of the shape.
Public Size As Extent
' This forces all derived classes to implement a method
' called Draw. Notice that a method marked with MustInherit
' has no body in the base class.
    Public MustOverride Sub Draw( )
' This subroutine moves a shape.
Public Sub Offset(ByVal Amount As Extent)
Origin.X += Amount.XExtent
Origin.Y += Amount.YExtent
End Sub
' This property allows the class user to find or set the
' center of a shape.
Public Property Center( ) As Point
Get
Dim retval As Point
retval.X = Origin.X + (Size.XExtent \ 2)
retval.Y = Origin.Y + (Size.YExtent \ 2)
Return retval
End Get
Set
Dim currentCenter As Point = Center
Origin.X += Value.X - currentCenter.X
Origin.Y += Value.Y - currentCenter.Y
End Set
End Property
End Class
Public Class Circle
Inherits Shape
Public Overrides Sub Draw( )
' Just a dummy statement for the example.
Console.WriteLine("Circle.Draw( ) was called.")
End Sub
End Class
Public Class Square
Inherits Shape
Public Overrides Sub Draw( )
' Just a dummy statement for the example.
Console.WriteLine("Square.Draw( ) was called.")
End Sub
End Class
```

Note the following:

- ❖ The MustInherit modifier in the Shape class declaration indicates that this class can't be instantiated—it can only be used as a base class in a derivation. In object-oriented design terminology, such a class is known as an *abstract* class.
- ❖ The Circle and Square classes inherit the public members declared in the Shape class.
- ❖ Using the MustOverride modifier on the Draw method declaration in the Shape class forces derived classes to provide an implementation for this method.
- ❖ Constructors aren't inherited. The Ellipse and Rectangle classes therefore declare their own constructors. When no constructor is explicitly provided in a class definition, the compiler automatically creates one. Therefore, all classes have at least one constructor. The autogenerated constructor (also known as the *default constructor*) created by the compiler is the same as if the following code were written in the class definition:

```
Public Sub New( )
MyBase.New( )
End Sub
```

That is, the default constructor simply calls the base class's parameterless constructor. If there is no parameterless constructor on the base class, the compiler generates an error. If a class defines a parameterized constructor, the compiler does not generate a default constructor.Therefore, if both parameterless and parameterized constructors are needed, both must be explicitly written in the class definition. It is possible to define a class from which it is not possible to inherit. This is done with the NotInheritable keyword in the class declaration, as shown here:

```
Public NotInheritable Class SomeClass
' ...
End Class
```

2.15.6 Methods

Methods are members that contain code. They are either subroutines (which don't have a return value) or functions (which do have a return value).
Subroutine definitions look like this:

```
[ method_modifiers ] Sub [ attribute_list ] _
method_name ( [ parameter_list ] ) [ handles_or_implements ]
[ method_body ]
```

```
End Sub
Function definitions look like this:
[ method_modifiers ] Function [ attribute_list ] _
method_name ( [ parameter_list ] ) [ As type_name ] _
[ handles_or_implements ]
[ method_body ]
End Function
```

The elements of method definitions are:

method_modifiers

Keywords that affect the accessibility or use of the method. These include the following:

Access modifiers
Public, Protected, Friend, Protected Friend, or Private, as described in Table 2-5. If no access-modifier keyword is given, Public is assumed.

Override modifiers
Overrides, MustOverride, Overridable, or NotOverridable. See Section 2.14.6.6.

Overloads keyword
Specifies that this method is an overload. See Section 2.14.6.7 later in this section.Shared keyword Specifies that this method does not access object state. That means that the method does not access any nonshared members.

Sub or Function
keyword Specifies whether this method is a subroutine or a function.

attribute_list
An optional list of attributes to be applied to the method. See Section 2.22 later in this chapter.

method_name
The name of the method.

parameter_list
An optional list of formal parameters for the method. See Section 2.14.6.1.

As *type_name*
For functions only, the data type of the value returned from this function. If Option Strict is off, the As *type_name* clause is optional; otherwise, it is required. If it is omitted, the function's return type defaults to Object. Subroutine declarations do not have an As *type_name* clause.

handles_or_implements
Either the Handles keyword followed by a list of events from the enclosing class's data members, or the Implements keyword followed by a list of methods from an interface implemented by the enclosing class. See Section 2.20 and Section 2.15 later in this chapter.

method_body

And then **End Sub or End Function** keywords Indicates the end of the method definition.

2.15.6.1 Method parameters

Methods can be defined to take arguments. As already shown, method definitions can take an optional parameter list. A parameter list looks like this:

parameter { , *parameter* }

That is, a parameter list is one or more parameters separated by commas. Each parameter in the list is of the form:

[Optional] [ParamArray] [ByRef | ByVal] [*attribute_list*] *_parameter_name* [As *type_name*] [= *constant_expression*]

The elements of each parameter declaration are:

Optional keyword

Specifies that an actual argument may be omitted for this parameter in a call to this method. If Optional is specified, the = *constant_expression*, which defines the default value of an omitted argument, must also be specified. Nonoptional parameters can't follow optional parameters in a parameter list. Optional and ParamArray parameters can't appear in the same parameter list.

ParamArray keyword

Specifies that the caller can provide a variable number of arguments. The actual arguments are passed to the method in an array. Only the last parameter in a list may have the ParamArray keyword attached to it. Optional and ParamArray parameters can't appear in the same parameter list. Parameter arrays are discussed later in this section, under Section 2.14.6.3.

ByRef or ByVal keyword

Specifies whether the actual argument will be passed to the method *by reference* or *by value*. When an argument is passed by reference, the address of the argument is passed to the routine; as a result, assignments to the parameter within the method affect the argument in the calling environment. When an argument is passed by value, a copy of the argument is passed to the routine; as a result, assignments to the parameter within the method do not affect the argument in the calling environment. Consider this code:

```
Public Shared Sub TestByRef(ByRef MyParameter As Integer)
MyParameter += 1
End Sub
Public Shared Sub TestByVal(ByVal MyParameter As Integer)
MyParameter += 1
End Sub
Public Shared Sub DoTest( )
Dim x As Integer = 1
TestByRef(x)
Console.WriteLine("x = " & x)
Dim y As Integer = 1
TestByVal(y)
Console.WriteLine("y = " & y)
End Sub
```

The output of the DoTest method is:

```
x = 2
y = 1
```

The TestByRef and TestByVal methods both increment the values of the arguments passed to them by one. However, because the parameter of the TestByRef method is ByRef, the new value is written back to the argument in the caller (in this case, the variable x in the DoTest method). In contrast, the TestByVal method's parameter is ByVal, so the assignment to the parameter doesn't affect the caller.Be aware of the effects of ByRef and ByVal on arguments that are reference types. ByRef means that a reference to the reference is being passed;

ByVal means that the reference itself is being passed. That means that inside the method, the reference could be used to modify the state of the object in the calling environment. For example:

```
Public Class SomeClass
Public a As Integer
End Class
Public Class TestSomeClass
Public Shared Sub TestByRef(ByRef MyParameter As SomeClass)
MyParameter.a += 1
End Sub
```

```
        Public Shared Sub TestByVal(ByVal MyParameter As SomeClass)
        MyParameter.a += 1
        End Sub
        Public Shared Sub DoTest( )
        Dim x As New SomeClass( )
        x.a = 1
        TestByRef(x)
        Console.WriteLine("x.a = " & x.a)
        Dim y As New SomeClass( )
        y.a = 1
        TestByRef(y)
        Console.WriteLine("y.a = " & y.a)
        End Sub
        End Class
```

The output of the DoTest method in this code is:

```
x.a = 2
y.a = 2
```

Observe that even though the variable y is passed by value to the TestByVal method, one of its members nevertheless is updated. In this case, ByVal merely keeps the reference in y from being overwritten by another reference.

attribute_list
Specifies a list of custom attributes to apply to the parameter.

parameter_name
Specifies the name of the parameter.

As *type_name*
Specifies the data type of the parameter. When the method is called, the type of the actual argument must be compatible with the type of the parameter. The As *type_name* element is optional if Option Strict is off; otherwise, it is required. If it is omitted, Object is assumed.

constant_expression
Specifies a constant expression that specifies what value the parameter should take if no actual argument is provided. This is permitted only on optional parameters.

2.15.6.2 Passing arrays as parameters

To declare a parameter as able to receive an array, include parentheses after the parameter name in the declaration. The caller leaves off the parentheses when naming the actual argument. For example:

```
    Public Shared Sub SomeMethod(ByVal x( ) As String)
        Dim str As String
        For Each str In x
        Console.WriteLine(str)
        Next
    End Sub

    Public Shared Sub TestSomeMethod( )
        Dim a(5) As String
        a(0) = "First"
        a(1) = "Second"
        a(2) = "Third"
        a(3) = "Fourth"
        a(4) = "Fifth"
        SomeMethod(a)
    End Sub
```

In the SomeMethod method, parameter x represents an array of String objects. In the TestSomeMethod method, a String array is allocated, its elements are assigned, and the array as a whole is passed to the SomeMethod method, which then prints the array's contents. All array types are reference types. That means that when passing an array as a parameter, only a reference to the array is passed. Because the target method receives a reference to the array, the array elements can be changed by the method, even if the array reference was passed by value. If the array reference is passed by reference, the array reference itself can be changed by the method. For example, the method could allocate a new array and return it through the ByRef parameter, like this:

```
Public Shared Sub DumpArray(ByVal x( ) As String)
    Dim str As String
    For Each str In x
    Console.WriteLine(str)
    Next
    End Sub
    Public Shared Sub CreateNewArray(ByRef x( ) As String)
    Dim newval(7) As String
    newval(0) = "1st"
    newval(1) = "2nd"
    newval(2) = "3rd"
    newval(3) = "4th"
    newval(4) = "5th"
    newval(5) = "6th"
    newval(6) = "7th"
    x = newval
End Sub

Public Shared Sub DoTest( )
    ' Set up a five-element string array and show its contents.
    Dim a(5) As String
    a(0) = "First"
    a(1) = "Second"
    a(2) = "Third"
    a(3) = "Fourth"
    a(4) = "Fifth"
    Console.WriteLine("a( ) before calling the CreateNewArray method:")
    DumpArray(a)
    ' Now pass it to the CreateNewArray method and then show its
    ' new contents.
    CreateNewArray(a)
    Console.WriteLine( )
    Console.WriteLine("a( ) after calling the CreateNewArray method:")
    DumpArray(a)
End Sub
```

In this code, the DoTest method creates a five-element string array and passes it to DumpArray to show the array's contents. The *DoTest* method then calls CreateNewArray, which allocates a new string array—this time with seven elements. It would not be possible, however, to pass back an array with a different number of dimensions, because the parameter is explicitly declared as one-dimensional. Visual Basic .NET considers the dimensionality of an array to be part of its type, but the size of any particular dimension is not part of the array's type.

2.15.6.3 Variable-length parameter lists

Some methods need to take a variable number of arguments. For example, a function to compute the average of the numbers passed to it should accommodate as few or as many numbers as needed.Visual Basic .NET provides this capability through *parameter arrays* . A parameter array is a parameter that to the method looks like an array but to the caller looks like a variable number of parameters. Here is the average-calculation method just mentioned:

```
Public Shared Function Avg(ParamArray ByVal Numbers( ) As Integer) As
Double
    Dim sum As Integer = 0
    Dim count As Integer = 0
    Dim n As Integer
```

```
        For Each n In Numbers
        sum += n
        count += 1
        Next
        Return sum / count
    End Function
```

This method declares only a single parameter—an array of Integers. However, it includes the ParamArray keyword in the declaration, which tells the compiler to allow calls such as this:

```
    ' Compute the average of some numbers.
    Dim d As Double = Avg(31, 41, 59, 26, 53, 58)
```

It's worth noting that an actual array can be passed through the ParamArray parameter—something that wasn't possible in Visual Basic 6. For example:

```
    ' Compute the average of some numbers.
    Dim args( ) As Integer = {31, 41, 59, 26, 53, 58}
    Dim d As Double = Avg(args)
```

2.15.6.4 Main method

When an executable application is compiled, some code must be identified as the startup routine. This portion is what is executed when the application is run. The Visual Basic .NET compiler looks for a method named Main to fulfill this need. In .NET, all code exists as methods within classes, even the Main method. To make it accessible without having to instantiate a class, the Main method must be declared as shared. For example:

```
    Public Class App
        Public Shared Sub Main( )
        ' ...
        End Sub
    End Class
```

The name of the class is not important. At compile time, the Visual Basic .NET compiler looks for a public shared subroutine named Main somewhere in the code. If more than one class has such a method, the developer must specify which one to use by setting the *startup object* in the *Project Properties*. If you're using the command-line compiler, specify the desired startup object with the /main:<*class*> switch.

A program's Main method can also appear within a Visual Basic .NET *module* (not to be confused with .NET modules). Because Visual Basic .NET modules are classes wherein everything is shared, the Shared keyword is not used in such a declaration:

```
    Module App

        Public Sub Main( )
        ' ...
        End Sub
    End Module
```

2.15.6.5 Implementing interface methods

Here we are taking consideration about methods use of interface,details interface will be discussed in next section.

Classes can be declared as implementing one or more interfaces. To implement an interface, the class must expose methods that correspond to the methods defined by the interface. This is done by declaring the methods in the usual way, but with an Implements clause as part of the declaration. Note the Implements keyword added to the declaration of the CompareTo method in this example:

```
    Public Class SomeClass
        Implements IComparablePublic Function CompareTo( _ByVal obj As Object _
        ) As Integer Implements IComparable.CompareTo
    ' ...
    End Function
End Class
```

Note: When appearing in a method declaration, the Implements keyword must be followed by the name of the interface and method that the given method implements. The class's method must have the same signature and return type as the interface's method, but they need not have the same name.

2.15.6.6 Overriding inherited methods

Now we discuss how a base class can be written such that it forces derived classes to implement certain methods. In this case, the *Shape class* contains this declaration:

```
Public MustOverride Sub Draw( )
```

This declares the Draw() method, which takes no arguments. The *MustOverride* keyword specifies that the base class does not provide an implementation for this method and that derived classes must do so. It is sometimes preferable to allow the base class to provide a default implementation, yet allow derived classes to substitute their own implementations. Classes that don't provide their own implementations use the base class's implementation by default. Consider the following class definitions:

```
Class BaseClass
    Public Overridable Sub SomeMethod( )
    Console.WriteLine("BaseClass definition")
    End Sub
    End Class ' BaseClass
    Class DerivedClass
    Inherits BaseClass
    Public Overrides Sub SomeMethod( )
    Console.WriteLine("DerivedClass definition")
    End Sub
    End Class ' DerivedClass
    Class DerivedClass2
    Inherits BaseClass
End Class ' DerivedClass2
```

The BaseClass class defines a method called SomeMethod. In addition to providing an implementation of this method, the declaration specifies the Overridable keyword. This signals to the compiler that it's okay to override the method in derived classes. Without this modifier, derived classes cannot override the method. The DerivedClass class overrides this method by defining a method having the same name and signature and by specifying the Overrides keyword. The Visual Basic .NET compiler requires that the Overrides keyword be present to ensure that the developer actually meant to override a base-class method. The DerivedClass2 class does not override the SomeMethod method. Calls to SomeMethod through objects of type DerivedClass2 will invoke the BaseClass definition of SomeMethod. Here is an example:

```
Dim b As New BaseClass( )
Dim d As New DerivedClass( )
Dim d2 As New DerivedClass2( )
b.SomeMethod( )
d.SomeMethod( )
d2.SomeMethod( )
```

This code results in the following output:

```
BaseClass definition
DerivedClass definition
BaseClass definition
```

The SomeMethod implementation in the DerivedClass class can itself be overridden by a class deriving from DerivedClass. This can be prevented, if desired, by specifying the NotOverridable keyword in the definition of the SomeMethod method of the DerivedClass class, as shown here:

```
Class DerivedClass
    Inherits BaseClass
    Public NotOverridable Overrides Sub SomeMethod( )
    ' ...
    End Sub
End Class
```

2.15.6.7 Overloading

When two or more different methods conceptually perform the same task on arguments of different types, it is convenient to give the methods the same name. This technique is called *overloading* and is supported by Visual Basic .NET. For example, the following code defines an overloaded SquareRoot method that can take either a Long or a Double as a parameter:

```
Public Function SquareRoot( _ByVal Value As Long _) As Double
    ' ...
End Function

Public Function SquareRoot( _ByVal Value As Double _) As Double
    ' ...
End Function
```

When a call is made to the SquareRoot method, the version called is determined by the type of the parameter passed to it. For example, the following code calls the version of the method that takes a Long:

```
Dim result As Double = SquareRoot(10)
And this code calls the version that takes a Double:
Dim result As Double = SquareRoot(10.1)
```

Careful readers will note that in the first case the type of the argument is actually Integer, not Long. The Long version of the method is invoked because it is the closest match to the given argument. If there were also an Integer version of the method, that version would have been invoked, because it is a better match to the given argument. The .NET runtime always attempts to invoke the most appropriate version of an overloaded function, given the arguments provided.

If no suitable overload is found, a compiler error occurs (if Option Strict is on) or a runtime exception occurs (if Option Strict is off). The name of a method together with the number and types of its arguments are called the *signature* of the method.

The signature uniquely identifies a specific overloaded version of a specific method. Note that the return type is not part of the signature. Two versions of an overloaded method are not permitted to differ only by return type.

2.15.6.8 Overloading inherited methods

A method can also overload a method in a base class. Be careful to note the difference between *overloading* a base-class method and *overriding* a base-class method. *Overriding* means that the base-class method and the derived-class method have the same signature and that the derived-class method is replacing the base-class method. In addition, the base-class method must be marked with the Overridable keyword. *Overloading* means that they have different signatures and that both methods exist as overloads in the derived class. When overloading a method defined in a base class, the derived-class method declaration must include the Overloads keyword, but the base-class method doesn't have any special keyword attached to it. Here's an example:

```
Public Class BaseClass
    Public Sub SomeMethod( )
    ' ...
    End Sub
End Class
```

```
Public Class DerivedClass
        Inherits BaseClass
        Public Overloads Sub SomeMethod(ByVal i As Integer)
        ' ...
        End Sub
End Class
```

The requirement for the Overloads keyword helps to document the fact that a base-class method is being overloaded. There is no technical reason that the compiler requires it, but it is required nevertheless to help prevent human error.

2.15.6.9 Shadowing

The Shadows keyword allows a derived-class method to hide all base-class methods with the same name. Consider the following code, which does not use the Shadows keyword:

```
Public Class BaseClass
    Public Overridable Sub SomeMethod( )
    ' ...
    End Sub

    Public Overridable Sub SomeMethod(ByVal i As Integer)
    ' ...
    End Sub
End Class

Public Class DerivedClass
    Inherits BaseClass
    Public Overloads Overrides Sub SomeMethod( )
    ' ...
    End Sub
End Class
```

The base class overloads the SomeMethod method, and the derived class overrides the version of the method having no parameters. Instances of the derived class not only possess the parameterless version defined in the derived class, but they also inherit the parameterized version defined in the base class. In contrast, consider the following code, which is the same except for the declaration of the SomeMethod method in the derived class:

```
Public Class BaseClass
    Public Overridable Sub SomeMethod( )
    ' ...
    End Sub

    Public Overridable Sub SomeMethod(ByVal i As Integer)
    ' ...
    End Sub
End Class
    Public Class DerivedClass
    Inherits BaseClass
    Public Shadows Sub SomeMethod( )
    ' ...
    End Sub
End Class
```

In this version, instances of the derived class possess only the parameterless version declared in the derived class. Neither version in the base class can be called through a reference to the derived class.

2.15.7 Properties

Properties are members that are accessed like fields but are actually method calls. The idea is that a class designer may wish to expose some data values but needs to exert more control over their reading and writing than is provided by fields. Properties

are also useful for exposing values that are calculated. This is demonstrated by the Center property in the Shape class of Example 2-7. The property can be read and written just like the Origin and Size fields. However, there is no actual data member called Center. When code reads the Center property, a call is generated to the Center property's *getter*. When a new value is assigned to the Center property, a call is generated to the property's *setter*.
Property declarations look like this:

```
[ property modifiers ] Property [ attributes ] Property_Name _
    ( [ parameter list ] ) [ As Type_Name ] [ implements list ]
[ getter ]
[ setter ]
End Property
```

The components of the declaration are:

property modifiers

Further defined as:

```
[ Default ][ access modifier ][ override modifier ] _[ overload modifier ] [
    shared modifier ] [read/write modifier ]
```

If the Default keyword is present, it specifies the property as the default property of the class. Only one property in a class can be the class's default property, and only parameterized (or *indexed*) properties can be default properties. So what is a default property? A default property is just a property that can be referenced without actually specifying the property's name. For example, if a class has a default property called Item, takes an Integer argument, and is of type String, the following two lines are equivalent to each other:

```
myObject.Item(3) = "hello, world"
myObject(3) = "hello, world"
```

Note: Previous versions of Visual Basic did not constrain parameterized properties as the only possible default properties. The *access modifier*, *override modifier*, *overload modifier*, and *shared modifier* clauses have the same meanings as discussed later in this chapter in relation to method definitions. The *read/write modifier* clause is defined as:

ReadOnly | WriteOnly
This clause determines whether the property is read/write (signified by the absence of ReadOnly and WriteOnly), read-only (signified by ReadOnly), or write-only (signified by WriteOnly).Property keyword Identifies this as a property definition.

attributes
Represents a comma-separated list of attributes to be stored as metadata with the property.Attributes are discussed earlier in this chapter.

Property_Name
Represents the name of the property.

parameter list
Permits properties to have parameters. Parameterized properties are also called indexed properties. See the discussion of parameter lists earlier in this chapter.

As *Type_Name*
Indicates the data type of the property. This clause is optional if Option Strict is off; otherwise, it is required. If this clause is omitted, the property defaults to type Object.

implements list
Has the same meaning as for method definitions.

getter
Provides the method that is executed when the property is read. Its form is:

```
    Get
        'line of codes
    End Get
```

The value returned by the *getter* is returned as the value of the property. To return a value from the *getter*, either use the Return statement with an argument or assign a value to the property name. The value returned must be compatible with the data type of the property. It is an error to provide a *getter* if the property has been marked WriteOnly.

setter

Provides the method that is executed when the property is written. Its form is:

```
    Set [ ( ByVal Value [ As Type_Name ] ) ]

        'line of codes

    End Set
```

The value assigned to the property is passed to the method through the parameter specified in the Set statement. The type specified in the Set statement must match the type specified in the Property statement. Alternatively, the parameter declaration can be omitted from the Set statement. In this case, the value assigned to the property is passed to the method through a special keyword called Value. For example, this *setter* copies the passed-in value to a class data member called MyDataMember:

```
    Set
        MyDataMember = Value
    End Set
```

The data type of Value is the same as the data type of the property.End Property keywords Indicates the end of the property definition.Review the property definition from Example 2-7:

```
' This property allows the class user to find or set the
' center of a shape.
Public Property Center( ) As Point
    Get
        Dim retval As Point
        retval.X = Origin.X + (Size.XExtent \ 2)
        retval.Y = Origin.Y + (Size.YExtent \ 2)
        Return retval
    End Get
    Set
        Dim currentCenter As Point = Center
        Origin.X += Value.X - currentCenter.X
        Origin.Y += Value.Y - currentCenter.Y
    End Set
End Property
```

This is a public property called Center that has a type of Point. The *getter* returns a value of type Point that is calculated from some other members of the class. The *setter* uses the passed-in value to set some other members of the class.

2.15.8 The Me and MyClass Keywords

There are several ways for code to access members of the class in which the code is running. As long as the member being accessed is not hidden by a like-named declaration in a more immediate scope, it can be referenced without qualification:

```
Public Class SomeClass
    Public SomeValue As Integer
        Public Sub SomeMethod( )
        ' ...
        SomeValue = 5
        ' ...
        End Sub
End Class
```

If the member is hidden by a more immediate declaration, the Me keyword can be used to qualify the reference. Unqualified references refer to the more local declaration, as shown here:

```
Public Class SomeClass
    Public SomeValue As Integer

    Public Sub SomeMethod(ByVal SomeValue As Integer)
    ' ...
    ' Assign the passed-in value to a field.
    Me.SomeValue = SomeValue
    ' ...
    End Sub

End Class
```

The Me keyword is an implicit variable that holds a reference to the object instance running the code.Related to the Me keyword, but subtly different, is the MyClass keyword. While the Me keyword can be used in any context in which an object reference is expected, the MyClass keyword is used only for member access; it must always be followed by a period and the name of a member, as shown here:

```
Public Class SomeClass
    Public SomeValue As Integer
    Public Sub SomeMethod(ByVal SomeValue As Integer)
    ' ...
    ' Assign the passed-in value to a field.
    MyClass.SomeValue = SomeValue
    ' ...
    End Sub
End Class
```

As you can see, there is overlap in the contexts in which these two keywords can be used, and for most circumstances they can be considered synonymous. However, there are situations in which the two keywords differ:

- The Me keyword can't be used in shared methods because it represents a specific object instance of the class, yet shared methods can be executed when no instance exists.
- The keywords can behave differently when used in a class from which other classes are derived. Consider this code:

```
Public Class BaseClass
    Public Sub Method1( )
        Console.WriteLine("Invoking Me.Method2...")
        Me.Method2( )
        Console.WriteLine("Invoking MyClass.Method2...")
        MyClass.Method2( )
    End Sub
        Public Overridable Sub Method2( )
        Console.WriteLine("BaseClass.Method2")
    End Sub
    End Class

    Public Class DerivedClass
        Inherits BaseClass
        Public Overrides Sub Method2( )
        Console.WriteLine("DerivedClass.Method2")
    End Sub
End Class
```

This code defines two classes:

BaseClass and *DerivedClass*.
BaseClass defines two methods:*Method1* and *Method2*.DerivedClass inherits *Method1* but provides its own implementation for *Method2*.

Now consider the following instantiation of DerivedClass, as well as a call through it to the Method1 method:

```
Dim d As New DerivedClass( )
d.Method1( )
```

This produces the following output:

```
Invoking Me.Method2...
DerivedClass.Method2
Invoking MyClass.Method2...
BaseClass.Method2
```

The call to Method1 through the DerivedClass instance calls the Method1 implementation inherited from BaseClass. Method1 calls Method2 twice: once through the Me keyword and once through the MyClass keyword. The Me keyword is a reference to the actual object instance, which is of type DerivedClass. Therefore, Me.Method2( ) invokes the DerivedClass class's implementation of Method2. In contrast, the MyClass keyword is used for referencing members in the class in which the code is defined, which in this case is the BaseClass class. Therefore, MyClass.Method2( ) invokes the BaseClass class's implementation of Method2.

2.15.9 The MyBase Keyword

The MyBase keyword is used to access methods on the base class. This feature is commonly used when an overriding method needs to call the base-class implementation of the same method:

```
 Public Class BaseClass
    Public Overridable Sub DoSomething( )
    ' ...
    End Sub
End Class

Public Class DerivedClass
    Inherits BaseClass
    Public Overrides Sub DoSomething( )
    ' Start by calling the base-class implemenation of DoSomething.
    MyBase.DoSomething( )
    ' Then continue on with additional stuff required by DerivedClass.
    ' ...
    End Sub
End Class
```

2.15.10 Nested Classes

Class definitions can be nested. The nested class is considered a member of the enclosing class. As with other members, dot notation is used for accessing the inner class definition. Consider this nested class definition:

```
Public Class OuterClass
    Public Class InnerClass
        Public Sub SomeMethod( )
            Console.WriteLine("Hello from InnerClass.SomeMethod!")
        End Sub
    End Class
End Class
```

Instantiating an object of type InnerClass requires qualifying the name with the name of the enclosing class:

```
Dim x As New OuterClass.InnerClass( )
```

```
    x.SomeMethod( )
```

The accessibility of the inner-class declaration can be controlled with the class declaration's access modifier. For example, in the following definition, InnerClass has been declared with the Private modifier, making it visible only within the confines of the OuterClass class:

```
    Public Class OuterClass
        Private Class InnerClass
        ' ...
        End Class
    End Class
```

Classes can be nested as deeply as desired.

2.15.11 Destructors

Just as constructors are methods that run when objects are instantiated, it is often convenient to define methods that run when objects are destroyed (that is, when the memory that was allocated to them is returned to the pool of free memory). Such a method is called a *destructor*.

It doesn't have special syntax for declaring destructors, as it does for constructors. Instead, Visual Basic .NET uses the specially named methods i.e.-
Finalize() and Dispose() to perform the work normally associated with destructors.

Because this mechanism is actually part of the .NET Framework rather than Visual Basic .NET, it is explained in later in this SLM.

2.15.12 Early Versus Late Binding

Declarations permit the compiler to know the data type of the item being declared. Here is the declaration of a variable of type String:

```
    Dim s As String
```

Knowing the data type of a variable (or parameter, field, etc.) allows the compiler to determine what operations are permitted on any object referenced by the variable. For example, given the previous declaration of s as String, the compiler knows that the expression s.Trim( ) is permitted (because it is defined in the String class), while s.Compact( ) is not (because there is no such method in the String class). During compilation, the Visual Basic .NET compiler complains if it encounters such errors. There is, however, one case in which the developer is permitted to relax this constraint. If Option Strict is turned off, the compiler forgoes this kind of checking on variables of type Object. For example, the following code will compile without difficulty, even though the Object class doesn't have a method called "Whatever":

```
    Option Strict Off
    ' ...
        Dim obj As Object
    obj.Whatever( )
```

With Option Strict off, the compiler compiles obj.Whatever( ) to code that checks to see if the runtime type of the object referenced by obj is a type that possesses a Whatever method. If it does, the Whatever method is called. If not, a runtime exception is raised. Here is such a scenario: Option Strict Off Public Class WhateverClass

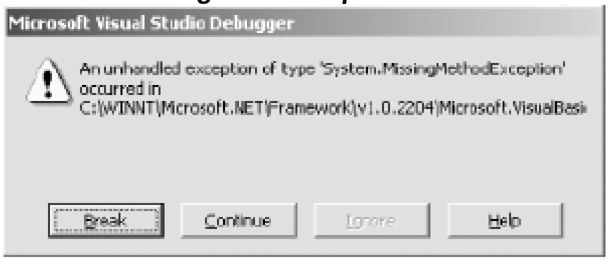
```
    Public Sub Whatever( )
        Console.WriteLine("Whatever!")
    End Sub

    Public Class TestClass
        Public Shared Sub TestMethod( )
        Dim obj As Object
        obj = New WhateverClass( )
        obj.Whatever( )
    End Sub
End Class
```

Because Option Strict is off, this code compiles just fine. Because obj references an object at runtime that is of a class that implements a Whatever method, it also runs just fine. However, consider what happens if the Whatever method is removed from the WhateverClass class:

```
Option Strict Off
    Public Class WhateverClass
End Class

Public Class TestClass
    Public Shared Sub TestMethod( )
    Dim obj As Object
    obj = New WhateverClass( )
    obj.Whatever( )
    End Sub
End Class
```

The code still compiles without a problem, because Option Strict is off. However, at runtime there is a problem, as shown in Figure 2-2.

Figure 2-2. A problem



Microsoft Visual Studio Debugger

An unhandled exception of type 'System.MissingMethodException' occurred in C:\WINNT\Microsoft.NET\Framework\v1.0.2204\Microsoft.VisualBasi

The technique of accessing members through a generic object of type Object is called *late binding*. "Late" means that whether the desired member is really there is not known until the statement is actually executed. In contrast, leaving Option Strict on and accessing members only through variables that have been declared as the appropriate type is known as *early binding*. "Early" means that whether the member access is legitimate is known at compile time.
Late binding is less efficient than early binding because additional checks are needed at runtime to determine whether the requested member actually exists on the runtime object and, if it does, to access that member. The worst part of late binding is that it can mask certain program errors (such as mistyped member names) until runtime. In general, this is bad programming practice.

2.14 - 2.15 Check Your Progress
State whether statements are True or False
1. Visual Basic. NET is a line-oriented language.
2. The call statement invokes a subrouting or function.
3. Object instantiation is done without using new keyword.
4. Fields can be declared with the Read Only modifier.
5. Math class contains definition of math related function.

2.16 INTERFACES

It is useful to make a distinction between a class's *interface* and its *implementation*. Conceptually, the interface of a class is the set of members that are visible to users of the class—i.e., the class's public members. The public members are thought of as comprising the class's interface because they are the only way that code outside of the class can interact (i.e., interface) with objects of that class. In contrast, the implementation is comprised of the class's code plus the set of members that are not public. It is possible to take this interface concept further and separate interface

definition from class definition altogether. This has benefits that will be shown shortly. To define an interface, use the Interface statement:
Public Interface ISomeInterface:

```
    Sub SomeSub( )
        Function SomeFunction( ) As Integer
        Property SomeProperty( ) As String
        Event SomeEvent( _ByVal sender As Object, _ByVal e As SomeEventArgs _)
    End Interface
```

An interface declaration defines methods, properties, and events that will ultimately be implemented by some class or structure definition. Because interfaces never include any implementation, the declarations are headers only—never any implementation code; End Sub, End Function, or End Property statements; or property get or set blocks. There are no access modifiers (Public, Private, etc.) because all members of an interface are public by definition. By convention, interface names start with the letter **"I"**.

To provide an implementation for a given interface, it is necessary to define a class or structure. For example, the following class implements the interface defined earlier:

```
        Public Class SomeClass
                ' This indicates that the class implements the methods,
                ' properties, and events of the ISomeInterface interface.
            Implements ISomeInterface
                ' This method implements the SomeSub method of the
                ' ISomeInterface interface.
            Private Sub SomeSub( ) Implements ISomeInterface.SomeSub
                ' ...
            End Sub

            ' This method implements the SomeFunction method of the
            ' ISomeInterface interface.

            Private Function SomeFunction( ) As Integer _
                Implements ISomeInterface.SomeFunction
                ' ...
            End Function

            ' This property implements the SomeProperty property of the
            ' ISomeInterface interface.Private Property SomeProperty( ) As String _

            Implements ISomeInterface.SomeProperty
                Get
                ' ...
                End Get
                Set
                ' ...
                End Set
            End Property
            ' This event implements the SomeEvent event of the
            ' ISomeInterface interface.
            Private Event SomeEvent( _ByVal sender As Object, _ByVal e As
                SomeEventArgs _) Implements ISomeInterface.SomeEvent
        End Class
```

The key elements of this class definition are:

- ❖ The class-declaration header is immediately followed by the Implements statement, indicating that this class will expose the ISomeInterface interface:

```
            Public Class SomeClass
                ' This indicates that the class implements the methods,
                ' properties, and events of the ISomeInterface interface.
               Implements ISomeInterface
```

This information is compiled into the class. Class users can find out whether a given class implements a given interface by attempting to assign the object reference to a variable that has been declared of the interface type, like this:

```
Dim obj As Object
Dim ifce As ISomeInterface
' ...
' Get an object reference from somewhere.
obj = New SomeClass( )
' ...
' Try to convert the object reference to a reference of type
' ISomeInterface. If the object implements the ISomeInterface
' interface, the conversion succeeds. If the object doesn't
' implement the ISomeInterface interface, an exception of
' type InvalidCastException (defined in the System namespace)
' is thrown.
ifce = CType(obj, ISomeInterface)
```

- ❖ For each method, property, and event in the interface, there is a corresponding method, property, or event in the class that has precisely the same signature and return value. The names don't have to match, although they match in the example.
- ❖ The declaration header for each method, property, and event in the class that implements a corresponding item in the interface must have an *implements clause*. This is the keyword Implements followed by the qualified name of the interface method, property, or event being implemented. Additional things to note about implementing interfaces include:
- ❖ The access modifiers in the class-member declarations need not be Public. Note that in the example all the members are marked as Private. This means that the members are accessible only when accessed *through* the ISomeInterface interface. This will be shown in a moment.
- ❖ The class definition can include members that are not part of the implemented interface. These can be public if desired. This results in a class that effectively has two interfaces: the *default interface*, which is the set of members defined as Public in the class definition; and the *implemented interface*, which is the set of members defined in the interface named in the Implements statement.
- ❖ Classes are permitted to implement multiple interfaces.

To access members defined by an interface, declare a variable as that interface type and manipulate the object through that variable. For example:

```
Dim x As ISomeInterface = New SomeClass( )
x.SomeFunction( )
```

This code declares x as a reference to an object of type ISomeInterface. That's right: interface definitions define new types. Declared in this way, x can take a reference to any object that implements the ISomeInterface interface and access all the members that IsomeInterface defines, confident that the underlying object can handle such calls. This is a powerful feature of defining and implementing explicit interfaces. Objects that explicitly implement an interface can be used in any context in which that interface is expected; objects that implement multiple interfaces can be used in any context in which any of the interfaces is expected. Interface definitions can inherit from other interface definitions in the same way that classes can inherit from other classes. For example:

```
Public Interface ISomeNewInterface
Inherits ISomeInterface
Sub SomeNewSub( )
End Interface
```

This defines a new interface called ISomeNewInterface that has all the members of the ISomeInterface interface plus a new member, called SomeNewSub. Any class or structure that implements the ISomeNewInterface interface must implement all members in both interfaces. Any such class is then considered to implement both interfaces and could be used in any context where either ISomeInterface or ISomeNewInterface is required.

2.17 STRUCTURES

Structures define value types. Variables of a value type store an actual value, as opposed to a reference to a value stored elsewhere. Contrast this with classes, which define reference types.Variables of a reference type store a reference (a pointer) to the actual value. See the discussion of value types versus reference types in Section 2.5 earlier in this chapter. Example 2-8 shows a structure definition.

Example 2-8. A structure definition

```
Public Structure Complex
        ' The IFormattable interface provides a generic mechanism for
        ' asking a value to represent itself as a string.
    Implements IFormattable
        ' These private members store the value of the complex number.
    Private m_RealPart As Double
    Private m_ImaginaryPart As Double
        ' These fields provide potentially useful values, similar to the
        ' corresponding values in the Double type. They are initialized
        ' in the shared constructor. The ReadOnly modifier indicates that
        ' they can be set only in a constructor.
    Public Shared ReadOnly MaxValue As Complex
    Public Shared ReadOnly MinValue As Complex
        ' This is a shared constructor. It is run once by the runtime
        ' before any other access to the Complex type occurs. Note again
        ' that this is run only once in the life of the program--not once
        ' for each instance. Note also that there is never an access
        ' modifier on shared constructors.
    Shared Sub New( )
        MaxValue = New Complex(Double.MaxValue, Double.MaxValue)
        MinValue = New Complex(Double.MinValue, Double.MinValue)
    End Sub
        ' The RealPart property gives access to the real part of the
        ' complex number.
    Public Property RealPart( ) As Double
    Get
        Return m_RealPart
    End Get
        Set(ByVal Value As Double)
        m_RealPart = Value
    End Set
    End Property

        ' The ImaginaryPart property gives access to the imaginary part
        ' of the complex number.

    Public Property ImaginaryPart( ) As Double
        Get
            Return m_ImaginaryPart
        End Get

        Set(ByVal Value As Double)
            m_ImaginaryPart = Value
        End Set
    End Property

        ' This is a parameterized constructor allowing initialization of
        ' a complex number with its real and imaginary values.

    Public Sub New( _ByVal RealPart As Double, _
        ByVal ImaginaryPart As Double _)
        m_RealPart = RealPart
        m_ImaginaryPart = ImaginaryPart
    End Sub

    ' This function computes the sum of two Complex values.
    Public Shared Function Add( _ByVal Value1 As Complex, _ByVal Value2 As
        Complex _) As Complex
            Dim retval As Complex
            retval.RealPart = Value1.RealPart + Value2.RealPart
            retval.ImaginaryPart = Value1.ImaginaryPart + Value2.ImaginaryPart
            Return retval
    End Function

        ' This function computes the difference of two Complex values.
    Public Shared Function Subtract( _ByVal Value1 As Complex, _  ByVal  Value2  As
        Complex _) As Complex
            Dim retval As Complex
            retval.RealPart = Value1.RealPart - Value2.RealPart
            retval.ImaginaryPart = Value1.ImaginaryPart - Value2.ImaginaryPart
            Return retval
    End Function

    ' This function computes the product of two Complex values.Public Shared Function

    Multiply( _ByVal Value1 As Complex, _ByVal Value2 As Complex _) As Complex
        Dim retval As Complex
        retval.RealPart = Value1.RealPart * Value2.RealPart _
        - Value1.ImaginaryPart * Value2.ImaginaryPart
        retval.ImaginaryPart = Value1.RealPart * Value2.ImaginaryPart _
        + Value1.ImaginaryPart * Value2.RealPart
        Return retval
        End Function

    ' This function computes the quotient of two Complex values.
    Public Shared Function Divide( _ByVal Value1 As Complex, _ByVal Value2 As
        Complex _) As Complex
            Dim retval As Complex
            Dim numerator1 As Double
            Dim numerator2 As Double
            Dim denominator As Double
            numerator1 = Value1.RealPart * Value2.RealPart _
            + Value1.ImaginaryPart * Value2.ImaginaryPart
            numerator2 = Value1.ImaginaryPart * Value2.RealPart _
            - Value1.RealPart * Value2.ImaginaryPart
            denominator = Value2.RealPart ^ 2 + Value2.ImaginaryPart ^ 2
            retval.RealPart = numerator1 / denominator
            retval.ImaginaryPart = numerator2 / denominator
            Return retval
    End Function

        ' This function implements IFormattable.ToString. Because it is
        ' declared Private, this function is not part of the Complex
        ' type's default interface. Note that the function name need
        ' not match the name as declared in the interface, nor need
        ' it be in the format shown here.

    Private Function IFormattable_ToString( _ByVal format As String, _ByVal
        formatProvider As IFormatProvider _) As String Implements
        IFormattable.ToString
        Dim realFormatter As IFormattable = m_RealPart
        Dim imaginaryFormatter As IFormattable = m_ImaginaryPart
        Return realFormatter.ToString(format, formatProvider) & " + " _
        & imaginaryFormatter.ToString(format, formatProvider) & "i"
    End Function

    ' This function formats the Complex value as a string.
    Public Overrides Function ToString( ) As String Return CType(Me,
        IFormattable).ToString(Nothing, Nothing)
    End Function
End Structure ' Complex
```

Structure definitions can include fields, properties, methods, constructors, and more—any member, in fact, that a class definition can have. Unlike class definitions, however, structures are constrained in several ways:

- Structures are not permitted to inherit from any other type. (However, structures implicitly inherit from System.ValueType, which in turn inherits from Object.)
- Structures cannot override methods implicitly inherited from System.ValueType.
- No type can inherit from a structure.
- Structures are not permitted to have parameterless constructors.
  Consider this array declaration:

```
Dim a(1000000) As SomeStructure
```

When an array of value types is created, it is immediately filled with instances of the value type. This behavior corresponds to what you'd expect from an array holding a primitive type (such as Integer). If parameterless constructors were permitted for structures, this array declaration would result in 1,000,000 calls to the constructor. Ouch.

- Structures are not permitted to have destructors.
- Field members in structures are not permitted to be initialized in their declarations. This includes the special cases of using As New *type* in the declaration or specifying an initial size in an array declaration.

2.17.1 Boxing and Unboxing

Value types are optimized for size and speed. They don't carry around the same amount of overhead as reference types. It would not be very efficient if every four-byte integer also carried around a fourbyte reference. There are times, however, when treating value types and reference types in a polymorphic way would be nice. Consider this method declaration, which takes any number of arguments of any type and processes them in some way:

```
Public Shared Sub Print(ParamArray ByVal objArray( ) As Object)
    Dim obj As Object
    For Each obj In objArray
    ' ...
    Next
End Sub
```

Clearly, objArray is an array of reference types, and obj is a reference type. Yet it would be nice to pass value types and reference types to the method, like this:
Print("hello, world", SomeObject, 4, True)
In fact, this is possible. When a value type is assigned to a variable of type Object or passed in a parameter of type Object, it goes through a process known as *boxing*. To box a value type means to allocate memory to hold a copy of the value, then copy the value into that memory, and finally manipulate or store a reference to the value. *Unboxing* is the opposite process: taking a reference to a value type and copying the referenced value into an actual value type.
Boxing and unboxing are done on your behalf by the .NET runtime—there is nothing you have to do to facilitate it. You should be aware of it, however, because the box and unbox operations aren't free.

2.18 ENUMERATIONS

An *enumeration* is a type whose values are explicitly named by the creator of the type. The .NET Framework and Visual Basic .NET define many enumerations for their and your use. In addition,
Visual Basic .NET provides syntax for defining new enumerations. Here is an example:

```
Public Enum Rainbow
    Red
    Orange
    Yellow
    Green
    Blue
    Indigo
    Violet
End Enum
```

This declaration establishes a new type, called Rainbow. The identifiers listed within the body of the declaration become constant values that may be assigned to variables of the Rainbow type. Here is a declaration of a variable of type Rainbow and an initial assignment to it:

```
Dim myRainbow As Rainbow = Rainbow.Blue
```

Note that the value name is qualified by the type name.
Enumerations are value types that implicitly inherit from the .NET Framework's System.Enum type (which in turn inherits from System.ValueType). That means that every enumeration has access to the members defined by System.Enum. One such member is the ToString method, which returns a string containing the name of the value. This is handy for printing:

```
Dim myRainbow As Rainbow = Rainbow.Blue
Console.WriteLine("The value of myRainbow is: " & myRainbow.ToString( ))
```

This code results in the following output:

```
The value of myRainbow is: Blue
```

The values of an enumeration are considered as ordered. Thus, comparisons are permitted between variables of the enumeration type:

```
Dim myRainbow As Rainbow
Dim yourRainbow As Rainbow
' ...
If myRainbow < yourRainbow Then
' ...
End If
```

Variables of an enumeration type can be used as indexes in For...Next statements. For example:

```
For myRainbow = Rainbow.Red To Rainbow.Violet
' ...
Next
```

Internally, Visual Basic .NET and the .NET Framework use values of type Integer to represent the values of the enumeration. The compiler starts with 0 and assigns increasing Integer values to each name in the enumeration. It is sometimes useful to override the default Integer values that are assigned to each name. This is done by adding an initializer to each enumeration constant. For example:

```
Public Enum MyLegacyErrorCodes
    NoError = 0
    FileNotFound = -1000
    OutOfMemory = -1001
    InvalidEntry = -2000
    End Enum
```

It is also possible to specify the type of the underlying value. For example:

```
    Public Enum Rainbow As Byte
    Red
    Orange
    Yellow
    Green
    Blue
    Indigo
    Violet
End Enum
```

This could be an important space-saving measure if many values of the enumeration will be stored somewhere. The only types that can be specified for an enumeration are

Byte, Short, Integer, and Long. Sometimes enumerations are used as flags, with the idea that multiple flags can be combined in a single value. Such an enumeration can be defined by using the Flags attribute. (Attributes are discussed later in this chapter.) Here is an example:

```
<Flags( )> Public Enum Rainbow
    Red = 1
    Orange = 2
    Yellow = 4
    Green = 8
    Blue = 16
    Indigo = 32
    Violet = 64
End Enum
```

Note two important things in this definition:

- The first line of the definition starts with <Flags( )>. This indicates that values of this type can be composed of multiple items from the enumeration.
- The items in the enumeration have values that are powers of two. This ensures that each combination of items has a unique sum. For example, the combination of Yellow, Blue, and Violet has a sum of 84, which can't be attained with any other combination of items.

Individual values are combined using the Or operator.
The ToString method is smart enough to sort out the value names when creating a string representation of the value. For example, given the previous assignment, consider the following call to the ToString method:

```
Console.WriteLine(myRainbow.ToString( ))
```

This statement produces the following output:

```
Green, Blue
```

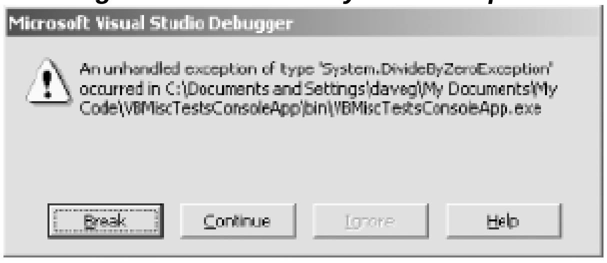
2.19 EXCEPTIONS

Sometimes errors or exceptional conditions prohibit a program from continuing its current activity. A classic example is division by zero:

```
Dim x As Integer = 0
Dim y As Integer = 1 \ x
```

When the process hits the line containing the integer division, an *exception* occurs. An exception is any occurrence that is not considered part of normal, expected program flow. The runtime detects, or *catches*, this exception and takes appropriate action, generally resulting in termination of the offending program. Figure 2-3 shows the message box that is displayed when this code is run within the Visual Studio .NET IDE.

Figure 2-3. A divide-by-zero exception


Microsoft Visual Studio Debugger

An unhandled exception of type 'System.DivideByZeroException' occurred in C:\Documents and Settings\daveg\My Documents\My Code\VBMiscTestsConsoleApp\bin\VBMiscTestsConsoleApp.exe

Visual Basic .NET programs can and should be written to catch exceptions themselves. This is done by wrapping potentially dangerous code in Try...End Try blocks. Example 2-9 shows how to
catch the divide-by-zero exception.

Example 2-9. Catching an exception

```
Try
    Dim x As Integer = 0
    Dim y As Integer = 1 \ x
Catch e As Exception
    Console.WriteLine(e.Message)
End Try
```

When the program attempts the division by zero, an exception occurs, and program execution jumps to the first statement in the Catch block. The Catch statement declares a variable of type Exception that receives information about the exception that occurred. This information can then be used within the Catch block to record or report the exception, or to take corrective action. The previous code merely displays the message associated with the exception that occurred, as shown here:Attempted to divide by zero.

After executing the statements in the *Catch block*, program execution continues with whatever follows the End Try statement. In *Try blocks* in which no exception occurs, execution continues through to the last statement of the Try block and then skips the statements in the Catch block.

The variable declared in the Catch statement of Example 2-9 is of type Exception (defined in the System namespace). All exceptions are represented by types that derive, either directly or indirectly, from the Exception type. The As *type_name* clause of the Catch statement specifies the type of exception that the associated block of code can handle. Exceptions of the indicated type, or of any type derived (directly or indirectly) from the indicated type, are handled by the associated block of code.
Look again at the Catch statement from Example 2-9:
Catch e As Exception

Because all exceptions derive from the Exception type, any exception that occurs during execution of the Try block in Example 2-9 results in execution of the Catch block. This behavior can be modified by providing a more specific exception type in the Catch statement. Example 2-10 is identical to Example 2-9, except that it catches *only* divide by zero exceptions.

Example 2-10. Catching a specific exception

```
Try
    Dim x As Integer = 0
    Dim y As Integer = 1 \ x
Catch e As System.DivideByZeroException

    Console.WriteLine(e.Message)
End Try
```

If any exception other than DivideByZeroException were to occur in the Try block of Example 2-10,it would not be caught by the code shown. What happens in that case depends on the rest of the code in the program. Try...End Try blocks can be nested, so if there is a surrounding Try...End Try block with a suitable Catch statement, it will catch the exception. Alternatively, if the calling routine couches the method call within a *Try...End Try block* having a suitable Catch statement, execution jumps out of the current method and into the associated Catch block in the calling routine. If no suitable Catch block exists in the calling routine, the search for a suitable Catch continues up the call chain until one is found or until all callers have been examined. If no suitable Catch block exists anywhere in the call chain, the runtime environment catches the exception and terminates the application. Try...End Try blocks can include multiple Catch blocks, which allows different exceptions to be handled in different ways. For example, the following code handles two specific exceptions, allowing all others to go unhandled:

```
Try
' ...
```

```
    Catch e As System.DivideByZeroException
    ' ...
    Catch e As System.OverflowException
    ' ...
    End Try
```

Because all exception types are derived from the Exception type, the properties and methods of the Exception type are available on all exception types. In fact, most exception types don't define any additional properties or methods. The only reason they're defined as specific types is so that they can be specifically caught. The properties of the Exception class are:

HelpLink
A URN or URL that links to a help-file topic that explains the error in further detail. The type is String.

HResult
A COM HRESULT representing the exception. This is used for interoperating with COM components (a topic that is not discussed in this book). The type is integer.

InnerException
Sometimes a method may choose to throw an exception because it has caught an exception from some other internal method call. The outer method throws an exception that is meaningful to it and its caller, but the exception thrown by the inner method should also be communicated to the caller. This is the purpose of the InnerException property. This property contains a reference to the internal exception (if any) that led to the current exception. The type is Exception.

Message
The message associated with the exception. In general, this is a description of the condition that led to the exception and, where possible, an explanation of how to correct it. The type is String.

Source
The name of the application or object in which the exception occurred. The type is String.

StackTrace
A textual representation of the program call stack at the moment the exception occurred. The type is String.

TargetSite
A reference to an object of type MethodBase (defined in the System.Reflection namespace) that represents the method in which the exception occurred. If the system cannot obtain this information, this property contains Nothing.

The methods of the Exception class are:

GetBaseException
As discussed for the InnerException property, indicates that the current exception may be the end exception in a chain of exceptions. The GetBaseException method returns the first exception in the chain. This method takes no parameters. The return type is Exception.

GetObjectData
Serializes the Exception object into a SerializationInfo object . The syntax is:

```
Overridable Public Sub GetObjectData( _ByVal info As SerializationInfo, _ByVal
    context As StreamingContext _) Implements ISerializable.GetObjectData
```

ToString
Returns a text representation of the Exception. This includes the exception type, the message, the stack trace, and similar information for any inner exceptions. When an exception occurs, there is no facility for retrying the statement that caused the exception. If such behavior is desired, it must be explicitly coded. Here's one possibility:

```
Dim bSuccess As Boolean = False
Do
Try
    ' Some code that is to be protected.
    ' ...
    bSuccess = True
Catch e As Exception
    ' Some recovery action.
    ' ...
End Try

Loop Until bSuccess
```

Sometimes you must ensure that certain code is executed regardless of whether there is an exception. For example, if a file is opened, the file should be closed even when an exception occurs. Try...End Try blocks can include Finally blocks for this purpose. Code appearing in a Finally block is executed regardless of whether an exception occurs. If no exception occurs, the statements in the Finally block are executed after the statements in the Try block have been executed. If an exception does occur, the statements in the Finally block are executed after the statements in the Catch block that handles the exception are executed. If the exception is not handled, or if there are no Catch blocks, the statements in the Finally block are executed prior to forwarding the exception to any enclosing exception handlers. Here's an example of using a Finally block:

```
Dim s As System.IO.Stream =_
System.IO.File.Open("c:\test.txt", System.IO.FileMode.CreateNew)
Try
    ' Do something with the open stream.
    ' ...
Catch e As Exception
' Handle any exceptions.
' ...
Finally
    ' The stream should be closed whether or not there is an error.
    s.Close( )
End Try
```

Visual Basic .NET applications can intentionally *throw* exceptions to indicate errors or other unusual occurrences. For example, if a method is expecting an argument that is within a specific range and the actual value passed to the method is outside of that range, the method can throw an exception of type ArgumentOutOfRangeException (defined in the System namespace). This is done with the Throw statement, as shown in Example 2-11.

Example 2-11. Throwing an exception

```
Public Sub SomeMethod(ByVal MyParameter As Integer)
    ' Ensure that the argument is valid.
    If (MyParameter < 10) Or (MyParameter > 100) Then
        Throw New ArgumentOutOfRangeException( )
    End If
    ' Remainder of method.
    ' ...
End Sub
```

The Throw statement requires an instance of some type derived from the Exception type. When the Throw statement is reached, the runtime looks for an appropriate Catch block in the calling code to handle the exception. If no suitable Catch block is found, the runtime catches the exception itself and terminates the application. See Appendix B for a list of exception types defined in the System namespace. Visual Basic .NET applications can create their own exception types simply by declaring types that derive from the Exception type. Example 2-12 shows how the exception handling of Example 2-11 can be made more specific to the actual error that occurs. In Example 2-12, a new exception type called MyParameterOutOfRangeException is declared. Next, a method is shown that throws this exception. Lastly, a method is shown that handles the exception.

Example 2-12. Defining and using a custom exception

```
' Define a custom exception class to represent a specific error condition.

Public Class MyParameterOutOfRangeException
    Inherits Exception

    Public Sub New( )
        ' The Exception type has a constructor that takes an error message
        ' as its argument. Because the Message property of the Exception
        ' type is read-only, using this constructor is the only way that
        ' the Message property can be set.
        MyBase.New("The value passed in the MyParameter parameter" _
        & " is out of range. The value must be in the range of" _& " 10 through 100.")
    End Sub
End Class

        ' ...
        ' Define a method that may throw a custom exception.

        Public Sub SomeMethod(ByVal MyParameter As Integer)
            ' Ensure that the argument is valid.
            If (MyParameter < 10) Or (MyParameter > 100) Then
            Throw New MyParameterOutOfRangeException( )
            End If
            ' Remainder of method.
            ' ...
        End Sub
            ' ...
            ' Call the SomeMethod method, catching only the
            ' MyParameterOutOfRangeException exception.

        Public Sub SomeCaller( )
            Try
                SomeMethod(500)
                Catch e As MyParameterOutOfRangeException
                ' ...
            End Try
        End Sub
```

What About On Error?

Visual Basic 6 did not have exception objects and Try...Catch blocks. Instead, it used the On Error statement to specify a line within the current procedure to which execution should jump if an error occurred. The code at that point in the procedure could then examine the Err intrinsic object to determine the error that had occurred. For compatibility with previous versions, Visual Basic .NET continues to support the On Error and related statements, but they should not be used in new development, for the following reasons:

- Structured exception handling is more flexible.
- Structured exception handling does not use error codes. (Applicationdefined error codes often clashed with error codes defined by other applications.)
- Structured exception handling exists at the .NET Framework level, meaning that regardless of the language in which each component is written, exceptions can be thrown and caught across component boundaries.

2.20 DELEGATES

A *delegate* is a programmer-defined type that abstracts the ability to call a method. A delegate-type declaration includes the declaration of the signature and return type that the delegate encapsulates. Instances of the delegate type can then wrap any method that exposes the same signature and return type, regardless of the class on which the method is defined and whether the method is an instance method or shared method of the defining class. The method thus wrapped can be invoked through the delegate object. The delegate mechanism provides polymorphism for methods having the same signature and return type. Delegates are often used to implement callback mechanisms. Imagine a class that will be used by a program you are writing. This class provides some useful functionality, including the ability to call in to a method that you must implement within your program. Perhaps this callback mechanism is provided to feed your program data as it becomes available in the class you are using. One way to achieve this capability is through the use of delegates. Here's how:

1. The writer of the class you're using (call it a server class) declares a public delegate type that defines the signature and return value of the method that you will implement.
2. The writer of the server class exposes a method for clients of the class to pass in an instance of the delegate type.
3. You implement a method having the appropriate signature and return value.
4. You instantiate a new object of the delegate type.
5. You connect your method to your delegate instance.
6. You call the method defined in Step 2, passing in your delegate instance.
7. The server class now has a delegate instance that wraps your method. The class can call your method through the delegate at any time.
8. Depending on the application, it might be appropriate for the writer of the server class to provide a method that allows the client application to disconnect its delegate from the server to stop receiving callbacks.

Example 2-13 shows an example of this mechanism.

Example 2-13. Defining and using a delegate type to implement acallback mechanism

```
    ' This class is defined in the server component.

Public Class ServerClass

    ‘Even though the following declaration looks similar to a ' method declaration, it is
    ‘actually a type declaration. It compiles to a type that ultimately derives from the
    ‘System.Delegate type. The purpose of the method syntax in this declaration is to
    ‘define the signature and return type ' of the methods that instances of this delegate
    ‘type are able to wrap.

Public Delegate Sub MessageDelegate(ByVal msg As String)

    ‘The following is a private field that will hold an instance of ' the delegate type. The
    ‘instance will be provided by the client  by calling the RegisterForMessages
    ‘method. Even though this field can hold only a single delegate instance, the '
    ‘System.Delegate class itself is designed such that a ' delegate instance can refer
    ‘to multiple other delegate instances. This feature is inherited by all delegate
    ‘types.Therefore, the client will be able to register multiple ' delegates, if desired.
    ‘See the RegisterForMessages and UnregisterForMessages methods in the current
    ‘class to see ' how multiple delegates are saved.

Private m_delegateHolder As MessageDelegate = Nothing

    ' The client calls the RegisterForMessages method to give the ' server a delegate
    ‘instance that wraps a suitable method on the client.

Public Sub RegisterForMessages(ByVal d As MessageDelegate)

    'The System.Delegate class's Combine method takes two ' delegates and returns a
    ‘delegate that represents them both. The return type is System.Delegate, which
    ‘must be  explicitly converted to the appropriate delegate type.

    Dim    sysDelegate    As    System.Delegate=System.Delegate.Combine
    (m_delegateHolder, d)
    m_delegateHolder = CType(sysDelegate, MessageDelegate)
End Sub
```

```
' The client calls the UnregisterForMessages method to tell
' the server not to send any more messages through a
' particular delegate instance.

Public Sub UnregisterForMessages(ByVal d As MessageDelegate)

   ' The System.Delegate class's Remove method takes two
   ' delegates. The first is a delegate that represents a list
   ' of delegates. The second is a delegate that is to be
   ' removed from the list. The return type is
   ' System.Delegate, which must be explicitly converted to
   ' the appropriate delegate type.

   Dim sysDelegate As System.Delegate = _
   System.Delegate.Remove(m_delegateHolder, d)

   m_delegateHolder = CType(sysDelegate, MessageDelegate)
End Sub
   ' The DoSomethingUseful method represents the normal
   ' processing of the server object. At some point during
   ' normal
   ' processing, the server object decides that it is time to
   ' send a message to the client(s).

Public Sub DoSomethingUseful( )
         ' ...
         ' Some processing has led up to a decision to send a
         ' message. However, do so only if a delegate has been
         ' registered.
   If Not (m_delegateHolder Is Nothing) Then
         ' The delegate object's Invoke method invokes the
         ' methods wrapped by the delegates represented by
         ' the given delegate.
      m_delegateHolder.Invoke("This is the msg parameter.")
   End If
         ' ...
End Sub
End Class ' ServerClass

   ' This class is defined in the client component.

Public Class ClientClass
   ' This is the callback method. It will handle messages
   ' received from the server class.

Public Sub HandleMessage(ByVal msg As String)
   Console.WriteLine(msg)
End Sub
   ' This method represents the normal processing of theclient
   ' object. As some point during normal processing,the client
   ' object creates an instance of the server class and passes
   ' it
   ' a delegate wrapper to the HandleMessage method.

Public Sub DoSomethingUseful( )
   ' ...
   Dim server As New ServerClass( )
   ' The AddressOf operator in the following initialization
   ' is a little misleading to read. It's not returning an
   ' address at all. Rather, a delegate instance is being
   ' created and assigned to the myDelegate variable.

Dim myDelegate As ServerClass.MessageDelegate _
= AddressOf HandleMessage
server.RegisterForMessages(myDelegate)
   ' ...
   ' This represents other calls to the server object, which
```

```
            ' might somehow trigger the server object to call back
            ' to
            ' the client object.
         server.DoSomethingUseful( )
            ' ...
            ' At some point, the client may decide that it doesn't
            ' want
            ' any more callbacks.
         server.UnregisterForMessages(myDelegate)
      End Sub
   End Class ' ClientClass
```

Delegates are central to event handling in the .NET Framework. See the next section for more information.
Programming Visual Basic .NET
Delegates don't provide any capabilities that can't be achieved in other ways. For example, the solution in Example 2-13 could have been achieved in at least two ways that don't involve delegates:

- The server component could define an abstract base class defining the method to be implemented by the client. The client would then define a class that inherits from the server's abstract base class, providing an implementation for the class's one method. The server would then provide methods for registering and unregistering objects derived from the abstract base class.
- The server component could define an interface that includes the definition of the method to be implemented by the client. The client would then define a class that implemented this interface, and the server would provide methods for registering and unregistering objects that expose the given interface. Any of these methods (including delegates) could be a reasonable solution to a given problem. Choose the one that seems to fit best. Delegates are sometimes characterized as *safe function pointers*. I don't think that this characterization aids the learning process, because delegates aren't any sort of pointer—safe or otherwise. They are objects that encapsulate method access. Delegate objects can invoke methods without knowing where the actual methods are implemented. In effect, this allows individual methods to be treated in a polymorphic way

2.16- 2.20 Check Your Progress
Fill in the blanks
1. …………….definitions can include properties, fields, methods, and constructor.
2. ………….. is the process of taking a reference to a value type and copying the referenced value into an actual value type.
3. An ………………….. is a type whose values are explicitly named by the creator of the type.
4. A ………….. is a programmer-defined type that abstracts the ability to call a method.
5. Delegates are the comparable to ………………….. in C++.

2.21 EVENTS

An *event* is a callback mechanism. With it, objects can notify users that something interesting has happened. If desired, data can be passed from the object to the client as part of the notification.Throughout this section, I use the terms *event producer*, *producer class*, and *producer object* to talk about a class (and its instances) capable of raising events. I use the terms *event consumer*, *consumer class*, and *consumer object* to talk about a class (and its instances) capable of receiving and acting on events raised by an event producer. Here is a class that exposes an event:

```
Public Class EventProducer
Public Event SomeEvent( )
   Public Sub DoSomething( )
   ' ...
   RaiseEvent SomeEvent( )
   ' ...
   End Sub
End Class
```

The Event statement in this code fragment declares that this class is capable of raising an event called SomeEvent. The empty parentheses in the declaration indicate that the event will not pass any data. An example later in this section will show how to define events that pass data. The RaiseEvent statement in the DoSomething method raises the event. Any clients of the object that have registered their desire to receive this event will receive it at this time. Receiving an event means that a method will be called on the client to handle the event. Here is the definition of a client class that receives and handles events from the EventProducer class:

```
Public Class EventConsumer
    Private WithEvents producer As EventProducer
    Public Sub producer_SomeEvent( ) Handles producer.SomeEvent
        C onsole.WriteLine("Hey, an event happened!!")
    End Sub

    Public Sub New( )
        producer() = New EventProducer( )
    End Sub

    Public Sub DoSomething( )
        ' ...
        producer().DoSomething( )
        ' ...
    End Sub
End Class
```

The key aspects here are:

❖ The consumer object has a field that contains a reference to the producer object.
❖ The consumer object has a method capable of handling the event. A method is capable of handling an event if the method and event have the same signature. The name of the method is not important.
❖ The handler-method declaration has a *handles clause*.
❖ The handles clause specifies the event to be handled. The identifier before the dot indicates the field with the object to generate events. The identifier after the dot indicates the name of the event. The handler method is called synchronously, which means that the statement following the RaiseEvent statement in the event producer does not execute until after the method handler in the consumer completes. If an event has multiple consumers, each consumer's event handler is called in succession. The order in which the multiple consumers are called is not specified.

Here's a class that exposes an event with parameters:

```
Public Class EventProducer
    Public Event AnotherEvent(ByVal MyData As Integer)
    Public Sub DoSomething( )
        ' ...
    RaiseEvent AnotherEvent(42)
        ' ...
    End Sub
End Class
```

And here's a class that consumes it:

```
Public Class EventConsumer
    Private WithEvents producer As EventProducer
    Public Sub New( )
        +producer = New EventProducer( )
    End Sub
    Public Sub producer_AnotherEvent(ByVal MyData As Integer) _
        Handles producer.AnotherEvent
        Console.WriteLine("Received the 'AnotherEvent' event.")
        Console.WriteLine("The value of MyData is {0}.", Format(MyData))
    End Sub
    Public Sub DoSomething( )
        ' ...
        producer().DoSomething( )
        ' ...
    End Sub
End Class
```

The result of calling the EventConsumer class's DoSomething method is:

```
Received the 'AnotherEvent' event.
The value of MyData is 42.
```

2.21.1 Using Events and Delegates Together

Under the covers, .NET uses delegates as part of its events architecture. Delegates are necessary in this architecture because they enable hooking up the consumer class's event-handler method to the event producer (recall that delegates encapsulate method invocation). The Visual Basic .NET compiler hides the details of this mechanism, quietly creating delegates as needed under the surface. However, the programmer is free to make this process explicit. The following definition of the EventProducer class is semantically equivalent to the previous one:

```
Public Class EventProducer
    Public Delegate Sub SomeDelegate(ByVal MyData As Integer)
    Public Event AnotherEvent As SomeDelegate
    Public Sub DoSomething( )
        ' ...
        RaiseEvent AnotherEvent(42)
        ' ...
    End Sub
End Class
```

Note here that the declaration of SomeDelegate defines a delegate capable of wrapping any subroutine whose signature matches the signature given in the declaration. The subsequent declaration of AnotherEvent defines an event that will use the signature defined by SomeDelegate.Regardless of which syntax is being used, events are actually fields whose type is some delegate type. Variations in syntax are possible on the consumer side, too. When the WithEvents and Handles keywords are used, Visual Basic .NET creates a delegate that wraps the given handler method and then registers that delegate with the object and event given in the Handles clause. The WithEvents and Handles keywords can be omitted, and the delegate declaration and hookup can be done explicitly, as shown here:

```
Public Class EventConsumer
    Private producer As EventProducer
    Public Sub New( )
        producer = New EventProducer( )
        AddHandler producer.AnotherEvent, _
        NewEventProducer.SomeDelegate(AddressOf producer_AnotherEvent)
    End Sub
    Public Sub producer_AnotherEvent(ByVal MyData As Integer)
        Console.WriteLine("Received the 'AnotherEvent' event.")
        Console.WriteLine("The value of MyData is {0}.", Format(MyData))
    End Sub
    Public Sub DoSomething( )
        ' ...
        producer.DoSomething( )
        ' ...
    End Sub
End Class
```

The hookup of the handler method to the event producer is done with this statement in the EventConsumer class's constructor:

AddHandler producer.AnotherEvent, _New EventProducer.SomeDelegate(AddressOf producer_AnotherEvent)
The AddHandler statement and its companion, the RemoveHandler statement, allow event handlers to be dynamically registered and unregistered. The RemoveHandler statement takes exactly the same parameters as the AddHandler statement.

2.22 STANDARD MODULES

A *standard module* is a type declaration. It is introduced with the Module statement, as shown here:

```
Public Module ModuleTest
    ' ...
End Module
```

Don't confuse the Visual Basic .NET term, *standard module*, with the .NET term, *module*. They are unrelated to each other. Later we see for information about .NET modules. Standard module definitions are similar to class definitions, with these differences:

1. Standard module members are implicitly shared.
2. Standard modules cannot be inherited.
3. The members in a standard module can be referenced without being qualified with the standard module name. Standard modules are a good place to put global variables and procedures that aren't logically associated with any class.

2.23 ATTRIBUTES

An *attribute* is a program element that modifies some declaration. Here is a simple example:

```
<SomeAttribute( )> Public Class SomeClass
    ' ...
End Class
```

This example shows a fictitious SomeAttribute attribute that applies to a class declaration.Attributes appear within angle brackets (<>) and are following by parentheses (( )), which may contain a list of arguments. To apply multiple attributes to a single declaration, separate them with commas within a single set of angle brackets, like this:

```
<SomeAttribute(), SomeOtherAttribute( )> Public Class SomeClass
    ' ...
End Class
```

Attributes can be placed on the following kinds of declarations:

Types

This includes classes, delegates, enumerations, events, interfaces, Visual Basic .NET standard modules, and structures. The attribute is placed at the beginning of the first line of the type declaration:

```
<SomeAttribute( )> Public Class SomeClass
    ' ...
End Class
```

Constructors

The attribute is placed at the beginning of the first line of the constructor declaration:

```
<SomeAttribute()> Public Sub New( )
    ' ...
End Sub
```

Fields

The attribute is placed at the beginning of the field declaration:

```
<SomeAttribute( )> Public SomeField As Integer
```

Methods

The attribute is placed at the beginning of the first line of the method declaration:

```
<SomeAttribute()> Public Sub SomeMethod( )
    ' ...
End Sub
```

Parameters

The attribute is placed immediately prior to the parameter declaration. Each parameter can have its own attributes:

```
Public Sub SomeMethod(<SomeAttribute( )> ByVal SomeParameter As Integer)
```

Properties

An attribute that applies to a property is placed at the beginning of the first line of the property declaration. An attribute that applies specifically to one or both of a property's Get or Set methods is placed at the beginning of the first line of the respective method declaration:

```
<SomeAttribute()> Public Property SomeProperty( ) As Integer
    Get
    ' ...
End Get
    <SomeOtherAttribute( )> Set(ByVal Value As Integer)
    ' ...
    End Set
End Property
```

Return values

The attribute is placed after the As keyword and before the type name:

```
Public Function SomeFunction() As <SomeAttribute( )> Integer
    ' ...
End Function
```

Assemblies

The attribute is placed at the top of the Visual Basic .NET source file, following any Imports statements and preceding any declarations. The attribute must be qualified with the Assembly keyword so that the compiler knows to apply the attribute to the assembly rather than the module. Assemblies and modules are explained later.

```
Imports ...
    <Assembly: SomeAttribute( )>
Public Class ...
```

Modules

The attribute is placed at the top of the Visual Basic .NET source file, following any Imports statements and preceding any declarations. The attribute must be qualified with the Module keyword so that the compiler knows to apply the attribute to the module rather than the assembly. Assemblies and modules are explained later.

```
Imports ...
<Module: SomeAttribute( )>
Public Class ...
```

Some attributes are usable only on a subset of this list. The .NET Framework supplies several standard attributes. For example, the Obsolete attribute provides an indication that the flagged declaration should not be used in new code. This allows component developers to leave obsolete declarations in the component for backward compatibility,while still providing a hint to component users that certain declarations should no longer be used. Here's an example:

```
<Obsolete("Use ISomeInterface2 instead.")> Public Interface ISomeInterface
    ' ...
End Interface
```

When this code is compiled, the Obsolete attribute and the associated message are compiled into the application. Tools or other code can make use of this information. For example, if the compiled application is a code library referenced by some project in Visual Studio .NET, Visual Studio .NET warns the developer when she tries to make use of any items that are flagged as Obsolete. Using the previous example, if the developer declares a class that implements ISomeInterface, Visual Studio .NET displays the following warning:Obsolete: Use ISomeInterface2 instead.

2.23.1 Creating Custom Attributes

The attribute mechanism is extensible. A new attribute is defined by declaring a class that derives from the Attribute type (in the System namespace) and that provides an indication of what declarations the attribute should be allowed to modify. Here's an example:

```
<AttributeUsage(AttributeTargets.All)> Public Class SomeAttribute
    Inherits System.Attribute
End Class
```

This code defines an attribute called SomeAttribute. The SomeAttribute class itself is modified bythe AttributeUsage attribute. The AttributeUsage attribute is a standard .NET Framework attribute that indicates which declarations can be modified by the new attribute. In this case, the value of AttributeTargets.All indicates that the SomeAttribute attribute can be applied to any and all declarations. The argument of the AttributeUsage attribute is of type AttributeTargets (defined in the System namespace). The values in this enumeration are:

- Assembly,
- Module,
- Class,
- Struct,
- Enum,
- Constructor,
- Method,
- Property,
- Field,
- Event,
- Interface,
- Parameter,
- Delegate,
- ReturnValue, and All.

To create an attribute that takes one or more arguments, add a parameterized constructor to the attribute class. Here's an example:

```
<AttributeUsage(AttributeTargets.Method)>_

PublicClass MethodDocumentationAttribute
    Inherits System.Attribute

    Public ReadOnly Author As String
    Public ReadOnly Description As String
    Public Sub New(ByVal Author As String, ByVal Description As String)
        Me.Author = Author
        Me.Description = Description
    End Sub
End Class
```

This code defines an attribute that takes two parameters: *Author* and *Description*. It could be used to modify a method declaration like this:

```
<MethodDocumentation("Dave Grundgeiger", "This is my method.")> _
    Public Sub SomeMethod( )
        ' ...
    End Sub
```

By convention, attribute names end with the word Attribute. Visual Basic .NET references attributes either by their full names—for example, MethodDocumentationAttribute—or by their names less the trailing Attribute—for example, MethodDocumentation. Attributes whose names do not end with the word Attribute are simply referenced by their full names.

2.23.2 Reading Attributes

Compiled applications can be programmatically examined to determine what attributes, if any, are associated with the applications' various declarations. For example, it is possible to write a Visual Basic .NET program that searches a compiled component for the Obsolete attribute and produces a report. This is done by using the .NET Framework's *reflection* capability. Reflection is the ability to programmatically examine type information. The .NET Framework provides a great deal of support for reflection in the Type class (in the System namespace) and in the types found in the System.Reflection namespace. Reflection deserves a book of its own, but here's a brief look to get you started:

```
Imports System
Imports System.Reflection
' ...
Dim typ As Type = GetType(System.Data.SqlClient.SqlConnection)
Dim objs( ) As Object = typ.GetCustomAttributes(False)
Dim obj As Object
For Each obj In objs
    Console.WriteLine(obj.GetType( ).FullName)
Next
```

This code fragment does the following:

- ❖ Uses the *GetType* function to get a Type object that represents the SqlConnection type (defined in the System.Data.SqlClient namespace). You can experiment with putting any type name here (including the types that you create). I chose SqlConnection because I know that it happens to have an attribute associated with it.

- ❖ Calls the GetCustomAttributes method of the Type object to get an array of objects that represent the attributes associated with the type. Each object in the array represents an attribute.

- ❖ Loops through the object array and prints the type name of each object. The type name is the name of the attribute. The output is shown here:

System.ComponentModel.DefaultEventAttribute Reflection is not discussed further in this book. Review the .NET documentation for the System.Reflection namespace for more information.

2.24 CONDITIONAL COMPILATION

Conditional compilation is the ability to specify that a certain block of code will be compiled into the application only under certain conditions. Conditional compilation uses precompiler directives to affect which lines are included in the compilation process. This feature is often used to wrap code used only for debugging. For example:

```
#Const DEBUG = True
Public Sub SomeMethod( )
#If DEBUG Then
Console.WriteLine("Entering SomeMethod( )")
#End If
```

```
' ...
#If DEBUG Then
Console.WriteLine("Exiting SomeMethod( )")
#End If
End Sub
```

The #Const directive defines a symbolic constant for the compiler. This constant is later referenced in the #If directives. If the constant evaluates to True, the statements within the #If block are compiled into the application. If the constant evaluates to False, the statements within the #If block are ignored. The scope of constants defined by the #Const directive is the source file in which the directive appears. However, if the constant is referenced prior to the definition, its value is Nothing. It is therefore best to define constants near the top of the file. Alternatively, compiler constants can be defined on the command line or within the Visual Studio .NET IDE. If you're compiling from the command line, use the /define compiler switch, like this:
vbc MySource.vb /define:DEBUG=TrueYou can set multiple constants within a single /define switch by separating the *symbol=value* pairs with commas, like this:vbc MySource.vb /define:DEBUG=True,SOMECONSTANT=42

To assign compiler constants in Visual Studio .NET:

1. Right-click on the project name in the Solution Explorer window and choose Properties. This brings up the Project Property Pages dialog box. (If the Solution Explorer window is not visible,choose View Solution Explorer from the Visual Studio .NET main menu to make it appear.)

2. Within the Project Property Pages dialog box, choose the Configuration Properties folder.
   Within that folder, choose the Build property page. This causes the configuration build options to appear on the right side of the dialog box.

3. Add values to the Custom constants text box on the right side of the dialog box.

2.21 - 2.24 Check Your Progress

Fill in the blanks.
1. An ……………….. is a callback mechanism.
2. A ……………… is a type declaration.
3. An …………….. is a program element that modifies declaration.
4. ……………… is the ability to specify that a certain block of code.

2.25 SUMMARY

This chapter provided an overview of the syntax of the Visual Basic .NET language. In Chapter 3,
you'll learn about the .NET Framework—an integral part of developing in any .NET language.Subsequent chapters will teach you how to accomplish specific programming tasks in VisualBasic .NET.

2.26 CHECK YOUR PROGRESS- *ANSWERS*

2.1 - 2.4
1. aspx
2. Namespace
3. Reserved
4. Literals

2.5 - 2.9
1. True
2. True
3. True
4. False
5. True

2.10 - 2.13
1. Visibility of identifiers
2. Access modifiers
3. Private
4. TypeOf
5. Unary

2.14 - 2.15
1. False
2. True
3. False
4. True
5. True

2.16 - 2.20
1. Structure
2. Unboxing
3. Enumerations
4. Delegate
5. Function Pointer

2.21 - 2.24
1. Event
2. Standard modules
3. Attribute
4. Conditional compilation

2.27 QUESTIONS FOR SELF-STUDY

1. Write short not on literals?
2. Explain branching statement?
3. Write short note on constructors & inheritance?
4. Explain Boxing & Unboxing?
5. Explain Enumerations?
6. What is the use of attributes in .Net
7. Compare delegate with function pointers of C++

2.28 SUGGESTED READINGS

1. Visual Basic .NET Black Book by Steven Holzner

References

Programming Visual Basic .NET by Dave Grundgeiger

◈◈◈

NOTES

NOTES

CHAPTER 3

THE .NET FRAMEWORK

3.0 OBJECTIVES

After you study The. NET Framework chapter you will able to :

- describe common language Infrastructure (CLI) & common language Runtime (CLR)
- explain common type systems (CTS), CLS.
- describe Modules & Assemblies, Application domains, common language specification (CLS) Intermediate Language & Just In Time Compilation, Metadata
- you will learn & understand Memary management & Garbeg collection. A bric tour of the. Net Framework Namespaces, Configuration

3.1 INTRODUCTION

The .NET Framework is the next iteration of Microsoft's platform for developing component-based software. It provides fundamental advances in runtime services for application software. It also supports development of applications that can be free of dependencies on hardware, operating system, and language compiler. This chapter provides an overview of the architecture of the .NET Framework and describes the base features found in the core of its class library. The .NET Framework is a software framework developed by Microsoft that runs primarily on Microsoft Windows. It includes a large library and provides language interoperability (each language can use code written in other languages) across several programming languages. Programs written for the .NET Framework execute in a software environment, known as the Common Language Runtime (CLR), an application virtual machine that provides services- such as security, memory management, and exception handling. The class library and the CLR together constitute the .NET Framework.

3.2 COMMON LANGUAGE INFRASTRUCTURE (CLI) AND COMMON LANGUAGE RUNTIME (CLR)

At the heart of the .NET Framework is a new mechanism for loading and running programs and
managing their interactions. This mechanism is described in the *Common Language Infrastructure* (CLI), a specification for a runtime environment that allows software components to:

- Pass data between each other without regard to the programming language in which each component is written

- Execute on different operating systems and on different hardware platforms without having to recompile the high-level source code (a low-level compilation still automatically occurs on the target platform, as will be discussed in this chapter) Although the CLI specification was created by Microsoft, it has since been submitted to the ECMA standards organization (http://www.ecma.ch), which now has responsibility and control over it.

The CLI is just a specification—it has to be implemented in order to be useful. An implementation of the CLI is known as a *Common Language Runtime* (CLR). Microsoft's CLR implementation on the Windows platform is not under ECMA's control, but it is Microsoft's intention that the CLR be a fully compliant implementation of the CLI. As of this writing, the CLI has not been implemented on non- Windows platforms, but Microsoft and others have announced intentions to do so. The CLI specifies how executable code is loaded, run, and managed. The portion of the CLR that performs the tasks of loading, running, and managing .NET applications is called the *virtual execution system* (VES). Code run by the VES is called *managed code* . The CLI greatly expands upon concepts that exist in Microsoft's *Component Object Model* (COM). As its core feature, COM specifies how object interfaces are laid out in memory. Any component that can create and consume this layout can share data with other components that do the same. COM was a big step forward when it was introduced (circa 1992), but it has its shortcomings. For example, in spite of its name, COM actually has no concept of an object—only object interfaces. Therefore, COM can't support passing native types from one component to another.

3.3 COMMON TYPE SYSTEM (CTS)

The CLI specification defines a rich type system that far surpasses COM's capabilities. It's called the *Common Type System* (CTS). The CTS defines at the runtime level how types are declared and used. Previously, language compilers controlled the creation and usage of types, including their layout in memory. This led to problems when a component written in one language tried to pass data to a component written in a different language. Anyone who has written Visual Basic 6 code to call Windows API functions, for instance, or who has tried to pass a JavaScript array to a component written either in Visual Basic 6 or C++, is aware of this problem. It was up to the developer to translate the data to be understandable to the receiving component. The CTS obliterates this problem by providing the following features:

- Primitive types (Integer, String, etc.) are defined at the runtime level. Components can easily pass instances of primitive types between each other because they all agree on how that data is formatted.

- Complex types (structures, classes, enumerations, etc.) are constructed in a way that is defined at the runtime level. Components can easily pass instances of complex types between each other because they all agree on how complex types are constructed from primitive types.

- All types carry rich type information with them, meaning that a component that is handed an object can find out the definition of the type of which the object is an instance. This is analogous to type libraries in COM, but the CTS is different because the type information is much richer and is guaranteed to be present.

3.3.1 Namespaces

Namespaces were introduced in Chapter 2 as a way to group related types. They are mentioned
again here because they aren't just a Visual Basic .NET concept; they are also used by the CLR and by other languages that target the .NET platform. It's important to keep in mind that to the CLR, a namespace is just part of a fully qualified type name, and nothing more. See Section 3.4.2 later in this chapter for more information.

3.4 PORTIONS OF THE CLI

The CLI specification recognizes that the CLR can't be implemented to the same extent on all platforms. For example, the version of the CLR implemented on a cell phone won't be as versatile as the one implemented on Windows 2000 or Windows XP. To address this issue, the CLI defines a set of *libraries*. Each library contains a set of classes that implement a certain portion of the CLI's functionality. Further, the CLI defines *profiles*. A profile is a set of libraries that must be implemented on a given platform. The libraries defined by the CLI are:

Runtime Infrastructure Library
This library provides the core services that are needed to compile and run an application that targets the CLI.

Base Class Library
This library provides the runtime services that are needed by most modern programming languages. Among other things, the primitive data types are defined in this library.

Network Library
This library provides simple networking services.

Reflection Library
This library provides the ability to examine type information at runtime and to invoke members of types by supplying the member name at runtime, rather than at compile time.

XML Library
This library provides a simple XML parser.

Floating Point Library
This library provides support for floating point types and operations.

Extended Array Library
This library provides support for multidimensional arrays. The profiles defined by the CLI at this time are:

Kernel Profile
This profile defines the minimal functionality of any system claimed as an implementation of the CLI. CLRs that conform to the Kernel Profile must implement the Base Class Library and the Runtime Infrastructure Library.

Compact Profile
This profile includes the functionality of the Kernel Profile, plus the Network Library, the Reflection Library, and the XML Library. It is intended that an implementation of the Compact Profile can be lightweight, yet provide enough functionality to be useful. Additional profiles will be defined in future versions of the CLI specification. Any given implementation of the CLI is free to implement more than the functionality specified by these minimal profiles. For example, a given

implementation could support the Compact Profile but also support the Floating Point Library. The .NET Framework on Windows 2000 supports all the CLI libraries, plus additional libraries not defined by the CLI. Note that the CLI does not include such major class libraries as Windows Forms, ASP.NET, and ADO.NET. These are Microsoft-specific class libraries for developing applications on Windows platforms. Applications that depend on these libraries will not be portable to other implementations of the CLI unless Microsoft makes those class libraries available on those other implementations.

3.1 - 3.4 Check Your Progress

Fill in the blanks

1. An implementation of the CLI is known as a ……………………….
2. ……………….……….. defines at the runtime level how types are declared and used.
3. …..………………… library provides the core service that are needed to compile and run an application that targets the CLI.
4. JIT compiler is do compilation at ……………………..
5. CLR gets the ………………………. code to be run.

3.5 MODULES AND ASSEMBLIES

A *module* is an *.exe* or *.dll* file. An *assembly* is a set of one or more modules that together make up an application. If the application is fully contained in an *.exe* file, fine—that's a one-module assembly. If the *.exe* is always deployed with two *.dll* files and one thinks of all three files as comprising an inseparable unit, then the three modules together form an assembly, but none of them does so by itself. If the product is a class library that exists in a *.dll* file, then that single *.dll* file is an assembly. To put it in Microsoft's terms, the assembly is the unit of deployment in .NET.

An assembly is more than just an abstract way to think about sets of modules. When an assembly is deployed, one (and only one) of the modules in the assembly must contain the *assembly manifest*, which contains information about the assembly as a whole, including the list of modules contained in the assembly, the version of the assembly, its culture, etc. The command-line compiler and the Visual Studio .NET compiler create single-module assemblies by default. Multiple-module assemblies are not used in this book. Assembly boundaries affect type resolution. When a source file is compiled, the compiler must resolve type names used in the file to the types' definitions. For types that are defined in the same source project, the compiler gets the definitions from the code it is compiling. For types that are defined elsewhere, the compiler must be told where to find the definitions. This is done by referencing the assemblies that contain the compiled type definitions. When the command-line compiler is used, the /reference switch identifies assemblies containing types used in the project being compiled. An assembly has the same name as the module that contains the assembly manifest, except for the file extension. In some cases, however, an assembly is specified by giving the full name of the module that contains the assembly manifest. For example, to compile an application that uses the system.Drawing.Point class, you could use the following command line:

```
vbc MySource.vb /reference:System.Drawing.dll
```

The documentation for the command-line compiler states that the argument to the reference switch is an assembly. This is not quite accurate. The argument is the name of the module that contains the assembly manifest for an assembly. If more than one assembly needs to be referenced, you can list them all in the same /reference switch, separated by commas, like this:

```
vbcMySource.vb /reference:System.Drawing.dll,System.Windows.Forms.dll
```

The Visual Basic .NET command-line compiler automatically references two assemblies:

mscorlib.dll, which contains most of the types found in the System namespace; *Microsoft.VisualBasic.dll* which contains the types found in the Microsoft.VisualBasic namespace.

When you're working within the Visual Studio .NET IDE, external assemblies are referenced by doing the following:

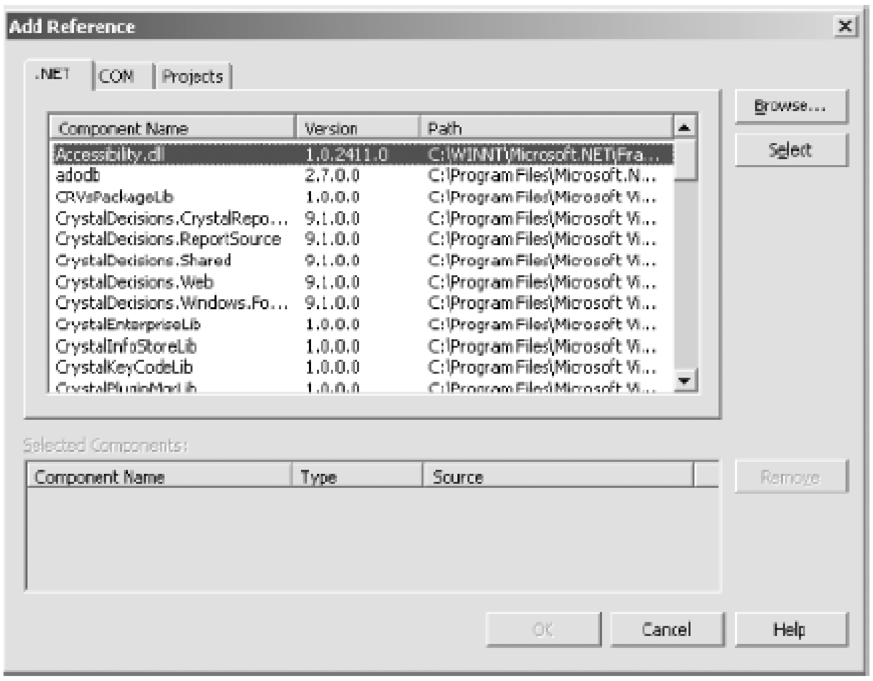
1. In the Solution Explorer window, right-click on References, then click on Add Reference. The Add Reference dialog box appears, as shown in Figure 3-1.

2. Scroll down to find the desired assembly.

3. Double-click or highlight the assembly name, and press the Select button. The assembly name appears in the Selected Components frame of the dialog box.

4. Select additional assemblies, or click OK.

Figure 3-1. The Add Reference dialog box



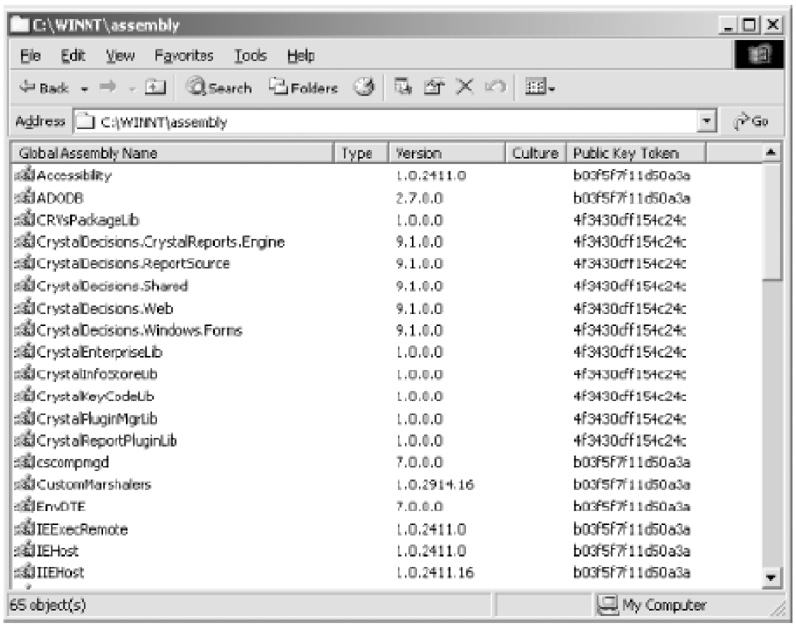
Component Name	Version	Path
Accessibility.dll	1.0.2411.0	C:\WINNT\Microsoft.NET\Fra...
adodb	2.7.0.0	C:\Program Files\Microsoft.N...
CRVsPackageLib	1.0.0.0	C:\Program Files\Microsoft Vi...
CrystalDecisions.CrystalRepo...	9.1.0.0	C:\Program Files\Microsoft Vi...
CrystalDecisions.ReportSource	9.1.0.0	C:\Program Files\Microsoft Vi...
CrystalDecisions.Shared	9.1.0.0	C:\Program Files\Microsoft Vi...
CrystalDecisions.Web	9.1.0.0	C:\Program Files\Microsoft Vi...
CrystalDecisions.Windows.Fo...	9.1.0.0	C:\Program Files\Microsoft Vi...
CrystalEnterpriseLib	1.0.0.0	C:\Program Files\Microsoft Vi...
CrystalInfoStoreLib	1.0.0.0	C:\Program Files\Microsoft Vi...
CrystalKeyCodeLib	1.0.0.0	C:\Program Files\Microsoft Vi...
CrystalPluginMgrLib	1.0.0.0	C:\Program Files\Microsoft Vi...

Component Name	Type	Source

3.5.1 Global Assembly Cache (GAC)

By default, assemblies are not shared. When one assembly is dependent on another, the two assemblies are typically deployed into a single application directory. This makes it easy to install and remove an application. To install an application, simply create the application directory and copy the files into it. To delete the application, just delete the application directory. The Windows Registry is not used at all.If an assembly must be shared among more than one program, either it can be copied into each appropriate application directory or it can be installed into the *global assembly cache* (GAC). The GAC is an area on disk (typically, it's the *assembly* subdirectory of the Windows directory) that holds assemblies to be shared among all applications. All of the .NET Framework assemblies reside in the GAC. (See Figure 3-2 for a partial view of the assemblies in a typical GAC.) Placing an assembly into the GAC should be avoided if possible: it makes application installation and removal more difficult. This is because the Windows Installer or *gacutil.exe* must be used to manipulate the GAC—you can no longer simply copy or remove the application directory. Installing assemblies into the GAC is not covered in this book. For information, point your browser to http://msdn.microsoft.com and perform a search for "Deploying Shared Components.

" ***Figure 3-2. Partial view of a typical GAC***


C:\WINNT\assembly

Address C:\WINNT\assembly

Global Assembly Name	Type	Version	Culture	Public Key Token
Accessibility		1.0.2411.0		b03f5f7f11d50a3a
ADODB		2.7.0.0		b03f5f7f11d50a3a
CRVsPackageLib		1.0.0.0		4f3430cff154c24c
CrystalDecisions.CrystalReports.Engine		9.1.0.0		4f3430cff154c24c
CrystalDecisions.ReportSource		9.1.0.0		4f3430cff154c24c
CrystalDecisions.Shared		9.1.0.0		4f3430cff154c24c
CrystalDecisions.Web		9.1.0.0		4f3430cff154c24c
CrystalDecisions.Windows.Forms		9.1.0.0		4f3430cff154c24c
CrystalEnterpriseLib		1.0.0.0		4f3430cff154c24c
CrystalInfoStoreLib		1.0.0.0		4f3430cff154c24c
CrystalKeyCodeLib		1.0.0.0		4f3430cff154c24c
CrystalPluginMgrLib		1.0.0.0		4f3430cff154c24c
CrystalReportPluginLib		1.0.0.0		4f3430cff154c24c
cscompmgd		7.0.0.0		b03f5f7f11d50a3a
CustomMarshalers		1.0.2914.16		b03f5f7f11d50a3a
EnvDTE		7.0.0.0		b03f5f7f11d50a3a
IEExecRemote		1.0.2411.0		b03f5f7f11d50a3a
IEHost		1.0.2411.0		b03f5f7f11d50a3a
IIEHost		1.0.2411.16		b03f5f7f11d50a3a

65 object(s)

3.5.2 Comparison of Assemblies, Modules, and Namespaces

It's easy to confuse the three concepts of *namespace*, *module*, and *assembly*. Here is a recap:

Namespace

A portion of a type name. Specifically, it is the portion that precedes the final period in a fully qualified type name.

Module

A file that contains executable code (*.exe* or *.dll*).

Assembly

A set of one or more modules that are deployed as a unit. The assembly name is the same as the name of the module that contains the assembly manifest, minus the filename extension.Depending on how things are named, it can seem like these three terms are interchangeable. For example, *System.Drawing.dll* is the name of a module that is deployed as part of the .NET Framework.

As it happens, this module is part of a single-module assembly. Because assemblies are named after the module that contains the assembly manifest, the assembly is called System.Drawing. A compiler will reference this assembly as *System.Drawing.dll*. Many (but not all) of the types in this assembly have a namespace of System.Drawing. (Other types in the System.Drawing assembly have namespaces of System.Drawing.Design, System.Drawing.Drawing2D, System.Drawing.Imaging, System.Drawing.Printing, and System.Drawing.Text.) Note that even though the namespace, module, and assembly are similarly named in this case, they are distinct concepts. Note in particular that importing a namespace and referencing an assembly are different operations with different purposes. The statement:
Imports System.Drawing allows the developer to avoid typing the fully qualified names of the types in the System.Drawing namespace. It does *not* reference the assembly in which those types are defined. To use the types, the System.Drawing assembly (contained in the *System.Drawing.dll* module) must be referenced as described earlier in this section. The Imports statement was introduced in Chapter 2. In other cases, namespace and assembly names *don't* correspond. One example is the System namespace. Some types with this namespace are found in the mscorlib assembly, and others are found in the System assembly. In addition, each of those assemblies has types with other namespaces. For example, the System assembly contains types with the Microsoft. VisualBasic namespace, even though most of the types with that namespace are found in the Microsoft.VisualBasic assembly. (The reason for this apparent inconsistency is actually quite sound. Namespaces group types according to functionality, while assemblies tend to group types according to which types are most likely to be used together. This improves performance because it minimizes the number of assemblies that have to be loaded at runtime.)

When thinking about namespaces, just remember that types can have any namespace at all, regardless of where they're defined—the namespace is just part of the type name.

3.6 APPLICATION DOMAINS

Application domains are to the CLR what processes are to an operating system. It may be surprising to note that the CLR can run multiple .NET applications within a single process, without any contention or security difficulties. Because the CLR has complete control over loading and executing programs, and because of the presence of type information, the CLR guarantees that .NET applications cannot read or write each other's memory, even when running in the same process. Because there is less performance overhead in switching between application domains than in switching between processes, this provides a performance gain. This is especially beneficial to web applications running in Internet Information Services (IIS), where scalability is an issue.

3.7 COMMON LANGUAGE SPECIFICATION (CLS)

The CLI defines a runtime that is capable of supporting most, if not all, of the features found in modern programming languages. It is not intended that all languages that target the CLR will support all CLR features. This could cause problems when components written in different languages attempt to interoperate. The CLI therefore defines a subset of features that are considered compatible across language boundaries. This subset is called the *Common Language Specification* (CLS). Vendors creating components for use by others need to ensure that all externally visible constructs (e.g., public types, public and protected methods, parameters on public and protected methods, etc.) are CLS-compliant. This ensures that their components will be usable within a broad array of languages, including Visual Basic .NET. Developers authoring components in Visual Basic .NET have an easy job because all Visual Basic .NET code is CLS-compliant (unless the developer explicitly exposes a public or protected type member or method parameter that is of a non-CLS-compliant type).Because Visual Basic .NET automatically generates CLS-compliant components, this book does not describe the CLS rules. However, to give you a sense of the kind of thing that the CLS specifies, consider that some languages support a feature called *operator overloading* . This allows the developer to specify actions that should be taken if the standard operator symbols (+, -, *, /, =, etc.) are used on user-defined classes. Because it is not reasonable to expect that all languages should implement such a feature, the CLS has a rule about it. The rule states that if a CLS-compliant component has public types that provide overloaded operators, those types must provide access to that functionality in another way as well (usually by providing a public method that performs the same operation).

3.8 INTERMEDIATE LANGUAGE (IL) AND JUST-IN-TIME (JIT) COMPILATION

All compilers that target the CLR compile source code to *Intermediate Language* (IL), also known as *Common Intermediate Language* (CIL). IL is a machine language that is not tied to any specific machine. Microsoft designed it from scratch to support the CLI's programming concepts. The CLI specifies that all CLR implementations can compile or interpret IL on the machine on which the CLR is running. If the IL is compiled (versus interpreted), compilation can occur at either of two times:

- Immediately prior to a method in the application being executed
- At deployment time In the first case, each method is compiled only when it is actually needed. After the method is compiled, subsequent calls bypass the compilation mechanism and call the compiled code directly. The compiled code is not saved to disk, so if the application is stopped and restarted, the compilation must occur again. This is known as *just-in-time* (JIT) compilation and is the most common scenario. In the second case, the application is compiled in its entirety at deployment time. IL is saved to *.exe* and *.dll* files. When such a file containing IL is executed, the CLR knows how to invoke the JIT compiler and execute the resulting code. Note that on the Microsoft Windows platforms, IL is always compiled—never interpreted.

3.5 - 3.8 Check Your Progress

Fill in the blanks

1. An ……………………. is a set of one or more modules that together up an application.
2. The ……………………. is an area on disk that holds assemblies to be shared among applications.
3. ……………………… are to the CLR what processes are to an operating system.
4. The CLI defines a subset of features known as ……………………..
5. ……………………. is a machine language that is not tied to any specific machine.

3.9 METADATA

Source code consists of some constructs that are *procedural* in nature and others that are *declarative* in nature. An example of a procedural construct is:

```
someObject.SomeMember = 5
```

This is procedural because it compiles into executable code that performs an action at runtime. Namely, it assigns the value 5 to the SomeMember member of the someObject object.In contrast, here is a declarative construct:

```
Dim someObject As SomeClass
```

This is declarative because it doesn't perform an action. It states that the symbol someObject is a variable that holds a reference to an object of type SomeClass. In the past, declarative nformation typically was used only by the compiler and did not compile directly into the executable. In the CLR, however, declarative information is everything! The CLR uses type and signature information to ensure that memory is always referenced in a safe way. The JIT compiler uses type and signature information to resolve method calls to the appropriate target code at JIT compile time. The only way for this to work is for this declarative information to be included alongside its associated procedural information. Compilers that target the CLR therefore store both procedural and declarative information in the resulting *.exe* or *.dll* file. The procedural information is stored as IL, and the declarative information is stored as *metadata*. Metadata is just the CLI's name for declarative information. The CLI has a mechanism that allows programmers to include arbitrary metadata in compiled applications. This mechanism is known as *custom attributes* and is available in Visual Basic .NET. Custom attributes were discussed in detail in Chapter 2.

3.10 MEMORY MANAGEMENT AND GARBAGE COLLECTION

In any object-oriented programming environment, there arises the need to instantiate and destroy objects. Instantiated objects occupy memory. When objects are no longer in use, the memory they occupy should be reclaimed for use by other objects. Recognizing when objects are no longer being used is called *lifetime management*, which is not a trivial problem. The solution the CLR uses has implications for the design and use of the components you write, so it is worth understanding. In the COM world, the client of an object notified the object whenever a new object reference was passed to another client. Conversely, when any client of an object was finished with it, the client notified the object of that fact. The object kept track of how many clients had references to it. When that count dropped to zero, the object was free to delete itself (that is, give its memory back to the memory heap). This method of lifetime management is known as *reference counting*. Visual Basic programmers were not necessarily aware of this mechanism because the Visual Basic compiler automatically generated the low-level code to perform this housekeeping. C++ developers had no such luxury. Reference counting has some drawbacks:

- A method call is required every time an object reference is copied from one variable to another and every time an object reference is overwritten.

- Difficult-to-track bugs can be introduced if the reference-counting rules are not precisely followed.
- Care must be taken to ensure that circular references are specially treated (because circular references can result in objects that never go away).

The CLR mechanism for lifetime management is quite different. Reference counting is not used. Instead, the memory manager keeps a pointer to the address at which free memory (known as the *heap*) starts. To satisfy a memory request, it just hands back a copy of the pointer and then increments the pointer by the size of the request, leaving it in a position to satisfy the next memory request. This makes memory allocation very fast. No action is taken at all when an object is no longer being used. As long as the heap doesn't run out, memory is not reclaimed until the application exits. If the heap is large enough to satisfy all memory requests during program execution, this method of memory allocation is as fast as is theoretically possible, because the only overhead is incrementing the heap pointer on memory allocations. If the heap runs out of memory, there is more work to do. To satisfy a memory request when the heap is exhausted, the memory manager looks for any previously allocated memory that can be reclaimed. It does this by examining the application variables that hold object references. The objects that these variables reference (and therefore the associated memory) are considered in use because they can be reached through the program's variables. Furthermore, because the runtime has complete access to the application's type information, the memory manager knows whether the objects contain members of the memory that is in use. During this process, it consolidates the contents of all this memory into one contiguous block at the start of the heap, leaving the remainder of the heap free to satisfy new memory requests. This process of freeing up memory is known as *garbage collection* (GC), a term that also applies to this overall method of lifetime management. The portion of the memory manager that performs garbage collection is called the *garbage collector*. The benefits of garbage collection are:

- No overhead is incurred unless the heap becomes exhausted.
- It is impossible for applications to cause memory leaks.
- The application need not be careful with circular references.

Although the process of garbage collection is expensive (on the order of a fraction of a second when it occurs), Microsoft claims that the total overhead of garbage collection is on average much less than the total overhead of reference counting (as shown by their benchmarks). This, of course, is highly dependent on the exact pattern of object allocation and deallocation that occurs in any given program.

3.10.1 Finalize

Many objects require some sort of cleanup (i.e., *finalization*) when they are destroyed. An example might be a business object that maintains a connection to a database. When the object is no longer in use, its database connection should be released. The .NET Framework provides a way for objects to be notified when they are about to be released, thus permitting them to release nonmemory resources. (Memory resources held by the object can be ignored because they will be handled automatically by the garbage collector.) Here's how it works: the Object class (defined in the System namespace) has a method called Finalize that can be overridden. Its default implementation does nothing. If it is overridden in a derived class, however, the garbage collector automatically calls it on an instance of that class when that instance is about to be reclaimed. Here's an example of overriding the Finalize method:

```
Public Class SomeClass
    Protected Overrides Sub Finalize( )
    ' Release nonmanaged resources here.
    MyBase.Finalize( )
        ' Important
    End Sub
End Class
```

The Finalize method should release any nonmanaged resources that the object has allocated.*Nonmanaged resources* are any resources other than memory (for example, database connections, file handles, or other OS handles). In contrast, *managed resources* are object references. As already mentioned, it is not necessary to release managed resources in a Finalize method—the garbage collector will handle it. After releasing resources allocated by the class, the Finalize method must always call the base class's Finalize implementation so that it can release any resources allocated by base-class code. If the class is derived directly from the Object class, technically this could be omitted (because the Object class's Finalize method doesn't do nything). However, calling it doesn't hurt anything, and it's a good habit to get into. An object's Finalize method should not be called by application code. The Finalize method has special meaning to the CLR and is intended to be called only by the garbage collector. If you're familiar with *destructors* in C++, you'll recognize that the Finalize method is the identical concept. The only difference between the Finalize method and C++ destructors is that C++ destructors automatically call their base class destructors, whereas in Visual Basic .NET, the programmer must remember to put in the call to the base class's Finalize method. It is interesting to note that C#—another language on the .NET platform—actually has destructors (as C++ does), but they are automatically compiled into Finalize methods that work as described here.

3.10.2 Dispose

The downside of garbage collection and the Finalize method is the loss of *deterministic finalization* . With reference counting, finalization occurs as soon as the last reference to an object is released (this is *deterministic* because object finalization is controlled by program flow). In contrast, an object in a garbage-collected system is not destroyed until garbage collection occurs or until the application exits. This is *nondeterministic* because the program has no control over when it happens. This is a problem because an object that holds scarce resources (such as a database connection) should free those resources as soon as the object is no longer needed. If this is not done, the program may run out of such resources long before it runs out of memory.Unfortunately, no one has discovered an elegant solution to this problem. Microsoft does have a recommendation, however. Objects that hold nonmanaged resources should implement the IDisposable interface (defined in the System namespace). The IDisposable interface exposes a single method, called Dispose, which takes no parameters and returns no result. Calling it tells the object that it is no longer needed. The object should respond by releasing all the resources it holds, both managed and nonmanaged, and should call the Dispose method on any subordinate objects that also expose the IDisposable interface. In this way, scarce resources are released as soon as they are no longer needed. This solution requires that the user of an object keep track of when it is done with the object. This is often trivial, but if there are multiple users of an object, it may be difficult to know which user should call Dispose. At the time of this writing, it is simply up to the programmer to work this out. In a sense, the Dispose method is an alternate destructor to address the issue of nondeterministic finalization when nonmanaged resources are involved. However, the CLR itself never calls the Dispose method. It is up to the client of the object to call the Dispose method at the appropriate time, based on the client's knowledge of when it is done using the object. This implies responsibilities for both the class author and client author. The class author must document the presence of the Dispose method so that the client author knows that it's necessary to call it. The client author must make an effort to determine whether any given class has a Dispose method and, if so, to call it at the appropriate time. Even when a class exposes the IDisposable interface, it should still override the Finalize method, just in case the client neglects to call the Dispose method. This ensures that nonmanaged resources are eventually released, even if the client forgets to do it. A simple (but incomplete) technique would be to place a call to the object's Dispose method in its Finalize method, like this:

```
' Incomplete solution. Don't do this.
Public Sub Dispose( ) Implements IDisposable.Dispose
    ' Release resources here.
End Sub
Protected Overrides Sub Finalize( )
    Dispose( )
    MyBase.Finalize( )
End Sub
```

In this way, if the client of the object neglects to call the Dispose method, the object itself will do so when the garbage collector destroys it. Microsoft recommends that the Dispose method be written so it is not an error to call it more than once. This way, even if the client calls it at the correct time, it's OK for it to be called again in the Finalize method. If the object holds references to other objects that implement the

IDisposable interface, the code just shown may cause a problem. This is because the order of object destruction is not guaranteed. Specifically, if the Finalize method is executing, it means that garbage collection is occurring. If the object holds references to other objects, the garbage collector may have already reclaimed those other objects. If the object attempts to call the Dispose method on a reclaimed object, an error will occur. This situation exists only during the call to Finalize—if the client calls the Dispose method, subordinate objects will still be there. (They can't have been reclaimed by the garbage collector because they are reachable from the application's code.) To resolve this race condition, it is necessary to take slightly different action when finalizing than when disposing. Here is the modified code:

```
Public Sub Dispose( ) Implements IDisposable.Dispose
    DisposeManagedResources( )
    DisposeUnmanagedResources( )
End Sub

Protected Overrides Sub Finalize( )
    DisposeUnmanagedResources( )
    MyBase.Finalize( )
End Sub

Private Sub DisposeManagedResources( )
    ' Call subordinate objects' Dispose methods.
End Sub

Private Sub DisposeUnmanagedResources( )
    ' Release unmanaged resources, such as database connections.
End Sub
```

Here, the Finalize method only releases unmanaged resources. It doesn't worry about calling the Dispose method on any subordinate objects, assuming that if the subordinate objects are also unreachable, they will be reclaimed by the garbage collector and their finalizers (and hence their Dispose methods) will run. An optimization can be made to the Dispose method. When the Dispose method is called by the client, there is no longer any reason for the Finalize method to be called when the object is destroyed. Keeping track of and calling objects' Finalize methods imposes overhead on the garbage collector. To remove this overhead for an object with its Dispose method called, the Dispose method should call the SuppressFinalize shared method of the GC class, like this: Public Sub Dispose( ) Implements IDisposable.Dispose

```
DisposeManagedResources( )
DisposeUnmanagedResources( )
GC.SuppressFinalize(Me)
End Sub
```

The type designer must decide what will occur if the client attempts to use an object after calling its Dispose method. If possible, the object should automatically reacquire its resources. If this is not possible, the object should throw an exception. Example 3-1 shows the latter.

Example 3-1. A complete Finalize/Dispose example

```
Public Class SomeClass
    Implements IDisposable
        ' This member keeps track of whether the object has been disposed.
        Private disposed As Boolean = False
        ' The Dispose method releases the resources held by the object.
        ' It must be called by the client when the client no longer needs
        ' the object.
        Public Sub Dispose( ) Implements IDisposable.Dispose
            If Not disposed Then
            DisposeManagedResources( )
            DisposeUnmanagedResources( )
            GC.SuppressFinalize(Me)
            disposed = True
        End If
    End Sub
    ' The Finalize method releases nonmanaged resources in the case
    ' that the client neglected to call Dispose. Because of the
    ' SuppressFinalize call in the Dispose method, the Finalize method
    ' will be called only if the Dispose method is not called.

    Protected Overrides Sub Finalize( )
        DisposeUnmanagedResources( )
        MyBase.Finalize( )
    End Sub

    Private Sub DisposeManagedResources( )
        ' Call subordinate objects' Dispose methods.
    End Sub

    Private Sub DisposeUnmanagedResources( )
        ' Release unmanaged resources, such as database connections.
    End Sub

    Private Sub DoSomething( )
        ' Call the EnsureNotDisposed method at the top of every method that
        ' needs to access the resources held by the object.
        EnsureNotDisposed( )
        ' ...
    End Sub

    Private Sub EnsureNotDisposed( )
    ' Make sure that the object hasn't been disposed.
    ' Instead of throwing an exception, this method could be written
    ' to reacquire the resources that are needed by the object.
        If disposed Then
            Throw New ObjectDisposedException(Me.GetType( ).Name)
        End If
    End Sub
    End
Class
```

3.11 A BRIEF TOUR OF THE .NET FRAMEWORK NAMESPACES

The .NET Framework provides a huge class library—something on the order of 6,000 types. To help developers navigate though the huge hierarchy of types, Microsoft has divided them into namespaces. However, even the number of namespaces can be daunting. Here are the most common namespaces and an overview of what they contain:

Microsoft.VisualBasic

Runtime support for applications written in Visual Basic .NET. This namespace contains the functions and procedures included in the Visual Basic .NET language.

Microsoft.Win32

Types that access the Windows Registry and provide access to system events (such as low memory, changed display settings, and user logout).

System

Core system types, including:

- Implementations for Visual Basic .NET's fundamental types (see "Types" in Chapter 2 for a list of fundamental types and the .NET classes that implement them).
- Common custom attributes used throughout the .NET Framework class library (see Appendix A), as well as the Attribute class, which is the base class for most (although not all) custom attributes in .NET applications.
- Common exceptions used throughout the .NET Framework class l ibrary (see Appendix B), as well as the Exception class, which is the base class for all exceptions in .NET applications.The Array class, which is the base class from which all Visual Basic .NET arrays implicitly inherit.

- The Convert class, which contains methods for converting values between various types.
- The Enum class, from which all enumerations implicitly derive.
- The Delegate class, from which all delegates implicitly derive.
- The Math class, which has many shared methods for performing common mathematical functions (e.g., Abs, Min, Max, Sin, and Cos). This class also defines two constant fields, E and PI, that give the values of the numbers e and pi, respectively, within the precision of the Double data type.
- The Random class, for generating pseudorandom numbers.
- . The Version class, which encapsulates version information for .NET assemblies.

System.CodeDom

Types for automatically generating source code (used by tools such as the wizards in Visual Studio .NET and by the ASP.NET page compiler).

System.Collections

Types for managing collections, including:

ArrayList

Indexed like a single-dimensional array and iterated like an array, but much more flexible than an array. With an ArrayList, it is possible to add elements without having to worry about the size of the list (the list grows automatically as needed), insert and remove elements anywhere in the list, find an element's index given its value, and sort the elements in the list.

BitArray

Represents an array of bits. Each element can have a value of True or False. The BitArray class defines a number of bitwise operators that operate on the entire array at once.

Hashtable

Represents a collection of key/value pairs. Both the key and value can be any object.

Queue

Represents a queue, which is a *first-in-first-out* (FIFO) list.

SortedList

Like a Hashtable, represents a collection of key/value pairs. When enumerated, however, the items are returned in sorted key order. In addition, items can be retrieved by index, which the Hashtable cannot do. Not surprisingly, SortedList operations can be slower than comparable Hashtable operations because of the increased work that must be done to keep the structure in sorted order.

Stack

Represents a stack, which is a *last-in-first-out* (LIFO) list.
Be aware that in addition to these types, there is also the Array type, defined in the System namespace, and the Collection type, defined in the Microsoft.VisualBasic namespace. The latter is a collection type that mimics the behavior of Visual Basic 6 collection objects.

System.ComponentModel

Support for building components that can be added to Windows Forms and Web Forms.

System.Configuration

Support for reading and writing program configuration.

System.Data

Support for data access. The types in this namespace constitute ADO.NET.

System.Diagnostics

Support for debugging and tracing.

System.Drawing
Graphics-drawing support.

System.EnterpriseServices
Transaction-processing support.

System.Globalization
Internationalization support.

System.IO
Support for reading and writing streams and files.

System.Net
Support for communicating over networks, including the Internet.

System.Reflection
Support for runtime type discovery.

System.Resources
Support for reading and writing program resources.

System.Security
Support for accessing and manipulating security information.

System.ServiceProcess
Types for implementing system services.

System.Text
Types for manipulating text and strings.

Note : Particular to the StringBuilder type. When strings are built from smaller parts, the methods on the StringBuilder class are more efficient than similar methods on the String class.This is because the instances of the String class can't be modified in place; every time a change is made to a String object, a new String object is actually created to hold the new value. In contrast, the methods in StringBuilder that modify the string actually modify the string in place.*System.Threading* Support for multithreaded programming.

System.Timers
Provides the Timer class, which can generate an event at predetermined intervals. This addresses one of the limitations of the Visual Basic 6 Timer control: it had to be hosted in a container and therefore could be used only in an application with a user interface.

System.Web
Support for building web applications. The types in this namespace constitute Web Forms and
ASP.NET.

System.Windows.Forms
Support for building GUI (fat client) applications. The types in this namespace constitute Windows Forms.

System.Xml
Support for parsing, generating, transmitting, and receiving XML.

3.12 CONFIGURATION

System and application configuration is managed by XML files with a *.config* extension. Configuration files exist at both the machine and application level. There is a single machine-level configuration file, located at *runtime_install_path\CONFIG\machine.config*. For example,

C:\WINNT\Microsoft.NET\Framework\v1.0.2914\CONFIG\machine.config.

Application-configuration files are optional. When they exist, they reside in the application's root folder and are named *application_file_name.config*. For example, *myApplication.exe.config*. Web applicationconfiguration files are always named *web.config*. They can exist in the web application's root folder and in subfolders of the application. Settings in subfolders' configuration files apply only to pages retrieved from the same folder and its child folders and override settings from configuration files in higher-level folders. Configuration files should be used for all application-configuration information; the Windows Registry should no longer be used for application settings.

3.12.1 Configuration File Format

Configuration files are XML documents, where the root element is <configuration>. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<!-- More stuff goes in here. -->
</configuration>
```

To be as flexible as possible, .NET configuration files use a scheme in which the application developer can decide on the names of the subelements within the <configuration> element. This is done using the <configSections>, <section>, and <sectionGroup> elements. Example 3-2 shows how this is done using the <configSections> and <section> elements; the <sectionGroup>
element is discussed later in this section.

Example 3-2. Defining a section in a configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

    <configSections>
        <section name="mySectionName"
            type="System.Configuration.SingleTagSectionHandler" />
    </configSections>
    <mySectionName
        someSetting="SomeValue"
        anotherSetting="AnotherValue" />
</configuration>
```

The name attribute of the <section> element specifies the name of an element that will (or could) appear later in the file. The type attribute specifies the name of a *configuration section handler*, which is a class that knows how to read an XML section that's formatted in a particular way. The .NET Framework provides stock configuration section handlers (notably the SingleTagSectionHandler and the NameValueSectionHandler classes, both of which will be discussed later in this section), which are sufficient for the majority of cases. Although it's beyond the scope of this book, a custom configuration section handler can be created by writing a class that implements the IConfigurationSectionHandler interface. The SingleTagSectionHandler configuration section handler reads XML sections that are of the form:

<sectionName key1Name="Value1" key2Name="Value2" etc... />

The element can contain any number of key/value pairs. The configuration section handler class is not used directly in code. To read information from an application's configuration file, use the GetConfig method in the ConfigurationSettings class (defined in the System.Configuration namespace). The syntax of the GetConfig method is:

```
Public Shared Function GetConfig(ByVal sectionName As String) As Object
```

Here's how the mechanism works (an example will follow):

1. The application calls the GetConfig method, passing it the name of the configuration section that is to be read.

2. Internally, the GetConfig method instantiates the configuration section handler class that is appropriate for reading that section. (Recall that it is the values found in the <configSections> portion of the configuration file that identify the appropriate configuration section handler class to use.)
3. The Create method of the configuration section handler is called and is passed the XML from the requested configuration section.
4. The configuration section handler's Create method returns an object containing the values read from the configuration section.
5. The object returned from the Create method is passed back to the caller of the GetConfig method. The type of object returned from GetConfig is determined by the specific configuration section handler that handles the given configuration section. The caller of the GetConfig method must have enough information about the configuration section handler to know how to use the object that is returned. Two stock configuration section handlers—and the objects they create—will be discussed in this section. Example 3-3 shows how to read the configuration file shown in Example 3-2. To run this example, do the following:
   1. Create a new directory for the application.
   2. Save the code from Example 3-3 into a file named *ConfigurationTest.vb*.
   3. Compile the code with this command line:
      vbc ConfigurationTest.vb /reference:System.dll
      The reference to the System assembly is required because the System assembly contains the definition of the ConfigurationSettings class.The compiler creates an executable file named *ConfigurationTest.exe*.
   4. Save the configuration file from Example 3-2 into a file named *ConfigurationTest.exe.config.*

Run the executable from the command prompt. The application prints the configuration values to the command window.

Example 3-3. Reading the configuration file shown in Example 3-2

```
Imports System
Imports System.Collections
Imports System.Configuration

Public Module SomeModule

    Public Sub Main( )
        Dim cfg As Hashtable
        Dim strSomeSetting As String
        Dim strAnotherSetting As String
        cfg = CType(ConfigurationSettings.GetConfig("mySectionName"),
            _Hashtable)
        If Not (cfg Is Nothing) Then
            strSomeSetting = CType(cfg("someSetting"), String)
            strAnotherSetting = CType(cfg("anotherSetting"), String)
        End If
        Console.WriteLine(strSomeSetting)
        Console.WriteLine(strAnotherSetting)
    End Sub

End Module
```

To read the configuration settings, the code in Example 3-3 calls the GetConfig method of the ConfigurationSettings class. The SingleTagSectionHandler configuration section handler creates a Hashtable object (defined in the System.Collections namespace) to hold the key/value pairs found in the configuration file. That is why the code in Example 3-3 calls the *CType* function to convert the reference returned by the GetConfig method to a Hashtable reference. After that is done, the code can do anything appropriate for a Hashtable object, including retrieving specific values by key (as shown in Example 3-3) or iterating through the Hashtable object's items. Also note that because Hashtable objects store values of type Object, the object references retrieved from the Hashtable have to be converted to the appropriate reference type, which in this case is String. The Visual Basic *CStr* function could have been used here, although in this case the Visual Basic *CType* function is called instead. The application does not specify the name of the configuration file in which to look for the configuration information. The system automatically looks in the *application_file_name.config* file found in the application's directory. If the requested section is not found in that file, the system automatically looks for it in the machine-configuration file. Another stock configuration section handler is the NameValueSectionHandler class. This handler also reads key/value pairs, but in a different format. Example 3-4 is the same as Example 3-2, but rewritten to use NameValueSectionHandler.

Example 3-4. Using the NameValueSectionHandler configuration section Handler

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<configSections>
<section
name="mySectionName"
type="System.Configuration.NameValueSectionHandler" />
</configSections>
<mySectionName>
<add key="someSetting" value="SomeValue" />
<add key="anotherSetting" value="AnotherValue" />
</mySectionName>
</configuration>
```

Example 3-5 shows the code that reads this configuration section.

Example 3-5. Reading the configuration file shown in Example 3-4

```
Imports System
Imports System.Collections.Specialized
Imports System.Configuration
Public Module SomeModule
    Public Sub Main( )
        Dim cfg As NameValueCollection
        Dim strSomeSetting As String
        Dim strAnotherSetting As String
        cfg = CType(ConfigurationSettings.GetConfig("mySectionName"),
            _NameValueCollection)
            If Not (cfg Is Nothing) Then
                strSomeSetting = CType(cfg("someSetting"), String)
            strAnotherSetting = CType(cfg("anotherSetting"), String)
        End If
        Console.WriteLine(strSomeSetting)
        Console.WriteLine(strAnotherSetting)
    End Sub
End Module
```

The main difference to note in Example 3-5 is that the NameValueSectionHandler creates an object of type NameValueCollection (defined in the System.Collections.Specialized namespace).

3.12.2 Configuration Section Groups

If application-configuration information is to be stored in the machine-configuration file, it is a good idea to introduce *configuration section groups* into the picture. (Recall that if the runtime doesn't find the requested section in the application-configuration file, it automatically looks for it in the machineconfiguration file.) This simply groups an application's settings into an enclosing group element in the configuration file, so that the contained elements won't potentially conflict with like-named elements for other applications. Example 3-6 shows how to introduce a section group. It is identical to the configuration file shown in Example 3-2, except that a section group is defined.

Example 3-6. Creating a section group

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <configSections>
    <sectionGroup name="myGroupName">
        <section
        name="mySectionName"
        type="System.Configuration.SingleTagSectionHandler" />
```

```
            </sectionGroup>
            </configSections>
        <myGroupName>
            <mySectionName
            someSetting="SomeValue"
            anotherSetting="AnotherValue" />
        </myGroupName>
    </configuration>
```

Example 3-7 shows how to read this configuration file in code.

Example 3-7. Reading the configuration file shown in Example 3-6

```
    Imports System
    Imports System.Collections
    Imports System.Configuration
    Public Module SomeModule
    Public Sub Main( )
        Dim cfg As Hashtable
        Dim strSomeSetting As String
        Dim strAnotherSetting As String
        cfg = CType( _
    ConfigurationSettings.GetConfig("myGroupName/mySectionName"), _Hashtable)
    If Not (cfg Is Nothing) Then
        strSomeSetting = CType(cfg("someSetting"), String)
        strAnotherSetting = CType(cfg("anotherSetting"), String)
    End If
        Console.WriteLine(strSomeSetting)
        Console.WriteLine(strAnotherSetting)
    End Sub
    End Module
```

The only difference between Example 3-7 and Example 3-3 is the path-style syntax in Example 3-7 used to specify the section name: "myGroupName/mySectionName". Group definitions can be nested, if desired.

3.12.3 The <appSettings> Section

Most applications just need a simple way to store key/value pairs. To support this, the *machine.config* file contains a predefined section definition called <appSettings>. It is always legal to include an <appSettings> section in any configuration file. The configuration section handler for the <appSettings> section is the NameValueSectionHandler class, so the section should be in this form:

```
    <appSettings>
    <add key="setting1" value="value1" />
    <add key="setting2" value="value2" />
    <add key="setting3" value="value3" />
    </appSettings>
```

Although the <appSettings> section can be read using the GetConfig method just like any other section, the ConfigurationSettings class has a property that is specifically intended to assist with reading the <appSettings> section. The read-only AppSettings property of the ConfigurationSettings class returns a NameValueCollection object that contains the key/value pairs found in the <appSettings> section. Example 3-8 shows how to read the settings shown in the previous code listing.

Example 3-8. Reading the <appSettings> section

```
Imports System
Imports System.Collections.Specialized
Imports System.Configuration
Public Module SomeModule
    Public Sub Main( )
        Dim cfg As NameValueCollection
        Dim strSetting1 As String
        Dim strSetting2 As String
        Dim strSetting3 As String
    cfg = CType(ConfigurationSettings.AppSettings,
    NameValueCollection)
        If Not (cfg Is Nothing) Then
            strSetting1 = CType(cfg("setting1"), String)
            strSetting2 = CType(cfg("setting2"), String)
            strSetting3 = CType(cfg("setting3"), String)
        End If
        Console.WriteLine(strSetting1)
        Console.WriteLine(strSetting2)
        Console.WriteLine(strSetting3)
    End Sub
End Module
```

The name/value pairs in the <appSettings> section are developer-defined. The CLR doesn't
attribute any intrinsic meaning to any particular name/value pair.

3.9 -3.12 Check Your Progress

1. In NET framework, the declarative information is stored as ……………………..
2. Recognizing when objects are no longer being used is called ……………………..
3. The process of freeing up memory is known as ……………………..
4. The …………………….. method is an alternate destructor address.
5. …………………… represents a collection of key / value pairs.

3.13 SUMMARY

The .NET Framework is a broad and deep new foundation for application development. At its core is a runtime that provides services that were previously found in compiler libraries. This runtime eliminates the application's need to possess knowledge of the underlying operating system and hardware, while providing performance on par with natively compiled code.

3.14 CHECK YOUR PROGRESS-*ANSWERS*

3.1- 3.3
1. Common Language Runtime
2. CTS
3. Runtime Infrastrucure
4. Run Time
5. Managed Code

3.4 - 3.7
1. Assembly
2. Global Assembly Cache
3. Application domains
4. Common Language Specification
5. Intermediate Language.

3.8- 3.11
1. Meta-data
2. Life management
3. Garbage collection
4. Dispose
5. Hash Table

3.15 QUESTIONS FOR SELF-STUDY

1. Describe GAC in detail.
2. What is assembly , differentiate between global and private assembly?
3. Compare GC of .Net with Garbage collector of JAVA.
4. Describe namespace and differeintiate that with Assembly.
5. How can we make a private assembly global, state all the steps.

3.16 SUGGESTED READINGS

1. Visual Basic .NET Black Book by Steven Holzner

References

Programming Visual Basic .NET by Dave Grundgeiger

◈◈◈

NOTES

NOTES

CHAPTER 4

WINDOWS FORMS I: DEVELOPING DESKTOP APPLICATIONS

4.0 OBJECTIVES

After you study The. NET Framework chapter you will be able to:-

- Create Forms
- Handle Form Events
- State relationships between Forms
- Create MDI Applications
- State Components Attributes
- Design 2-D Graphics Programming with GDI+
- Code printing in application

4.1 INTRODUCTION

Windows Forms is a set of classes that encapsulates the creation of the graphical user interface (GUI) portion of a typical desktop application. Previously, each programming language had its own way of creating windows, text boxes, buttons, etc. This functionality has all been moved into the .NET Framework class library—into the types located in the System.Windows.Forms namespace. Closely related is the System.Drawing namespace, which contains several types used in the creation of GUI applications. The capabilities provided by the types in the System.Drawing namespace are commonly referred to as GDI+ (discussed more fully later in this chapter). In this chapter, we'll examine the form (or window) as the central component in a classic desktop application. We'll look at how forms are programmatically created and how they're hooked to events. We'll also examine how multiple forms in a single application relate to one another and how you handle forms in an application that has one or more child forms. Finally, we'll discuss two topics, printing and 2-D graphics, that are relevant to desktop application development.

4.2 CREATING A FORM

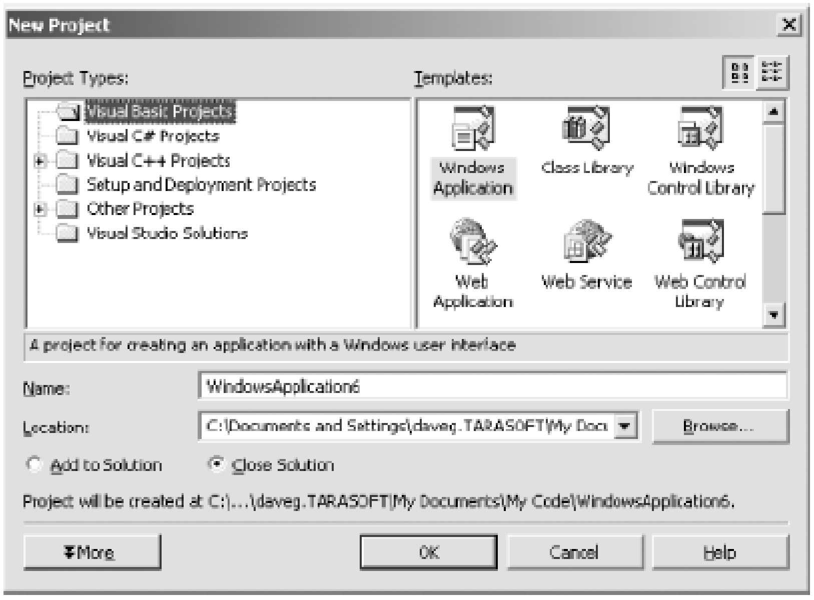
The easiest way to design a form is to use the Windows Forms Designer in Visual Studio .NET. The developer can use visual tools to lay out the form, with the designer translating the layout into Visual Basic .NET source code. If you don't have Visual Studio .NET, you can write the Visual Basic .NETcode directly and not use the designer at all. This section will demonstrate both methods. Programmatically, a form is defined by deriving a class from the Form class (defined in
System.Windows.Forms). The Form class contains the know-how for displaying an empty form, including its title bar and other amenities that we expect from a Windows form. Adding members to the new class and overriding members inherited from the Form class add visual elements and behavior to the new form.

4.2.1 Creating a Form Using Visual Studio .NET

To create a GUI application in Visual Studio .NET:

1. Select File New Project. The New Project dialog box appears, as shown in Figure 4-1.

Figure 4-1. The New Project dialog box



A project for creating an application with a Windows user interface

Name: WindowsApplication6

Location: C:\Documents and Settings\daveg.TARASOFT\My Docu

Close Solution

Project will be created at C:\...\daveg.TARASOFT\My Documents\My Code\WindowsApplication6.

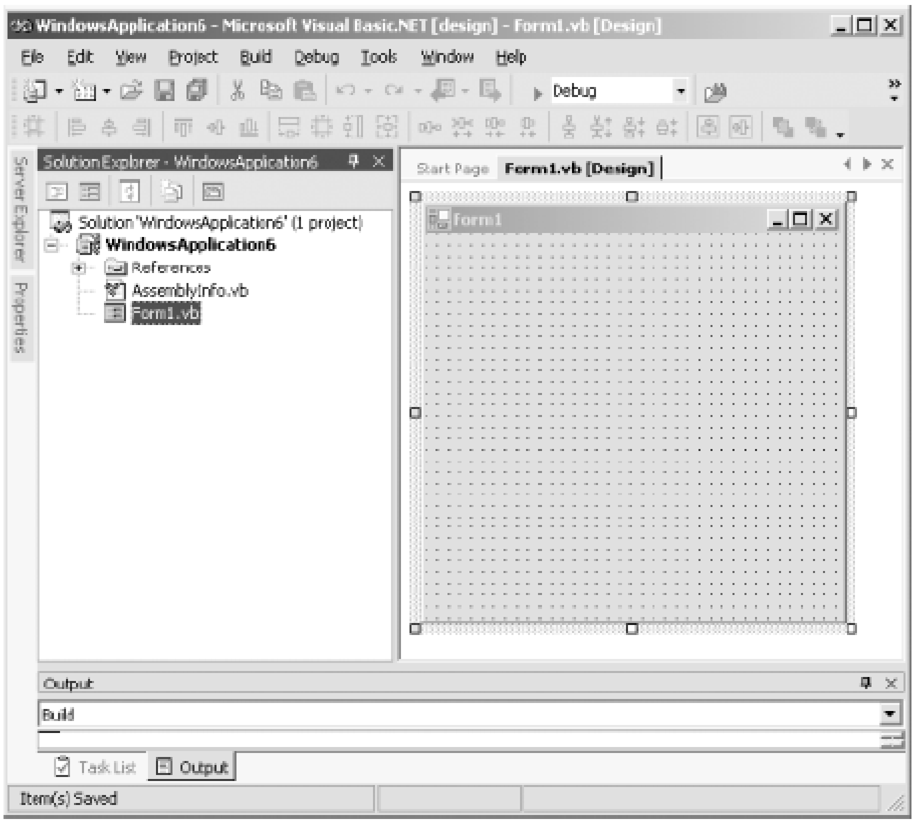
2. Select Visual Basic Projects in the Project Types pane on the left side of the dialog box.

3. Select Windows Application in the Templates pane on the right side of the dialog box.

4. Enter a name in the Name text box.

5. Click OK. Visual Studio .NET creates a project with a form in it and displays the form in a designer, as shown in Figure 4-2.

Figure 4-2. The Windows Forms Designer


WindowsApplication6 - Microsoft Visual Basic.NET [design] - Form1.vb [Design]
File Edit View Project Build Debug Tools Window Help
Debug
Solution Explorer - WindowsApplication6
Solution 'WindowsApplication6' (1 project)
WindowsApplication6
References
AssemblyInfo.vb
Form1.vb
Start Page
Form1.vb [Design]
Form1
Output
Build
Task List
Output
Item(s) Saved
Server Explorer
Properties

To see the code created by the form Windows Forms Designer, right-click on the form, then select View Code. Doing this for the blank form shown in Figure 4-2 reveals the code shown here:
Public Class Form1 Inherits System.Windows.Forms.Form Windows Form Designer generated code End Class This shows the definition of a class named Form1 that inherits from the Form class. The Windows Forms Designer also creates a lot of boilerplate code that should not be modified by the developer. By default, it hides this code from view. To see the code, click on the "+" symbol that appears to the left of the line that says "Windows Form Designer generated code." Doing so reveals the code shown in Example 4-1.

Example 4-1. The Windows Forms Designer-generated code for a blank form

```
Public Class Form1
            Inherits System.Windows.Forms.Form
            #Region " Windows Form Designer generated code "
                Public Sub New( )
                    MyBase.New( )
                    'This call is required by the Windows Form Designer.
                    InitializeComponent( )
                    'Add any initialization after the InitializeComponent( ) call
                End Sub
            'Form overrides dispose to clean up the component list.
            Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
                    If disposing Then If Not (components Is Nothing) Then
                        components.Dispose( )
                    End If
        End If
            MyBase.Dispose(disposing)
        End Sub
```

'Required by the Windows Form Designer Private components As System.ComponentModel.Container 'NOTE: The following procedure is required by the Windows Form Designer. It can be modified using the Windows Form Designer.

```
        'Do not modify it using the code editor.

        <System.Diagnostics.DebuggerStepThrough( )> Private Sub
        InitializeComponent( )
        components = New System.ComponentModel.Container( )
            Me.Text = "Form1"
            End Sub
            #End Region
    End Class
```

The Windows Forms Designer autogenerates the code for four class members:

New method (the class constructor)

The constructor calls the base class's constructor and then invokes the InitializeComponent method. Developer-supplied initialization code should follow the call to InitializeComponent.After the constructor is generated, the designer doesn't touch it again.

Dispose method
The Dispose method is where the object gets rid of any expensive resources. In this case, it calls the base class's Dispose method to give it a chance to release any expensive resources that it may hold, then it calls the components field's Dispose method. (For more on the components field, see the next item.) This in turn calls the Dispose methods on each individual component in the collection. If the derived class uses any expensive resources, the developer should add code here to release them. When a form is no longer needed, all code that uses the form should call the form's Dispose method. After the Dispose method is generated, the designer doesn't touch it again.

Components field
The components field is an object of type IContainer (defined in the System.ComponentModel namespace). The designer-generated code uses the components field to manage finalization of components that may be added to a form (for example, the Timer component).

InitializeComponent method
The code in this method should not be modified or added to by the developer in any way. The Windows Forms Designer automatically updates it as needed. When controls are added to the form using the designer, code is added to this method to instantiate the controls at runtime and set their initial properties.

Note also in Example 4-1 that properties of the form itself (such as Text and Name) are initialized in this method. One thing missing from this class definition is a Main method. Recall from previous chapter that .NET applications must expose a public, shared Main method. This method is called by the CLR when an application is started. So why doesn't the designer-generated form include a Main method? It's because the Visual Basic .NET compiler in Visual Studio .NET automatically creates one as it compiles the code. In other words, the compiled code has a Main method in it even though the source code does not. The Main method in the compiled code is a member of the Form1 class and is equivalent to this:

```
    <System.STAThreadAttribute( )> Public Shared Sub Main( )
        System.Threading.Thread.CurrentThread.ApartmentState = _
        System.Threading.ApartmentState.STA
        System.Windows.Forms.Application.Run(New Form1( ))
    End Sub
```

Note that the Visual Basic .NET command-line compiler doesn't automatically generate the Main method. This method must appear in the source code if the command-line compiler is to be used.

The next steps in designing the form are to name the code file something meaningful and to set some properties on the form, such as the title-bar text. To change the name of the form's code file, right-click on the filename in the Solution Explorer window and select Rename. If you're following along with this example, enter *HelloWindows.vb* as the name of the file.Changing the name of the file doesn't change the name of the class. To change the name of the class, right-click the form in the designer and choose Properties. In the Properties window, change the value of the Name property. For this example, change the name to "HelloWindows".

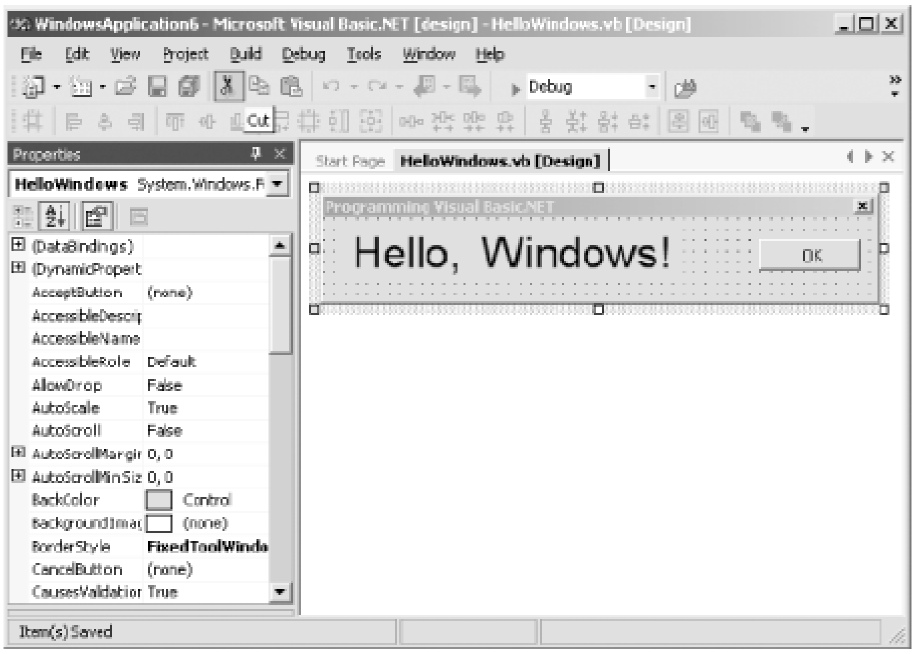
To change the form's caption, set the form's Text property to a new value. Set the Text property in this example to "Programming Visual Basic .NET". Next, controls can be added to the form from the Visual Studio .NET toolbox. To display the toolbox, select View Toolbox from the Visual Studio .NET main menu.

For this example, double-click on the Label control in the toolbox to add a Label control on the form. Use the Properties window to change the label's Text property to "Hello, Windows!" and its Font property to Arial 24pt.

Next, double-click on the Button control in the toolbox to add a Button control to the form. Use the Properties window to change the button's Name property to "OkButton" and its Text property to "OK". Finally,

position the controls as desired, size the Label control and the form to be appealing, and set the form's FormBorderStyle property to "FixedToolWindow". The resulting form should look something like the one shown in Figure 4-3.

Figure 4-3. A form with controls

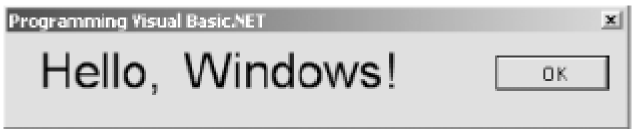


HelloWindows System.Windows.F

(DataBindings)	
(DynamicPropert	
AcceptButton	(none)
AccessibleDescri	
AccessibleName	
AccessibleRole	Default
AllowDrop	False
AutoScale	True
AutoScroll	False
AutoScrollMargir	0, 0
AutoScrollMinSiz	0, 0
BackColor	Control
BackgroundImag	(none)
BorderStyle	**FixedToolWindo**
CancelButton	(none)
CausesValidatior	True

Press the F5 key to build and run the program. The result should look something like Figure 4-4.

Figure 4-4. Hello, Windows!, as created by the Windows Forms Designer



The code generated by the designer is shown in Example 4-2.

Example 4-2. Hello, Windows! code, as generated by the Windows Forms Designer

```
Public Class HelloWindows
    Inherits System.Windows.Forms.Form
    #Region " Windows Form Designer generated code "
    Public Sub New( )
```

```
            MyBase.New( )
            'This call is required by the Windows Form Designer.
            InitializeComponent( )
            'Add any initialization after the InitializeComponent( ) call
        End Sub
        'Form overrides dispose to clean up the component list.

        Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
            If disposing Then
            If Not (components Is Nothing) Then
            components.Dispose( )
            End If
            End If
            MyBase.Dispose(disposing)
        End Sub

        Friend WithEvents Label1 As System.Windows.Forms.Label
        Friend WithEvents OkButton As System.Windows.Forms.Button
            'Required by the Windows Form Designer
            Private components As System.ComponentModel.Container
            'NOTE: The following procedure is required by the Windows Form Designer
            'It can be modified using the Windows Form Designer.
            'Do not modify it using the code editor.

    <System.Diagnostics.DebuggerStepThrough( )> _
    Private Sub InitializeComponent( )
        Me.Label1 = New System.Windows.Forms.Label( )
        Me.OkButton = New System.Windows.Forms.Button( )
            Me.SuspendLayout( )
            '
            'Label1
            '
        Me.Label1.Font = New System.Drawing.Font("Arial", 24!, _
        System.Drawing.FontStyle.Regular, _
        System.Drawing.GraphicsUnit.Point, CType(0, Byte))
        Me.Label1.Location = New System.Drawing.Point(8, 8)
        Me.Label1.Name = "Label1"
        Me.Label1.Size = New System.Drawing.Size(264, 48)
        Me.Label1.TabIndex = 0
        Me.Label1.Text = "Hello, Windows!"
        '
        'OkButton
        '
        Me.OkButton.Location = New System.Drawing.Point(280, 16)
        Me.OkButton.Name = "OkButton"
        Me.OkButton.TabIndex = 1
        Me.OkButton.Text = "OK"
        '
        'HelloWindows
        '
        Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
        Me.ClientSize = New System.Drawing.Size(362, 58)
        Me.Controls.AddRange(New System.Windows.Forms.Control( ) _
        {Me.OkButton, Me.Label1})
        Me.FormBorderStyle = _
        System.Windows.Forms.FormBorderStyle.FixedToolWindow
        Me.Name = "HelloWindows"
        Me.Text = "Programming Visual Basic .NET"
        Me.ResumeLayout(False)
    End Sub
    #End Region
    End Class
```

Note that the designer made the following modifications to the code:

- Two Friend fields were added to the class, one for each of the controls that were added to the form:

- Friend WithEvents Label1 As System.Windows.Forms.Label Friend WithEvents OkButton As System.Windows.Forms.Button The Friend keyword makes the members visible to other code within the project, but it hides them from code running in other assemblies. The WithEvents keyword allows the HelloWindows class to handle events generated by the controls. In the code shown, no event handlers have been added yet, but you'll see how to do that later in this section. Note that the field names match the control names as shown in the Properties window.
- Code was added to the InitializeComponent method to instantiate the two controls and assign their references to the member fields:
- Me.Label1 = New System.Windows.Forms.Label( ) Me.OkButton = New System.Windows.Forms.Button( )
- Code was added to the InitializeComponent method to set various properties of the label, button, and form. Some of these assignments directly correspond to the settings made in the Properties window, while others are the implicit result of other actions taken in the designer (such as sizing the form).

4.2.1.1 Adding event handlers

The *Hello, Windows!* application built thus far has an OK button, but the application doesn't yet respond to button clicks. To add a Click event handler for the OK button, double-click on the button in the Windows Forms Designer. The designer responds by switching to the form's code view and inserting a subroutine that handles the Click event (i.e., it will be called when the user of the running application clicks the OK button). The subroutine the designer creates looks like this (note that I added the line-continuation character for printing in this book):

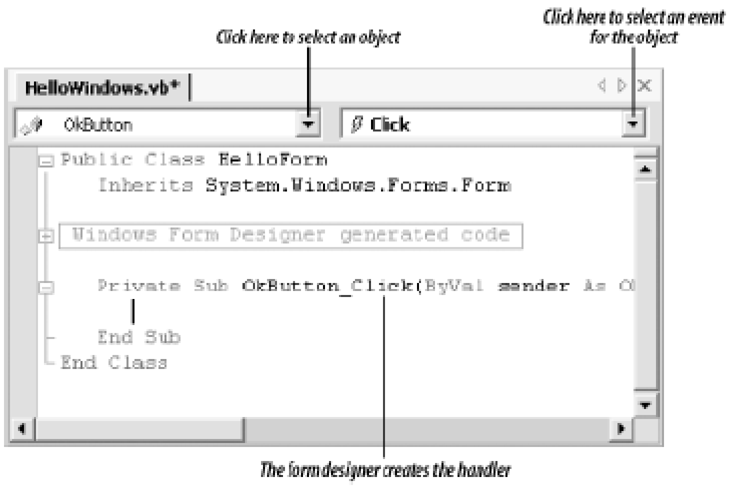
```
Private Sub OkButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles OkButton.Click
End Sub
```

The body of the subroutine can then be added. This would be a likely implementation for this event handler:

```
Private Sub OkButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles OkButton.Click
    Me.Close( )
    Me.Dispose( )
End Sub
```

An alternative way to add an event handler is to use the drop-down lists at the top of the form's codeview window. In the lefthand drop-down list, choose the object for which you would like to add an event handler. Then, in the righthand drop-down list, choose the desired event. See Figure 4-5.

Figure 4-5. Adding an event handler using the code view's drop-down Lists


Click here to select an object
Click here to select an event for the object
HelloWindows.vb*
OkButton
Click
The form designer creates the handler

```
Public Class HelloForm
    Inherits System.Windows.Forms.Form

 Windows Form Designer generated code

    Private Sub OkButton_Click(ByVal sender As O

    End Sub
End Class
```

Event handlers can be typed directly into the form's code if you know the correct signature for the handler. Event-handler signatures are documented in the Microsoft Developer Network (MSDN) Library.

4.2.2 Creating a Form in Code

Although form designers are convenient, it is certainly possible to code a form directly. To do so, follow these steps:

1. Define a class that is derived from the Form class (defined in the System.Windows.Forms namespace).
2. If the form is to be the startup form for an application, include a public, shared Main method. For example:
3. Imports System.Windows.Forms
4. Public Class HelloWindows
5. Inherits Form
6. ' Include this method only if this is the application's startup form.
7. ' Alternatively, place this method in a separate module in the
8. ' application. If it is placed in a separate module, remove the
9. ' Shared keyword.
10. <System.STAThreadAttribute( )> Public Shared Sub Main( )
11. System.Threading.Thread.CurrentThread.ApartmentState = _
12. System.Threading.ApartmentState.STA
13. Application.Run(New HelloWindows( ))
14. End Sub ' Main
15. End Class
16. Declare a data member for each control that is to appear on the form. If you want to handle events from the control, use the WithEvents keyword in the declaration. For example:
17. Imports System.Windows.Forms
18. Public Class HelloWindows
19. Inherits Form
20. Private lblHelloWindows As Label
21. Private WithEvents btnOK As Button.
22. <System.STAThreadAttribute( )> Public Shared Sub Main( )
23. System.Threading.Thread.CurrentThread.ApartmentState = _
24. System.Threading.ApartmentState.STA
25. Application.Run(New HelloWindows( ))
26. End Sub ' Main
27. End Class

    The visibility (Private, Friend, Protected, or Public) of these data members is a design issue that depends on the project and on the developer's preferences. My own preference is to make all data members private. If code external to the class needs to modify the data held by these members, specific accessor methods can be added for the purpose. This prevents internal design changes from affecting external users of the class.
28. Declare a constructor. Perform the following operations in the constructor:
    a. Instantiate each control.
    b. Set properties for each control and for the form.
    c. Add all controls to the form's Controls collection.
    For example:

```
Imports System.Drawing
Imports System.Windows.Forms
Public Class HelloWindows
Inherits Form
Private lblHelloWindows As Label
Private WithEvents btnOK As Button

Public Sub New
            ' Instantiate a label control and set its
      properties.
      lblHelloWindows = New Label( )
      With lblHelloWindows
            .Font = New Font("Arial", 24)
            .Location = New Point(16, 8)
            .Size = New Size(248, 40)
            .TabIndex = 0
            .Text = "Hello, Windows!"
      End With
            ' Instantiate a button control and set its
      properties.
      btnOK = New Button( )
      With btnOK
            .Location = New Point(320, 16)
            .TabIndex = 1
            .Text = "OK"
      End With

      ' Set properties on the form.
      FormBorderStyle = FormBorderStyle.FixedToolWindow
      ClientSize = New Size(405, 61)
      Text = "Programming Visual Basic .NET"
      ' Add the controls to the form's Controls collection.
      Controls.Add(lblHelloWindows
      Controls.Add(btnOK)
End Sub
      <System.STAThreadAttribute( )> Public Shared Sub Main( )
      System.Threading.Thread.CurrentThread.ApartmentState = _
      System.Threading.ApartmentState.STA
      Application.Run(New HelloWindows( ))
End Sub ' Main
End Class
```

An Imports statement was added to give access to types in the System.Drawing namespace, such as Point and Size.

4.2.2.1 Adding event handlers

Define event handlers directly in code for any events that you wish to handle. For example:

```
   Private Sub btnOK_Click(ByVal sender As Object, _
      ByVal e As System.EventArgs) Handles btnOK.Click
      Close( )
      Dispose( )
   End Sub
```

The complete code for a standalone Windows Forms application is shown in Example 4-3. Compile it from the command line with this command:

```
vbc HelloWindows.vb
/r:System.dll,System.Drawing.dll,System.Windows.Forms.dll /t:winexe
```

(Note that the command should be typed on a single line.)

Example 4-3. Hello, Windows! code generated outside of Visual Studio

```
   Imports System.Drawing
   Imports System.Windows.Forms
   Public Class HelloWindows
      Inherits Form
   Private lblHelloWindows As Label
   Private WithEvents btnOK As Button
   Private Sub btnOK_Click(ByVal sender As Object, _
   ByVal e As System.EventArgs) Handles btnOK.Click
      Close()
      Dispose()
   End Sub
   Public Sub New
      ' Instantiate a label control and set its properties.
      lblHelloWindows = New Label ()
```

```
        With lblHelloWindows
            .Font = New Font("Arial", 24)
            .Location = New Point(16, 8)
            .Size = New Size(248, 40)
            .TabIndex = 0
            .Text = "Hello, Windows!"
    End With
        ' Instantiate a button control and set its properties.
        btnOK = New Button()
        With btnOK
            .Location = New Point(320, 16)
            .TabIndex = 1
            .Text = "OK"
    End With
    ' Set properties on the form.
    FormBorderStyle = FormBorderStyle.FixedToolWindow
    ClientSize = New Size(405, 61)
    Text = "Programming Visual Basic .NET"
    ' Add the controls to the form's Controls collection.
    Controls.Add(lblHelloWindows)
    Controls.Add(btnOK)
    End Sub
    <System.STAThreadAttribute()> Public Shared Sub Main()
    System.Threading.Thread.CurrentThread.ApartmentState = _
    System.Threading.ApartmentState.STA
    Application.Run(New HelloWindows())
    End Sub ' Main
    End Class
```

4.2.2.2 Handling Form Events

The base Form class may at times raise events. These events can be handled by the derived Form class. One way to do this is to define a handler subroutine that uses the MyBase keyword in the Handles clause, like this:

```
    ' This is not the preferred technique.
    Private Sub Form_Closing( _
    ByVal sender As Object, _
    ByVal e As System.ComponentModel.CancelEventArgs _
    ) Handles MyBase.Closing
    ' ...
    End Sub
```

However, a better technique is to override the protected methods, which are provided by the Form class for this purpose. For example, the following method could be placed in the derived class's definition, providing a way to respond to the form's imminent closing:

```
    ' Assumes Imports System.ComponentModel

    Protected Overrides Sub OnClosing( _ByVal e As CancelEventArgs _)
    ' ...
    MyBase.OnClosing(e) ' Important
    End Sub
```

Note that the implementation of the OnClosing method includes a call to the base class's implementation. This is important. If this is not done, the Closing event won't be raised, which will affect the behavior of any other code that has registered for the event.Following is the list of events the Form class defines, including a brief description of each event and the syntax for overriding the protected method that corresponds to each event. Note also that the Form class indirectly derives from the Control class and that the Control class also exposes events and overridable methods that aren't shown here.

Activated
Fired when the form is activated. Its syntax is:

Protected Overrides Sub OnActivated(ByVal e As System.EventArgs)

Closed
Fired when the form has been closed. Its syntax is:
Protected Overrides Sub OnClosed(ByVal e As System.EventArgs)

Closing
Fired when the form is about to close. Its syntax is:

```
Protected Overrides Sub OnClosing(_ByVal e As
    System.ComponentModel.CancelEventArgs)
```

The CancelEventArgs.Cancel property can be set to True to prevent the form from closing; its default value is False.

Deactivate
Fired when the form is deactivated. Its syntax is:
Protected Overrides Sub OnDeactivate(ByVal e As System.EventArgs)

InputLanguageChanged
Fired when the form's input language has been changed. Its syntax is:

```
Protected Overrides Sub OnInputLanguageChanged( _ByVal e As
System.Windows.Forms.InputLanguageChangedEventArgs)
```

The InputLanguageChangedEventArgs class has three properties that identify the new language: CharSet, which defines the character set associated with the new input language;Culture, which contains the culture code (see Appendix C) of the new input language; and InputLanguage, which contains a value indicating the new language.
InputLanguageChanging Fired when the form's input language is about to be changed. Its syntax is:

```
Protected Overrides Sub OnInputLanguageChanging( _ByVal e As
System.Windows.Forms.InputLanguageChangingEventArgs)
```

The InputLanguageChangingEventArgs class has a Culture property that identifies the proposed new language and locale. It also has a Cancel property that can be set to True within the event handler to cancel the change of input language; the default value of the Cancel property is False.

Load
Fired when the form is loaded. Its syntax is:
Protected Overrides Sub OnLoad(ByVal e As System.EventArgs)

MaximizedBoundsChanged
Fired when the value of the form's MaximizedBounds property (which determines the size of the maximized form) is changed. Its syntax is:
Protected Overrides Sub OnMaximizedBoundsChanged(_ByVal e As
System.EventArgs)

MaximumSizeChanged
Fired when the value of the form's MaximumSize property (which defines the maximum size to which the form can be resized) is changed. Its syntax is:
Protected Overrides Sub OnMaximumSizeChanged(ByVal e As
System.EventArgs)

MdiChildActivate
Fired when an MDI child window is activated. Its syntax is:
Protected Overrides Sub OnMdiChildActivate(ByVal e As System.EventArgs)

MenuComplete
Fired when menu selection is finished. Its syntax is:
Protected Overrides Sub OnMenuComplete(ByVal e As System.EventArgs)

MenuStart
Fired when a menu is displayed. Its syntax is:

Protected Overrides Sub OnMenuStart(ByVal e As System.EventArgs)

MinimumSizeChanged
Fired when the value of the form's MinimumSize property (which defines the minimum size to which the form can be resized) is changed. Its syntax is:
Protected Overrides Sub OnMinimumSizeChanged(ByVal e As System.EventArgs)

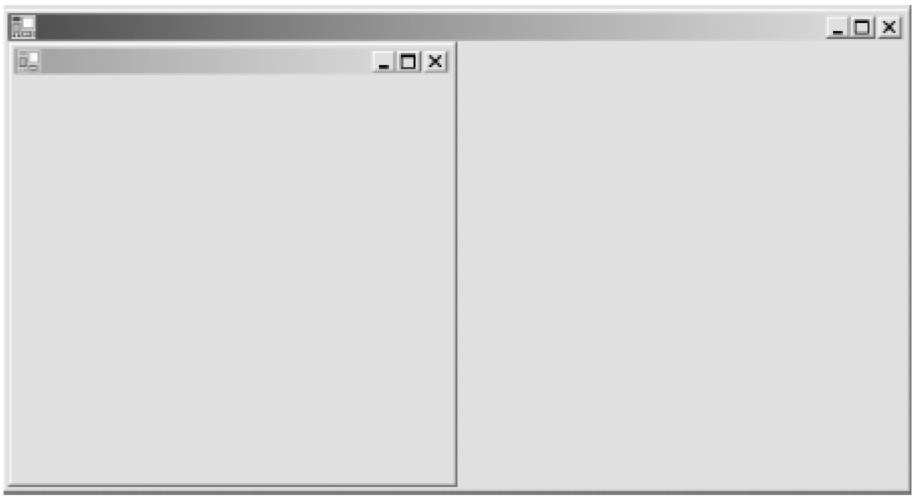
4.3 RELATIONSHIPS BETWEEN FORMS

The Form class has two properties that control a form's relationship to other forms: the Parent property (inherited from the Control class) and the Owner property. Setting the Parent property causes the constrained form to appear only within the bounds of the parent—and always to appear on top of the parent. This gives an effect similar to MDI applications (which have other features as well and are discussed later in this chapter). When a form has a parent, it can be docked to the parent's edges, just like any other control. The code in Example 4-4 demonstrates this. It can be compiled from the command line with this command:

Vbc*filename.vb* /r:System.dll,System.Drawing.dll,System.Windows.Forms.dll
/t:winexe

The result is displayed in Figure 4-6.

Figure 4-6. A form with a parent



Example 4-4. Creating a form with a parent
```
Imports System.Drawing
Imports System.Windows.Forms
Module modMain
    <System.STAThreadAttribute( )> Public Sub Main( )
    System.Threading.Thread.CurrentThread.ApartmentState = _
    System.Threading.ApartmentState.STA
    System.Windows.Forms.Application.Run(New MyParentForm( ))
    End Sub
    End Module
    Public Class MyParentForm
    Inherits Form
    Public Sub New( )
        ' Set my size.
        Me.ClientSize = New System.Drawing.Size(600, 400)
        ' Create and show a child form.
        Dim frm As New MyChildForm(Me)
        frm.Show( )
    End Sub
End Class
    Public Class MyChildForm
        Inherits Form
    Public Sub New(ByVal Parent As Control)
```

```
        ' TopLevel must be False for me to have a parent.
        Me.TopLevel = False
        ' Set my parent.
        Me.Parent = Parent
        ' Dock to my parent's left edge.
        Me.Dock = DockStyle.Left
    End Sub
End Class
```

If the child form is maximized, it expands to fill the parent form. If the child form is minimized, it shrinks to a small rectangle at the bottom of the parent window. Because the child form in this example has a title bar and a sizable border, it can be moved and sized even though it has been docked. This behavior can be changed by modifying the form's FormBorderStyle property.

Setting the Owner property of a form causes another form to own the first. An owned form is not constrained to appear within the bounds of its owner, but when it does overlay its owner, it is always on top. Furthermore, the owned form is always minimized, restored, or destroyed when its owner is minimized, restored, or destroyed. Owned forms are good for floating-tool windows or Find/Replacetype dialog boxes. The code in Example 4-5 creates an owner/owned relationship. Compile it with this command:

Vbc*filename.vb* /r:System.dll,System.Drawing.dll,System.Windows.Forms.dll /t:winexe

Example 4-5. Creating a form with an owner

```
Imports System.Drawing
Imports System.Windows.Forms
Module modMain
    <System.STAThreadAttribute( )> Public Sub Main( )
    System.Threading.Thread.CurrentThread.ApartmentState = _
    System.Threading.ApartmentState.STA
    System.Windows.Forms.Application.Run(New MyOwnerForm( ))
End Sub
End Module
Public Class MyOwnerForm
    Inherits Form
Public Sub New( )
    ' Set my size.
    Me.ClientSize = New System.Drawing.Size(600, 450)
    ' Create and show an owned form.
    Dim frm As New MyOwnedForm(Me)
    frm.Show( )
End Sub
End Class
Public Class MyOwnedForm
    Inherits Form
Public Sub New(ByVal Owner As Form)
    ' Set my owner.
    Me.Owner = Owner
End Sub
End Class
```

4.1 - 4.3 Check Your Progress

State whether the following statements are TRUE OR FALSE

1. .NET command line complier does not automatically generate the main method
2. The form class has three properties that control form’s relationship to other forms
3. Printer dialog box is under windows common dialg boxes.
4. Child form is the property of Parent MDI form.

4.4 MDI APPLICATIONS

Multiple document interface (MDI) applications permit more than one document to be open at a time. This is in contrast to *single document interface* (SDI) applications, which can manipulate only one document at a time. Visual Studio .NET is an example of an MDI application—many source files and design views can be open at once. In contrast, Notepad is an example of an SDI application—opening a document closes any previously opened document. There is more to MDI applications than their ability to have multiple files open at once. The Microsoft Windows platform SDK specifies several UI behaviors that MDI applications should implement. The Windows operating system provides support for these behaviors, and this support is exposed through Windows Forms as well.

4.4.1 Parent and Child Forms

MDI applications consist of a main form, which does not itself display any data, and one or more child forms, which appear only within the main form and are used for displaying documents. The main form is called the *MDI parent*, and the child forms are called the *MDI children*.
The Form class has two properties that control whether a given form is an MDI parent, MDI child, or neither. The Boolean IsMdiContainer property determines whether a form behaves as an MDI parent. The MdiParent property (which is of type Form) controls whether a form behaves as an MDI child. Setting the MdiParent property of a form to reference the application's MDI parent form makes the form an MDI child form. Example 4-6 shows the minimum amount of code required to display an MDI parent form containing a single MDI child form.

Example 4-6. A minimal MDI application

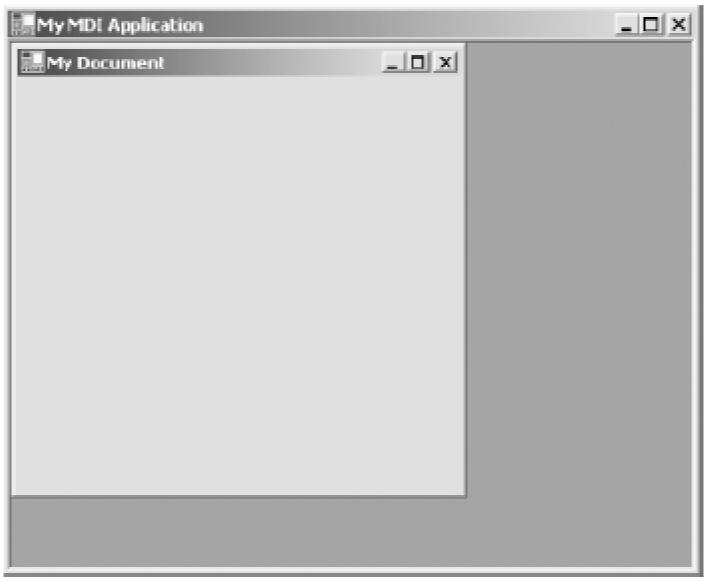
```
Imports System
Imports System.Windows.Forms
Public Module AppModule

    Public Sub Main( )
        Application.Run(New MainForm( ))
    End Sub
End Module

Public Class MainForm
    Inherits Form
    Public Sub New( )
        ' Set the main window caption.
        Text = "My MDI Application"
        ' Set this to be an MDI parent form.
        IsMdiContainer = True
        ' Create a child form.
        Dim myChild As New DocumentForm("My Document", Me)
        myChild.Show
    End Sub
End Class

Public Class DocumentForm
    Inherits Form
    Public Sub New(ByVal name As String, ByVal parent As Form)
        ' Set the document window caption.
        Text = name
        ' Set this to be an MDI child form.
        MdiParent = parent
    End Sub
End Class
```

Assuming that the code in Example 4-6 is saved in a file named *MyApp.vb*, it can be compiled from
the command line with this command:

```
vbc MyApp.vb /r:System.dll,System.Windows.Forms.dll
```

Running the resulting executable produces the display shown in Figure 4-7.

Figure 4-7. A minimal MDI application (the output of the code in Example 4-6)

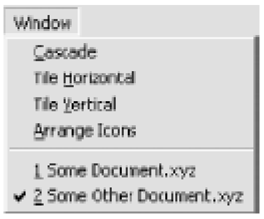


The Form class has two read-only properties related to MDI behavior. The IsMdiChild property returns a Boolean value that indicates whether the form is an MDI child. The MdiChildren property of a parentform contains a collection of references to the form's child forms. The IsMdiChild and MdiChildren properties are both automatically maintained in response to setting the child forms' MdiParent properties.

4.4.2 Creating a Window Menu

MDI applications usually have a main-menu item called Window. On this menu appear standard items for cascading, tiling, and activating child windows and arranging the icons of minimized child windows. Figure 4-8 shows a typical example.

Figure 4-8. A typical Window menu



Such a menu is easy to create using the support in Windows Forms. Assuming that you were to do it programmatically, Example 4-7 shows a revised version of Example 4-6 that has been modified to include a Window menu; the added code is shown in boldface. For details on how to work with menus from the Visual Studio IDE, as well as programmatically, see Section 5.5 in Chapter 5.

Example 4-7. An MDI application with a Window menu

```
Imports System
Imports System.Windows.Forms
Public Module AppModule
Public Sub Main( )
    Application.Run(New MainForm( ))
End Sub
End Module
Public Class MainForm
    Inherits Form
    ' Declare MainForm's main menu
    Private myMainMenu As MainMenu
    ' Declare Windows menu
    Protected WithEvents mnuWindow As MenuItem
    Protected WithEvents mnuTileHoriz As MenuItem
    Protected WithEvents mnuCascade As MenuItem
    Protected WithEvents mnuTileVert As MenuItem
    Protected WithEvents mnuArrangeAll As MenuItem
Public Sub New( )
    ' Set the main window caption.
    Text = "My MDI Application"
    ' Set this to be an MDI parent form.
    IsMdiContainer = True
    ' Create main menu
    MyMainMenu = New MainMenu( )
    ' Define menu items
    mnuWindow = New MenuItem( )
    mnuTileHoriz = New MenuItem( )
    mnuTileVert = New MenuItem( )
    mnuCascade = New MenuItem( )
    mnuArrangeAll = New MenuItem( )
    ' Set menu properties
    mnuWindow.Text = "&Window"
    mnuWindow.MdiList = True
    mnuTileHoriz.Text = "Tile Horizontally"
    mnuTileVert.Text = "Tile Vertically"
    mnuCascade.Text = "Cascade"
    mnuArrangeAll.Text = "Arrange Icons"
    ' Add items to menu
    MyMainMenu.MenuItems.Add(mnuWindow)
    mnuWindow.MenuItems.Add(mnuCascade)
    mnuWindow.MenuItems.Add(mnuTileHoriz)
    mnuWindow.MenuItems.Add(mnuTileVert)
    mnuWindow.MenuItems.Add(mnuArrangeAll)
    ' Assign menu to form
    Me.Menu = MyMainMenu
    ' Create a child form.
    Dim myChild As New DocumentForm("My Document", Me)
    myChild.Show
End Sub
Public Sub mnuCascade_Click(o As Object, e As EventArgs) _
    Handles mnuCascade.Click
    LayoutMdi(MdiLayout.Cascade)
End Sub
Public Sub mnuTileHoriz_Click(o As Object, e As EventArgs) _
    Handles mnuTileHoriz.Click
    LayoutMdi(MdiLayout.TileHorizontal)
End Sub
Public Sub mnuTileVert_Click(o As Object, e As EventArgs) _
    Handles mnuTileVert.Click
    Programming Visual Basic .NET

    LayoutMdi(MdiLayout.TileVertical)
End Sub
Public Sub mnuArrangeAll_Click(o As Object, e As EventArgs) _
    Handles mnuArrangeAll.Click
    LayoutMdi(MdiLayout.ArrangeIcons)
End Sub
End Class
Public Class DocumentForm
    Inherits Form
    Public Sub New(ByVal name As String, ByVal parent As Form)
    ' Set the document window caption.
    Text = name
    ' Set this to be an MDI child form.
    MdiParent = parent
End Sub
End Class
```

To add a Window menu to the parent form of an MDI application, perform the following steps. First, add a menu item to the MDI parent form's main menu, setting its Text property to anything desired (usually Window) and its MdiList property to True. It is the MdiList property that makes the Window menu a Window menu. Setting the MdiList property to True causes the Windows Forms framework to add and delete menu items to and from this menu item as necessary. This in turn will always display the current list of MDI child windows in the menu.Next, add menu items for Cascade, Tile Horizontally, Tile Vertically, and Arrange Icons. In the Click event handler for each of these menu items, call the Form class's LayoutMdi method, passing the appropriate parameter value for the desired action. The syntax of the LayoutMdi method is:
Public Sub LayoutMdi(ByVal value As MdiLayout)
The method's single argument must be a value from the MdiLayout enumeration (defined in the System.Windows.Forms namespace). The values in this enumeration are:ArrangeIcons Indicates that the icons for the minimized MDI child windows should be neatly arranged.
Cascade Indicates that the MDI child windows should be cascaded (displayed overlapping each other).
TileHorizontal Indicates that the MDI child windows should be tiled (displayed without overlapping), with each child window filling the width of the MDI parent.
TileVertical
Indicates that the MDI child windows should be tiled, with each child window filling the height of the MDI parent.

4.4.3 Merging Menus

Often, the items that should appear on an MDI application's main menu are dependent on the type of document being displayed or on whether any document is displayed at all. Of course, this effect could be achieved in code by dynamically adding and removing menu items each time a child window is activated. However, the Windows Forms framework provides an easier way.If an MDI child form has a main menu of its own, it and the MDI parent form's main menu are merged to produce the menu that is shown to the user when the child form is displayed. Two properties of the MenuItem class affect how the menu items are merged. First, the MergeOrder property determines the order in which the menu items are displayed. This property can be set to any Integer value, and the values don't have to be contiguous. The menu items from the two menus are sorted on this value to determine the order in which the menu items are displayed on screen. For example, consider an MDI parent form that has a main menu with three menu items representing File, Window, and Help menus. Further, say that the MergeOrder properties of these menu items are 10, 20, and 30, respectively. Now, if an MDI child form is displayed and its main menu has, for example, an Edit item with a MergeOrder property value of 15, the menu displayed to the user will have four items: File, Edit, Window, and Help, in that order. Example 4-8 shows a revised version of Example 4-6 that contains the code necessary to create such a menu; lines shown in boldface have been added to define the main menu and its menu items.

Example 4-8. An MDI application with merged menus

```
Imports System
Imports System.Windows.Forms
Public Module AppModule
Public Sub Main( )
    Application.Run(New MainForm( ))
End Sub
End Module

Public Class MainForm
Inherits Form
' Declare MainForm's main menu.
Private myMainMenu As MainMenu
' Declare the Window menu.
Protected WithEvents mnuFile As MenuItem
Protected WithEvents mnuWindow As MenuItem
Protected WithEvents mnuHelp As MenuItem

    Public Sub New( )
        ' Set the main window caption.
        Text = "My MDI Application"
        ' Set this to be an MDI parent form.
        IsMdiContainer = True
        ' Create main menu
        MyMainMenu = New MainMenu( )
        ' Define menu items
        mnuFile = New MenuItem( )
        mnuWindow = New MenuItem( )
        mnuHelp = New MenuItem( )
        ' Set menu properties
        mnuFile.Text = "&File"
        mnuFile.MergeOrder = 10
        mnuWindow.Text = "&Window"
        mnuWindow.MergeOrder = 20
        mnuWindow.MdiList = True
        mnuHelp.Text = "&Help"
        mnuHelp.MergeOrder = 30
        ' Add items to menu
        MyMainMenu.MenuItems.Add(mnuFile)
        MyMainMenu.MenuItems.Add(mnuWindow)
        MyMainMenu.MenuItems.Add(mnuHelp)
        ' Assign menu to form
        Me.Menu = MyMainMenu
        ' Create a child form.
        Dim myChild As New DocumentForm("My Document", Me)
        myChild.Show
    End Sub

End Class

Public Class DocumentForm
Inherits Form
    ' Declare menu Private mdiMenu As New MainMenu
    ' Declare menu items Protected WithEvents mnuEdit As MenuItem
    Public Sub New(ByVal name As String, ByVal parent As Form)
        ' Set the document window caption.
        Text = name
        ' Set this to be an MDI child form.
        MdiParent = parent
        ' Instantiate menu and menu items mdiMenu = New MainMenu( ) mnuEdit = New
        MenuItem( )
        ' Set menu properties mnuEdit.Text = "&Edit" mnuEdit.MergeOrder = 15
        ' Add item to main menu mdiMenu.MenuItems.Add(mnuEdit)
        ' Add menu to child window Me.Menu = mdiMenu
    End Sub

End Class
```

If a menu item in the MDI child form menu has the same MergeOrder value as a menu item in the MDI parent form menu, a second property comes into play. The MergeType property of both MenuItem objects is examined, and the behavior is determined by the combination of their values. The MergeType property is of type MenuMerge (an enumeration defined in the System.Windows.Forms namespace) and can have one of the following values:

Add

The menu item appears as a separate item in the target menu, regardless of the setting of the other menu item's MergeType property.
MergeItems If the other menu item's MergeType property is also set to MergeItems, the two menu items are merged into a single item in the target menu. Merging is then recursively applied to the subitems of the source menus, using their MergeOrder and MergeType properties.
If the other menu item's MergeType property is set to Add, both menu items appear in the target menu (just as though both had specified Add). If the other menu item's MergeType property is set to Remove, only this menu item appears in the target menu

(again, the same as specifying Add for this menu item). If the other menu item's MergeType property is set to Replace, only the child form's menu item is displayed, regardless of which one is set to MergeItems and which one is set to Replace. (This seems like inconsistent behavior and may be a bug.)
Remove
The menu item isn't shown in the target menu, regardless of the setting of the other menu item's MergeType property.
Replace
If the other menu item's MergeType property is set to Add, both menu items appear in the target menu (just as though both had specified Add). If the other menu item's MergeType property is set to MergeItems or Replace, only the child form's menu item is shown. (This seems like inconsistent behavior and may be a bug.) If the other menu item's MergeType property is also set to Replace, only the child form's menu item is shown.

4.4.4 Detecting MDI Child Window Activation

Code in the MDI parent form class can be notified when an MDI child form becomes active inside an MDI parent form. ("Active" means that the child form receives the input focus after another MDI child form or the MDI parent form had the input focus.) To receive such notification, the MDI parent form must override the OnMdiChildActivate method (defined in the Form class). For example:

```
' Place this within the class definition of the MDI parent form.

Protected Overrides Sub OnMdiChildActivate(ByVal e As EventArgs)
MyBase.OnMdiChildActivate(e) ' Important
' ...
End Sub
```

It is important to call the base-class implementation of OnMdiChildActivate within the overriding function, so that any necessary base-class processing (including raising of the MdiChildActivate event) can occur. The *e* parameter carries no information. To find out which MDI child form became active, read the ActiveMdiChild property of the MDI parent form. This property is of type Form. Convert it to the MDI child form's type to gain access to any public members that are specific to that type. For example:

```
 Protected Overrides Sub OnMdiChildActivate(ByVal e As EventArgs)
    MyBase.OnMdiChildActivate(e)
    '   Assumes that SomeFormType is defined elsewhere and inherits
    '   from Form. Also assumes that the MDI child forms in the
    '   application are always of this type.
    Dim childForm As SomeFormType = _
    CType(ActiveMdiChild, SomeFormType)
    '   Do something with childForm here.
    ' ...
End Sub
```

To have code outside of the MDI parent form class notified when an MDI child form becomes active, write a handler for the MDI parent form's MdiChildActivate event. This event is defined in the Form class as:

```
Public Event MdiChildActivate( _ByVal sender As Object, _ByVal e As EventArgs
    _)
```

The *sender* parameter is the MDI parent form, not the MDI child form that has been activated. The *e* parameter does not contain any additional information about the event. As when overriding the OnMdiChildActivate method, read the MDI parent form's ActiveMdiChild property to discover which MDI child form has been activated.

4.5 COMPONENT ATTRIBUTES

As explained in Chapter 2, attributes can be added to code elements to provide additional information about those elements. The System.ComponentModel namespace defines several attributes for use in component, control, and form declarations. These attributes don't affect component behavior. Rather, they provide information that is used or displayed by the Visual Studio .NET IDE. The following is a description of each attribute:

AmbientValueAttribute
For ambient properties, specifies the property value that will mean "get this property's actual value from wherever ambient values come from for this property." *Ambient properties* are properties able to get their values from another source. For example, the BackColor property of a Label control can be set either to a specific Color value or to the special value Color.Empty, which causes the Label's background color to be the same as the background color of the form on which it is placed. Putting this attribute on a property definition isn't what causes the property to behave as an ambient property: the control itself must be written such that when the special value is written to the property, the control gets the actual value from the appropriate location. This attribute simply provides the Windows Forms Designer with a way to discover what the special value is. When specifying this attribute, pass the special value to the attribute's constructor. For example, <AmbientValue(0)>.

BindableAttribute
Indicates whether a property is typically usable for data binding. Specify <Bindable(True)> to indicate that the property can be used for data binding or <Bindable(False)> to indicate that the property typically is not used for data binding. This attribute affects how a property is displayed in the IDE, but does not affect whether a property can be bound at runtime. By default, properties are considered not bindable.

BrowsableAttribute
Indicates whether a property should be viewable in the IDE's Properties window. Specify
<Browsable(True)> to indicate that the property should appear in the Properties window or
<Browsable(False)> to indicate that it should not. By default, properties are considered browsable.
CategoryAttribute
Indicates the category to which a property or event belongs ("Appearance," "Behavior," etc.).
The IDE uses this attribute to sort the properties and events in the Properties window. Specify
the category name as a string argument to the attribute. For example, <Category("Appearance")>. The argument can be any string. If it is not one of the standard strings, the Properties window will add a new group for it. The standard strings are:

Action
Used for events that indicate a user action, such as the Click event.

Appearance
Used for properties that affect the appearance of a component, such as the BackColor property.
Behavior Used for properties that affect the behavior of a component, such as the AllowDrop property.
Data.Used for properties that relate to data, such as the DecimalPlaces property of the NumericUpDown control.
Design. Used for properties that relate to the design-time appearance or behavior of a component.

DragDrop
Used for properties and events that relate to drag and drop. No Windows Forms components have any properties or events marked with this category.

Focus
Used for properties and events that relate to input focus, such as the CanFocus property and the GotFocus event.

Format
Used for properties and events that relate to formats. No Windows Forms components have any properties or events marked with this category.

Key
Used for events that relate to keyboard input, such as the KeyPress event.

Layout
Used for properties and events that relate to the visual layout of a component, such as the Height property and the Resize event.

Mouse
Programming Visual Basic .NET
Used for events that relate to mouse input, such as the MouseMove event.

WindowStyle
Used for properties and events that relate to the window style of top-level forms. No Windows Forms components have any properties or events marked with this category.
If no CategoryAttribute is specified, the property is considered to have a category of Misc.

DefaultEventAttribute
Indicates the name of the event that is to be considered the default event of a component. For example, <DefaultEvent("Click")>. When a component is double-clicked in the Windows Forms Designer, the designer switches to code view and displays the event handler for the default event. This attribute can be used only on class declarations.
DefaultPropertyAttribute Indicates the name of the property that is to be considered the default property of a component. For example, <DefaultProperty("Text")>. This attribute can be used only on class declarations. The default property is the property that the Windows Forms Designer highlights in the Properties window when a component is clicked in design view. Don't confuse this usage of the term default property with the usage associated with the Default modifier of a property declaration. The two concepts are unrelated. Refer to Chapter 2 for details on the Default modifier.
DefaultValueAttribute Indicates the default value of a property. For example, <DefaultValue(0)>. If the IDE is used to set a property value to something other than the default value, the code generator will generate the appropriate assignment statement in code. However, if the IDE is used to set a property to the default value, no assignment statement is generated.DescriptionAttribute Provides a description for the code element. For example, <Description("The text contained in the control.")>. The IDE uses this description in tool tips and IntelliSense.DesignerAttribute Identifies the class that acts as the designer for a component. For example,
<Designer("MyNamespace.MyClass")>. The designer class must implement the IDesigner interface. This attribute can be used only on class declarations and is needed only if the built-in designer isn't sufficient. Creating custom designers is not discussed in this book.
DesignerCategoryAttribute
Used with custom designers to specify the category to which the class designer belongs.

DesignerSerializationVisibilityAttribute
Used with custom designers to specify how a property on a component is saved by the designer.

DesignOnlyAttribute
Indicates when a property can be set. Specify <DesignOnly(True)> to indicate that the property can be set at design time only or <DesignOnly(False)> to indicate that the property can be set at both design time and runtime (the default).

EditorAttribute
Identifies the "editor" to use in the IDE to allow the user to set the values of properties that have the type on which the EditorAttribute attribute appears. In this way, a component can declare new types and can declare properties having those types, yet still allow the user
to set the values of the properties at design time. Creating custom type editors is not discussed in this book.

EditorBrowsableAttribute
Indicates whether a property is viewable in an editor. The argument to the attribute's constructor must be one of the values defined by the EditorBrowsableState

enumeration (defined in the System.ComponentModel namespace). The values of this enumeration are: Advanced Only advanced users should see the property. It is up to the editor to determine when it's appropriate to display advanced properties.

Always
The property should always be visible within the editor.

Never
The property should never be shown within the editor.
ImmutableObjectAttribute
Indicates whether a type declaration defines a state that can change after an object of that type is constructed. Specify <ImmutableObject(True)> to indicate that an object of the given type is immutable. Specify <ImmutableObject(False)> to indicate that an object of the given type is not immutable. The IDE uses this information to determine whether to render a property as read-only.

InheritanceAttribute
Used to document an inheritance hierarchy that can be read using the
IInheritanceService interface. This facility is not discussed in this book.
InstallerTypeAttribute
Used on type declarations to specify the installer for the type. Installers are not discussed in this book.

LicenseProviderAttribute
Indicates the license provider for a component.
ListBindableAttribute
Indicates whether a property can be used as a data source. (A *data source* is any object that exposes the IList interface.) Specify <ListBindable(True)> to indicate that the property can be used as a data source. Specify <ListBindable(False)> to indicate that the
property can't be used as a data source.

LocalizableAttribute
Indicates whether a property's value should be localized when the application is localized.
Specify <Localizable(True)> to indicate that the property's value should be localized.
Specify <Localizable(False)> or omit the LocalizableAttribute attribute to indicate that the property's value should not be localized. The values of properties declared with <Localizable(True)> are stored in a resource file, which can be localized.

MergablePropertyAttribute
Indicates where property attributes can be merged. By default, when two or more components are selected in the Windows Forms Designer, the Properties window typically shows the properties that are common to all of the selected components. If the user changes a value in the Properties window, the value is changed for that property in all of the selected components.
Placing <MergableProperty(False)> on a property declaration changes this behavior. Any property declared in this way is omitted from the Properties window when two or more components are selected. Specifying <MergableProperty(True)> is the same as omitting the attribute altogether.

NotifyParentPropertyAttribute
Indicates whether the display of a property's *parent property* should be refreshed when the given property changes its value.

ParenthesizePropertyNameAttribute
Indicates whether the property name should be parenthesized in the Properties window.
Specify <ParenthesizePropertyName(True)> to indicate that the property name should appear within parentheses. Specify <ParenthesizePropertyName(False)> to indicate that the property name should not appear within parentheses. Omitting the attribute is the same as specifying <ParenthesizePropertyName(False)>.The only benefit to parenthesizing property names in the property window is that they are sorted to the top of the list. Microsoft has no specific recommendations for when to parenthesize a property name.

PropertyTabAttribute
Specifies a type that implements a custom property tab (or tabs) for a component. This facility is not discussed in this book.

ProvidePropertyAttribute
Used with *extender providers*—i.e., classes that provide properties for *other* objects. Extender providers are not discussed in this book.

ReadOnlyAttribute
Indicates whether a property is read-only at design time. Specify <ReadOnly(True)> to indicate that the property is read-only at design time. Specify <ReadOnly(False)> to indicate that the property's ability to be modified at design time is determined by whether a Set method is defined for the property. Omitting the attribute is the same as specifying <ReadOnly(False)>.

RecommendedAsConfigurableAttribute
Indicates whether a property is configurable. Configurable properties aren't discussed in this book.
RefreshPropertiesAttribute
Determines how the Properties window is refreshed when the value of the given property changes.

RunInstallerAttribute
Indicates whether an installer should be invoked during installation of the assembly that contains the associated class. This attribute can be used only on class declarations, and the associated class must inherit from the Installer class (defined in the System.Configuration.Install namespace). Installers are not discussed in this book.

ToolboxItemAttribute
Specifies a type that implements a toolbox item related to the declaration on which the attribute is placed. This facility is not discussed in this book.

ToolboxItemFilterAttribute
Specifies a filter string for a toolbox-item filter. This facility is not discussed in this book.

TypeConverterAttribute
Indicates the type converter to be used with the associated item. Type converters are not discussed in this book.

4.6 2-D GRAPHICS PROGRAMMING WITH GDI+

The Windows operating system has always included support for drawing two-dimensional graphics. This support is known as the Graphics Device Interface (GDI) library. The GDI library is now easier to use and provides additional features. The new capabilities are known collectively as GDI+. GDI+ features are exposed in the .NET Framework through classes in the System.Drawing, System.Drawing.Drawing2D,System.Drawing.Imaging, and System.Drawing.Text namespaces. This section discusses some of those capabilities.

4.6.1 The Graphics Class

Objects of type Graphics (defined in the System.Drawing namespace) represent two-dimensional surfaces on which to draw. A Graphics object must be obtained before any drawing can be done. A common way to obtain a Graphics object is to override the OnPaint method of a form or user control, as shown in the following code fragment:

```
Public Class MyControl
Inherits UserControl
Protected Overrides Sub OnPaint(ByVal e As PaintEventArgs)
e.Graphics.FillEllipse(New SolidBrush(Me.ForeColor), _
Me.ClientRectangle)
End Sub
Public Sub New( )
Me.ResizeRedraw = True
```

```
    End Sub
    End Class
```

The single argument passed to the OnPaint method, *e*, is of type PaintEventArgs. This class has a property called Graphics, which holds a reference to the Graphics object to be used for drawing on the user control or form. The PaintEventArgs class is defined in the System.Windows.Forms namespace. It has two properties:

ClipRectangle
Defines the area that needs to be drawn. Drawing done outside the limits of the clip rectangle will not be displayed. The coordinates of the rectangle are relative to the client rectangle of the user control or form. The syntax of the ClipRectangle property is:

Public ReadOnly Property ClipRectangle( ) As System.Drawing.Rectangle

Graphics
Defines the graphics surface on which to draw. The syntax of the Graphics property is:
Public ReadOnly Property Graphics( ) As System.Drawing.Graphics
The following list shows some of the Graphics class's many methods that are available for drawing various lines and shapes, and Example 5-7 in Chapter 5 gives an example of drawing a filled ellipse. This list is just to get you started; it is beyond the scope of this book to document the syntax of each of these methods.

DrawArc
Draws an arc (that is, a portion of an ellipse).
DrawBezier
Draws a Bezier curve.
DrawBeziers
Draws a series of Bezier curves.
DrawClosedCurve

Is the same as the DrawCurve method (see the next item in this list), except that the last point in the curve is connected back to the first point.

DrawCurve
Draws a smooth, curved figure that passes through a given array of points.

DrawEllipse
Draws an ellipse.

DrawIcon
Draws an icon. Icons are represented by objects of type Icon (defined in the System.Drawing namespace). The Icon class defines various methods for loading icons.

DrawIconUnstretched
Is the same as the DrawIcon method, but does not stretch the icon to fit the clipping rectangle.

DrawImage
Draws an image. Images are represented by objects of type Image (defined in the System.Drawing namespace). The Image class defines various methods for loading images in standard formats, such as bitmaps and JPEGs.

DrawImageUnscaled
Is the same as DrawImage, except that the DrawImageUnscaled method ignores any width and height parameters passed to it.

DrawLine
Draws a line.

DrawLines
Draws a series of lines.

DrawPath

Draws a series of lines and curves that are defined by a GraphicsPath object. The GraphicsPath class is beyond the scope of this book.

DrawPie
Draws a pie section.

DrawPolygon
Draws lines to connect a series of points.

DrawRectangle
Draws a rectangle.

DrawRectangles
Draws a series of rectangles.

DrawString
Draws text.

FillClosedCurve
Draws a filled, closed curve.

FillEllipse
Draws a filled ellipse.

FillPath
Draws a filled figure whose shape is given by a GraphicsPath object. The GraphicsPath class is beyond the scope of this book.

FillPie
Draws a filled pie section.

FillPolygon
Draws a filled polygon (see the DrawPolygon method earlier in this list).

FillRectangle
Draws a filled rectangle.

FillRectangles
Draws a series of filled rectangles.

FillRegion
Draws a filled figure whose shape is given by a Region object.

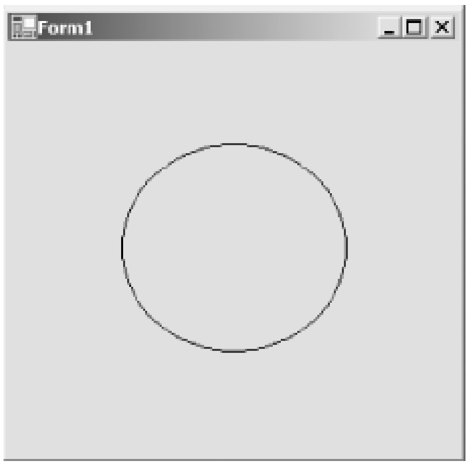
4.6.2 The Pen Class

Pen objects hold the settings used when drawing lines. All of the Graphics class's Draw...methods (DrawArc, DrawBezier, etc.) require that the caller supply a Pen object. The supplied Pen object determines the properties of the line used for drawing (for example, its color, width, etc.). Example 4-9 shows an OnPaint method that can be used to draw an ellipse on a user control or a form. It is similar to the code in Example 5-6 in Chapter 5, but displays the ellipse a little smaller, and with only a border. The resulting display is shown in Figure 4-9.

Example 4-9. Drawing an ellipse on a form

```
Protected Overrides Sub OnPaint(ByVal e As PaintEventArgs)
Dim pn As New Pen(Me.ForeColor)
Dim rect As Rectangle
    rect.X = Me.ClientRectangle.X + (Me.ClientRectangle.Width \ 4)
    rect.Y = Me.ClientRectangle.Y + (Me.ClientRectangle.Height \ 4)
    rect.Width = Me.ClientRectangle.Width \ 2
    rect.Height = Me.ClientRectangle.Height \ 2
    e.Graphics.DrawEllipse(pn, rect)
    pn.Dispose( )
End Sub
```

Figure 4-9. The ellipse drawn by the code in Example 4-9



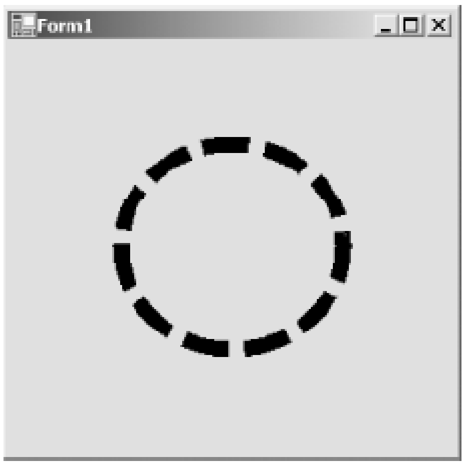
In Example 4-9, the Graphics class's DrawEllipse method is passed a Pen object, which determines the appearance of the line used for drawing the ellipse, and a rectangle, which defines the shape of the ellipse. The Pen class has four constructors. The constructor used in Example 4-9 takes a parameter of type Color (defined in System.Drawing). The color passed to the Pen class constructor in Example 4-9 is the foreground color of the form (Me.ForeColor). This is a nice touch ensuring that the ellipse will be drawn using whatever color is set as the foreground color of the form on which the ellipse is drawn. See Section 4.6.4 later in this chapter for information on choosing and manipulating colors. Finally, note this line in Example 4-9:

pn.Dispose( ) By convention, objects that allocate scarce resources expose a Dispose method to allow the object client to tell the object to release its resources. When using any object that exposes a Dispose method (as the Pen object does), the Dispose method must be called when the client code is finished using the object. If the Dispose method isn't called (or if it isn't implemented), resources will be held longer than necessary, which may in turn result in resources being unavailable for other code that needs them. The .NET Framework provides a number of predefined pens through the properties of the Pens and SystemPens classes (defined in the System.Drawing namespace). For example, the Blue property of the Pens class returns a Pen object whose color is set to Color.Blue. Thus, the following line of code draws a blue ellipse:
e.Graphics.DrawEllipse(Pens.Blue, rect) Similarly, the SystemPens class's WindowText property returns a Pen object whose color is set to the system's window text color. Using the standard pens provided by the Pens and SystemPens classes can be more efficient than instantiating new Pen objects. However, their properties (such as line width) cannot be altered. See Table 4-1, later in this chapter, for the list of Pen objects available through the Pens class. See Section 4.6.4.1 in Section 4.6.4 later in this chapter for the list of Pen objects available through the SystemPens class.
When working with a user-instantiated pen, you can modify the line that is drawn by setting properties of the Pen object. The code in Example 4-10 sets the Pen object's Width property to widen the outline of the ellipse. The lines of code that differ from Example 4-9 are shown in bold. The resulting display is shown in Figure 4-10.

Example 4-10. Setting Pen properties

```
Protected Overrides Sub OnPaint(ByVal e As PaintEventArgs)
Dim pn As New Pen(Me.ForeColor)

    pn.Width = 10
    pn.DashStyle = Drawing.Drawing2D.DashStyle.Dash
    Dim rect As Rectangle
    rect.X = Me.ClientRectangle.X + (Me.ClientRectangle.Width \ 4)
    rect.Y = Me.ClientRectangle.Y + (Me.ClientRectangle.Height \ 4)
    rect.Width = Me.ClientRectangle.Width \ 2
    rect.Height = Me.ClientRectangle.Height \ 2
    e.Graphics.DrawEllipse(pn, rect)
    pn.Dispose( )
End Sub
```

Figure 4-10. The ellipse drawn by the code in Example 4-10



Example 4-10 sets the Pen object's Width and DashStyle properties to attain the effect shown in **Figure 4-10**.

The Width property is a value of type Single that determines the width of lines drawn with this pen. The default is 1. The unit of measurement is determined by the PageUnit property of the Graphics object in which the lines are drawn. The PageUnit property is of the enumeration type GraphicsUnit (defined in the System.Drawing namespace). The values of GraphicsUnit that are appropriate for assignment to the PageUnit property are:

Display
Units are specified in 1/75 of an inch.

Document
Units are specified in 1/300 of an inch.

Inch
Units are specified in inches.

Millimeter
Units are specified in millimeters.

Pixel
Units are specified in pixels.

Point
Units are specified in points (1/72 of an inch).

The DashStyle property of the Pen object determines the whether the line is solid or dashed, as well as the style of the dash. The DashStyle property is of the enumeration type DashStyle (defined in the System.Drawing.Drawing2D namespace), which defines the following values:

Custom
Specifies a programmer-defined dash style. If this value is used, other properties of the Pen object control the exact appearance of the dashes in the line. Creating custom dash styles is not discussed in this book.

Dash
Specifies a dashed line.

DashDot
Specifies a line consisting of alternating dashes and dots.

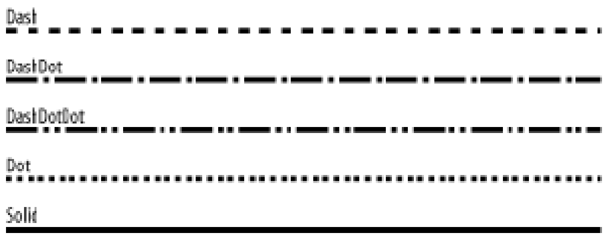
DashDotDot
Specifies a line consisting of alternating dashes and two dots.

Dot
Specifies a dotted line.

Solid
Specifies a solid line.

The standard dash styles are shown in Figure 4-11.

Figure 4-11. The standard DashStyle values



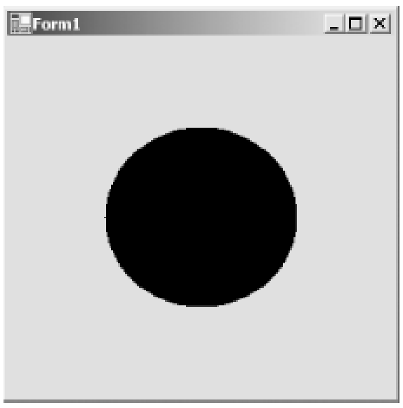
4.6.3 The Brush Class

Brush objects hold the settings used when filling graphics areas. All of the Graphics class's
Fill...methods (FillClosedCurve, FillEllipse, etc.) require that the caller supply a Brush object. The
supplied Brush object determines how the interior of the figure will be painted. Example 4-11 shows an OnPaint method that can be used to draw an ellipse on a user control or a form. It is similar to Example 4-9, but draws a filled ellipse rather than an outline. The lines that differ from Example 4-9 are shown in bold. The resulting display is shown in Figure 4-12.

Example 4-11. Drawing a filled ellipse on a form

```
Protected Overrides Sub OnPaint(ByVal e As PaintEventArgs)

    Dim br As New SolidBrush(Me.ForeColor)
    Dim rect As Rectangle
    rect.X = Me.ClientRectangle.X + (Me.ClientRectangle.Width \ 4)
    rect.Y = Me.ClientRectangle.Y + (Me.ClientRectangle.Height \ 4)
    rect.Width = Me.ClientRectangle.Width \ 2
    rect.Height = Me.ClientRectangle.Height \ 2
    e.Graphics.FillEllipse(br, rect)
    br.Dispose( )
End Sub
```

Figure 4-12. The ellipse drawn by the code in Example 4-11



Note that Example 4-11 is not entirely parallel to Example 4-9. Specifically,
Example 4-9
instantiated a Pen object directly, but Example 4-11 instantiates an object from a class that derives from the Brush class rather than directly from the Brush class. Objects

can't be directly instantiated from the Brush class. The classes that derive from Brush are:

HatchBrush
Fills an area with a hatch pattern. Hatch patterns are patterns of lines and spaces. The HatchBrush class's HatchStyle property determines the exact pattern of the hatch. It is defined in the System.Drawing.Drawing2D namespace.

LinearGradientBrush
Fills an area with a gradient blend of two or more colors. It is defined in the System.Drawing.Drawing2D namespace.

PathGradientBrush
Fills the internal area of a GraphicsPath object with a gradient blend of two or more colors. It is defined in the System.Drawing.Drawing2D namespace.

SolidBrush
Fills an area with a solid color. It is defined in the System.Drawing namespace.

TextureBrush
Fills an area with an image. It is defined in the System.Drawing namespace.

The .NET Framework provides a number of predefined brushes through the properties of the Brushes and SystemBrushes classes (defined in the System.Drawing namespace). *For example*, the Blue property of the Brushes class returns a Brush object that fills areas with solid blue. Thus, the following line of code draws a solid blue ellipse:

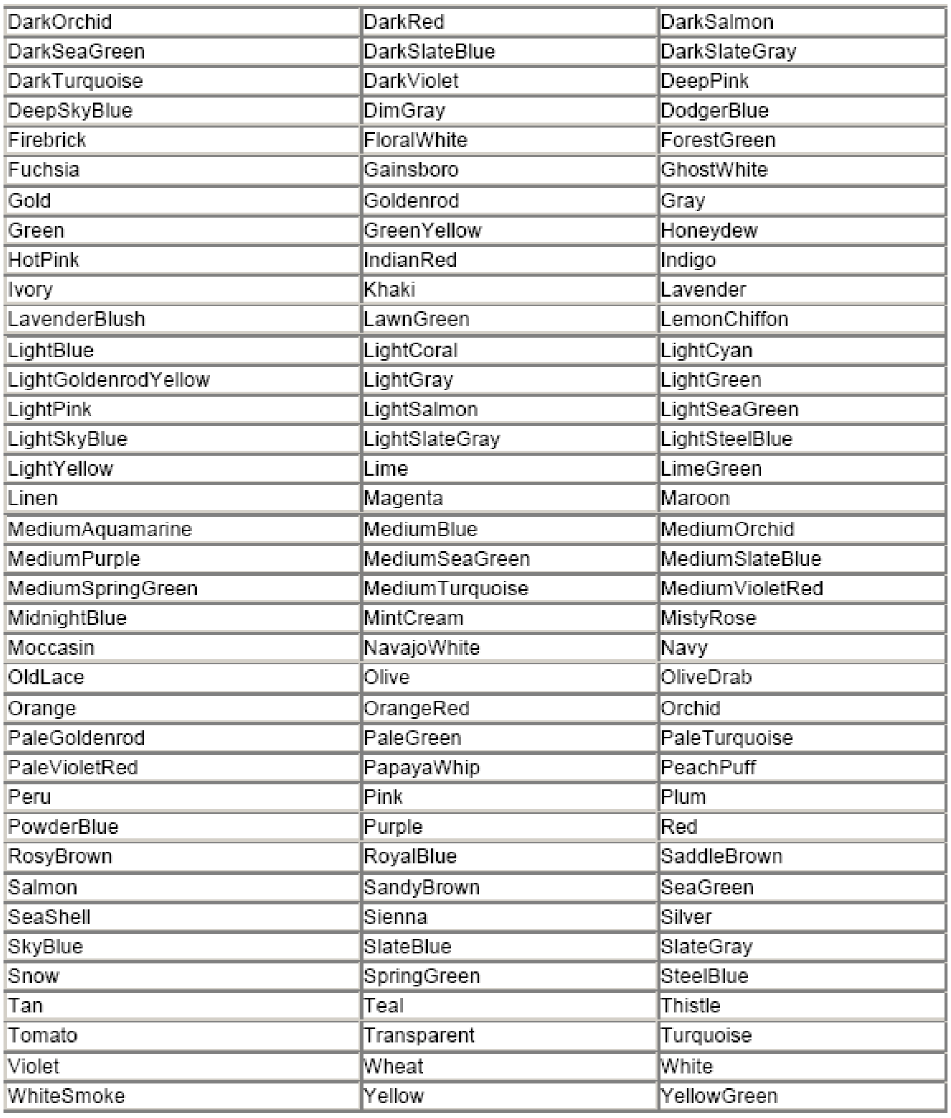
```
    e.Graphics.FillEllipse(Brushes.Blue, rect)
```

Similarly, the SystemBrushes class's Window property returns a Brush object whose color is set to the background color of the system's window client area. Using the standard brushes provided by the Brushes and SystemBrushes classes can be more efficient than instantiating new Brush objects However, their properties cannot be altered.

4.6.4 The Color Structure

Colors are represented by values of type Color. The Color structure defines 141 named colors and exposes them as shared read-only properties whose values are of type Color. They serve the purpose of color constants. For example, the following code fragment sets the background color of the form frm to white:
frm.BackColor = Color.White
The color properties exposed by the Color structure have the same names as the pen properties exposed by the Pens class and the brush properties exposed by the Brushes class. The list is lengthy, so it is printed here only once, in Table 4-1.

Table 4-1. Properties common to the Color, Pens, and Brushes classes



DarkOrchid	DarkRed	DarkSalmon
DarkSeaGreen	DarkSlateBlue	DarkSlateGray
DarkTurquoise	DarkViolet	DeepPink
DeepSkyBlue	DimGray	DodgerBlue
Firebrick	FloralWhite	ForestGreen
Fuchsia	Gainsboro	GhostWhite
Gold	Goldenrod	Gray
Green	GreenYellow	Honeydew
HotPink	IndianRed	Indigo
Ivory	Khaki	Lavender
LavenderBlush	LawnGreen	LemonChiffon
LightBlue	LightCoral	LightCyan
LightGoldenrodYellow	LightGray	LightGreen
LightPink	LightSalmon	LightSeaGreen
LightSkyBlue	LightSlateGray	LightSteelBlue
LightYellow	Lime	LimeGreen
Linen	Magenta	Maroon
MediumAquamarine	MediumBlue	MediumOrchid
MediumPurple	MediumSeaGreen	MediumSlateBlue
MediumSpringGreen	MediumTurquoise	MediumVioletRed
MidnightBlue	MintCream	MistyRose
Moccasin	NavajoWhite	Navy
OldLace	Olive	OliveDrab
Orange	OrangeRed	Orchid
PaleGoldenrod	PaleGreen	PaleTurquoise
PaleVioletRed	PapayaWhip	PeachPuff
Peru	Pink	Plum
PowderBlue	Purple	Red
RosyBrown	RoyalBlue	SaddleBrown
Salmon	SandyBrown	SeaGreen
SeaShell	Sienna	Silver
SkyBlue	SlateBlue	SlateGray
Snow	SpringGreen	SteelBlue
Tan	Teal	Thistle
Tomato	Transparent	Turquoise
Violet	Wheat	White
WhiteSmoke	Yellow	YellowGreen

4.6.4.1 System colors

It is useful to discover the colors that Windows uses to draw specific window elements, such as the active window's title bar. If the color itself is required, it can be obtained from the SystemColors class. If a pen or brush of the appropriate color is needed, the pen or brush can be obtained from the corresponding property of the Pens or Brushes class, respectively. The property names exposed by these three classes overlap and, therefore, are presented here in a single list:

ActiveBorder
The color of the filled area of the border of the active window. (Not available on the Pens class.)

ActiveCaption
The background color of the title bar of the active window. (Not available on the Pens class.)

ActiveCaptionText
The text color in the title bar of the active window.

AppWorkspace
The background color of MDI parent windows. (Not available on the Pens class.)

Control
The background color of controls.

ControlDark
The shadow color of controls (for 3-D effects).

ControlDarkDark
The very dark shadow color of controls (for 3-D effects).

ControlLight
The highlight color of controls (for 3-D effects).

ControlLightLight
The very light highlight color of controls (for 3-D effects).

ControlText
The color of text on controls.

Desktop
The color of the Windows desktop. (Not available on the Pens class.)

GrayText
The text color of disabled controls or other disabled visual elements. (Not available on the Brushes class.)

Highlight
The background color of highlighted (selected) text.

HighlightText
The text color of highlighted (selected) text.

HotTrack
The background color of a *hot tracked* item. Hot tracking is highlighting an item as the mouse moves over it. Windows menus use hot tracking. (Not available on the Pens class.)

InactiveBorder
The color of the filled areas of the borders of inactive windows. (Not available on the Pens class.)

InactiveCaption
The background color of the title bars of inactive windows. (Not available on the Pens class.)

InactiveCaptionText
The text color in the title bars of inactive windows. (Not available on the Brushes class.)

Info
The background color of tool tips. (Not available on the Pens class.)

InfoText
The text color of tool tips. (Not available on the Brushes class.)

Menu
The background color of menus. (Not available on the Pens class.)

MenuText
The text color of menus. (Not available on the Brushes class.)

ScrollBar
The color of scroll bars in the area not occupied by the scroll box (or *thumb*). (Not available on the Pens class.)

Window
The background color of the client areas of windows. (Not available on the Pens class.)

WindowFrame
The color of the frames surrounding windows. (Not available on the Brushes class.)

WindowText
The color of the text in the client areas of windows.
Note that some of these properties aren't available on either the Pens class or the Brushes class. In such cases, it is still possible to get a Pen or Brush object of the appropriate color by instantiating a new Pen or Brush object, passing to its constructor the desired color value, like this:

```
Dim br As New SolidBrush(SystemColors.InfoText)
```

4.6.5 Alpha Blending

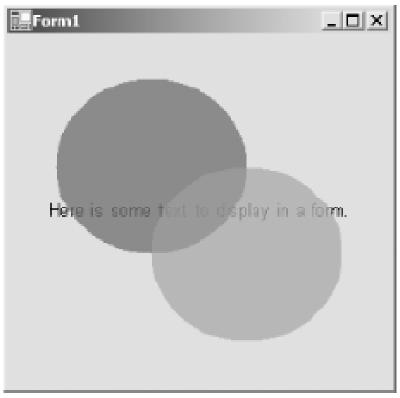
Alpha blending is a process that allows a Graphics object to appear transparent, causing Graphics objects beneath it to be seen through the object. The degree of transparency can be controlled in steps from completely transparent (invisible) to completely opaque (obscuring any objects beneath it). To draw a transparent object, instantiate Pen and Brush objects having colors whose *alpha* component is less than the maximum value of 255. A color's alpha component is given by the A property of the Color structure. This property is a Byte, so it can take values from 0 (invisible) to 255 (completely opaque). Example 4-12 shows an OnPaint method that draws text and then draws two overlapping, transparent ellipses in the same space as the text. Normally, the ellipses would obscure the text, and the second ellipse would obscure the first. In this case, however, the text can be seen through the ellipses because the ellipses are transparent, and the first ellipse can be seen through the second.
The result is shown in Figure 4-13.

Example 4-12. Drawing transparent figures

```
Protected Overrides Sub OnPaint(ByVal e As PaintEventArgs)

    ' Determine the text to display and its font.
                Dim str As String = "Here is some text to display in a form."
                Dim fnt As New Font("Arial", 10, FontStyle.Regular,
                GraphicsUnit.Point)

      ' Determine the X and Y coordinates at which to draw the text so
      ' that it is centered in the window.
                Dim szf As SizeF = e.Graphics.MeasureString(str, fnt)
                Dim xText As Single = (Me.DisplayRectangle.Width - szf.Width) / 2
                Dim yText As Single = (Me.DisplayRectangle.Height - szf.Height) / 2

      ' Draw the text.
                e.Graphics.DrawString(str, fnt, Brushes.Black, xText, yText)

      ' Create a blue brush that is mostly transparent.

                Dim br As New SolidBrush(Color.FromArgb(160, Color.Blue))

      ' Determine the bounding rectangle for the first ellipse.

                Dim rect As Rectangle
                rect.X = Me.DisplayRectangle.X + (Me.DisplayRectangle.Width \ 8)
                rect.Y = Me.DisplayRectangle.Y + (Me.DisplayRectangle.Height \ 8)
                rect.Width = Me.DisplayRectangle.Width \ 2
                rect.Height = Me.DisplayRectangle.Height \ 2

      ' Draw the first ellipse.

                e.Graphics.FillEllipse(br, rect)

      ' Release the brush.

                br.Dispose( )
```

```
        ' Create a red brush that is mostly transparent.
                br = New SolidBrush(Color.FromArgb(160, Color.Red))

        ' Determine the bounding rectangle for the second ellipse.
                rect.X += (Me.DisplayRectangle.Width \ 4)
                rect.Y += (Me.DisplayRectangle.Height \ 4)

        ' Draw the second ellipse.
                e.Graphics.FillEllipse(br, rect)

        ' Release the brush.
                br.Dispose( )
    End Sub
```

Figure 4-13. The display drawn by the code in Example 4-12



4.6.6 Antialiasing

Antialiasing is a technique for making the edges of graphics figures appear less jagged. To turn on antialiasing, set the Graphics object's SmoothingMode property to SmoothingMode.AntiAlias.
(SmoothingMode is an enumeration defined in the System.Drawing.Drawing2D namespace.) Compare the arcs shown in Figure 4-14. Both arcs were drawn by calling the DrawArc method of the Graphics class, but the arc on the left was drawn with the SmoothingMode property set to
SmoothingMode.None (the default), and the arc on the right was drawn with the SmoothingMode
property set to SmoothingMode.AntiAlias. Figure 4-15 shows a close-up comparison view of
the upper portion of both arcs.

Figure 4-14. Nonantialiased versus antialiased arcs

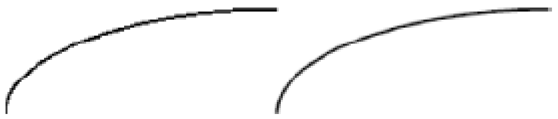


Figure 4-15. Close-up view of nonantialiased and antialiased arcs

As Figure 4-15 shows, antialiasing appears to improve pixel resolution by using gradient shades of the color being rendered and of the background color (in this case, black and white, respectively). The downside to antialiasing is that it takes more time to render.

4.7 PRINTING

Most Visual Basic .NET programs will never need to use the .NET Framework's native printing capabilities. Reporting tools such as Crystal Reports, as well as RAD tools for laying out reports, provide most of the printing facilities that typical Visual Basic .NET programs need. However, for the cases in which a reporting tool is not flexible enough, this section describes the .NET Framework's support for outputting text and graphics directly to a printer.

4.7.1 Hello, Printer!

Example 4-13 shows a minimal printing example.

Example 4-13. Hello, Printer!

```
Imports System.Drawing
Imports System.Drawing.Drawing2D
Imports System.Drawing.Printing
' ...
' These two lines initiate printing. Place this code in an
' appropriate place in the application.
Dim pd As New HelloPrintDocument( )
pd.Print( )
' ...
' This class manages the printing process.

Public Class HelloPrintDocument
    Inherits PrintDocument

        Protected Overrides Sub OnPrintPage(ByVal e As PrintPageEventArgs)
            MyBase.OnPrintPage(e)
            ' Draw text to the printer graphics device.
            Dim fnt As New Font("Arial", 10, FontStyle.Regular, _
            GraphicsUnit.Point)
            e.Graphics.DrawString("Hello, Printer!", fnt, Brushes.Black, 0, 0)
            fnt.Dispose( )
            ' Indicate that there are no more pages.
            e.HasMorePages = False
        End Sub

End Class
```

Printing is managed by defining a class that inherits from the PrintDocument class (defined in the
System.Drawing.Printing namespace). Printing is initiated by instantiating the derived class and calling its Print method (inherited from the PrintDocument class). The Print method repeatedly calls the OnPrintPage method, until the HasMorePages property of the PrintPageEventArgs parameter is set to False. It is the job of the OnPrintPage method to generate each page of output that is sent to the printer. Take a closer look at the OnPrintPage method in Example 4-13, starting with the first line:
MyBase.OnPrintPage(e)
This line of code calls the OnPrintPage method implemented by the base PrintDocument class,
passing it the same argument that was passed into the derived class's OnPrintPage method. This call is important because the PrintDocument class's OnPrintPage method is responsible for firing the PrintDocument object's PrintPage event. If this is not done, any code that has registered for this event will not receive it. This is not an issue in this small example, but it's nevertheless good programming practice. The next three lines of code are responsible for handling the printing:

```
' Draw text to the printer graphics device.
    Dim fnt As New Font("Arial", 10, FontStyle.Regular, _
    GraphicsUnit.Point)
    e.Graphics.DrawString("Hello, Printer!", fnt, Brushes.Black, 0, 0)
    fnt.Dispose( )
```

This code draws some text to the Graphics object provided through the PrintPageEventArgs parameter. Everything you learned in the previous section about the Graphics object is applicable here.Finally, since we're printing just one page in our example program, PrintPageEventArgs.HasMorePages is set to False:

```
' Indicate that there are no more pages.
e.HasMorePages = False
```

This line indicates to the Print method that the end of the document has been reached. If more pages need to be printed, the OnPrintPage method should set the HasMorePages property to True.

4.7.2 The PrintPageEventArgs Class

The PrintPageEventArgs class is declared in the System.Drawing.Printing namespace as:

```
Public Class PrintPageEventArgs
Inherits System.EventArgs
```

Its properties are:

Cancel

An indication of whether the print job is being canceled. This property is set to True by the printing system if the user cancels the print job. Code in the OnPrintPage method can read this value and take any appropriate action. However, programmatically setting this property
back to False does not resume the print job. On the other hand, programmatically setting it to True does cancel the print job, even if the user has not clicked the Cancel button. The syntaxof the Cancel property is:

```
Public Property Cancel( ) As Boolean
```

Graphics

The Graphics object that represents the page surface. The syntax of the Graphics property is:

```
Public ReadOnly Property Graphics( ) As System.Drawing.Graphics
```

HasMorePages

The mechanism for the OnPrintPage method to indicate to the printing system whether there are more pages to be printed after the current page. The OnPrintPage method should set this property to True when there are more pages to print and to False when there are no more pages to print. The syntax of the HasMorePages property is:

```
Public Property HasMorePages( ) As Boolean
```

MarginBounds

A rectangle that specifies the area of the page that is within the document margins (i.e., the area of the page on which rendering should occur). The syntax of the MarginBounds property is:

```
Public ReadOnly Property MarginBounds( ) As System.Drawing.Rectangle
```

PageBounds

A rectangle that specifies the full area of the page, including the area outside the margins. The syntax of the PageBounds property is:

```
Public ReadOnly Property PageBounds( ) As System.Drawing.Rectangle
```

PageSettings

The page settings that apply to the page currently being printed. The syntax of the PageSettings property is:

```
Public ReadOnly Property PageSettings( ) As _
System.Drawing.Printing.PageSettings
```

The PageSettings class is described later in this section.

4.7.3 The OnBeginPrint and OnEndPrint Methods

The PrintDocument class provides the OnBeginPrint and OnEndPrint methods for managing the start and finish of print jobs. The OnBeginPrint method is called prior to the first call to OnPrintPage, and the OnEndPrint method is called after the final call to OnPrintPage. The OnBeginPrint method is a good place to set up objects that will be used throughout the life of the print job—pens, brushes, and fonts, for example. The HelloPrintDocument class in Example 4-13 instantiates a Font object during the OnPrintPage method. This is acceptable here because only one page is being printed. However, in practice documents may contain many pages, so it is better to move this code to the OnBeginPrint method. Example 4-14 shows how the helloPrintDocument looks when modified in this way.

Example 4-14. Using OnBeginPrint and OnEndPrint to set up and tear down objects used during printing

```
Public Class HelloPrintDocument
Inherits PrintDocument
' Private member to hold the font that will be used for printing.

Private m_fnt As Font
    Protected Overrides Sub OnBeginPrint(ByVal e As PrintEventArgs)
        MyBase.OnBeginPrint(e)
        ' Create the font that will be used for printing.
        m_fnt = New Font("Arial", 10, FontStyle.Regular, _
        GraphicsUnit.Point)
        End Sub
        Protected Overrides Sub OnEndPrint(ByVal e As PrintEventArgs)
        MyBase.OnEndPrint(e)
        ' Release the font.
        m_fnt.Dispose( )
    End Sub

    Protected Overrides Sub OnPrintPage(ByVal e As PrintPageEventArgs)
        MyBase.OnPrintPage(e)
        ' Draw text to the printer graphics device.
        Dim rect As Rectangle = e.MarginBounds
        e.Graphics.DrawString("Hello, Printer!", m_fnt, Brushes.Black, 0, 0)
        ' Indicate that there are no more pages.
        e.HasMorePages = False
    End Sub
End Class
```

4.7.4 Choosing a Printer

The code given in Examples Example 4-13 and Example 4-14 merely prints to the default printer.To allow the user to select a specific printer and set other printer options, pass the PrintDocument object to a PrintDialog object and call the PrintDialog object's ShowDialog method. The ShowDialog method displays a PrintDialog dialog box (shown in Figure 5-19 in Chapter 5). When the user clicks OK in the PrintDialog dialog box, the ShowDialog method sets the appropriate values in the given PrintDocument object. The PrintDocument object's Print method can then be called to print the document to the selected printer. **Here is the code:**

```
' Create the PrintDocument object and the dialog box object.
Dim pd As New HelloPrintDocument( )
Dim dlg As New PrintDialog( )

' Pass the PrintDocument object to the dialog box object.
dlg.Document = pd

' Show the dialog box. Be sure to test the result so that printing
' occurs only if the user clicks OK.

If dlg.ShowDialog = DialogResult.OK Then
    ' Print the document.
```

```
        pd.Print( )
    End If
```
This code assumes the presence of the HelloPrintDocument class defined in Example 4-13 or Example 4-14. Note that the HelloPrintDocument class itself does not need to be modified to support choosing a printer.

4.7.5 The PageSettings Class

As mentioned earlier, the PrintPageEventArgs object passed to the OnPrintPage method has a
PageSettings property that holds a PageSettings object. This object holds the settings applicable to printing a single page. The properties of the PageSettings class are:

Bounds
Represents a rectangle that specifies the full area of the page, including the area outside the margins. This is the same value found in the PageBounds property of the PrintPageEventArgs class. The syntax of the Bounds property is:

Public ReadOnly Property Bounds( ) As System.Drawing.Rectangle

Color
Indicates whether the page should be printed in color. The syntax of the Color property is:

Public Property Color( ) As Boolean

Landscape
Indicates whether the page is being printed in landscape orientation. The syntax of the Landscape property is:

Public Property Landscape( ) As Boolean

Margins
Indicates the size of the margins. The syntax of the Margins property is:
Public Property Margins( ) As System.Drawing.Printing.Margins
The Margins class has four properties, Left, Top, Right, and Bottom, each of which is an Integer expressing the size of the respective margin.

PaperSize
Indicates the size of the paper. The syntax of the PaperSize property is:
Public Property PaperSize( ) As System.Drawing.Printing.PaperSize
The PaperSize class has four properties:

> ***Width***
> An Integer expressing the width of the paper. This is the same value found in the Width member of the Bounds property of the PageSettings object.
>
> ***Height***
> An Integer expressing the height of the paper. This is the same value found in the Height member of the Bounds property of the PageSettings object.
>
> ***Kind***
> An enumeration of type PaperKind expressing the size of the paper in terms of standard named sizes, such as Letter and Legal.
> *PaperName*
> A string giving the name of the paper size, such as "Letter" and "Legal".
>
> ***PaperName***
> A string giving the name of the paper size, such as `"Letter"` and `"Legal"`.

PaperSource
Indicates the paper tray from which the page will be printed. The syntax of the PaperSource property is:
Public Property PaperSource( ) As System.Drawing.Printing.PaperSource
The PaperSource class has two properties:

Kind
An enumeration of type PaperSourceKind expressing the paper source in terms of standard names, such as Lower and Upper.
SourceName
A string giving the name of the paper source, such as "Lower" and "Upper".

PrinterResolution
Indicates the resolution capability of the printer. The syntax of the PrinterResolution property is:
Public Property PrinterResolution( ) As _System.Drawing.Printing.PrinterResolution
The PrinterResolution class has three properties:

X
An Integer expressing the horizontal resolution of the printer in dots per inch.

Y
An Integer expressing the vertical resolution of the printer in dots per inch.

Kind
An enumeration of type PrinterResolutionKind expressing the resolution mode. The values of this enumeration are Draft, Low, Medium, High, and Custom.

PrinterSettings
Indicates the settings applicable to the printer being used. The syntax of the PrinterSettings property is:

```
Public Property PrinterSettings( ) As _
System.Drawing.Printing.PrinterSettings
```

The PrinterSettings class is described in the next section.

4.7.6 The PrinterSettings Class

The PrinterSettings class holds values that describe the capabilities and settings of a specific printer. It exposes these properties:

CanDuplex
Indicates whether the printer can print on both sides of the paper. The syntax of the CanDuplex property is:

```
Public ReadOnly Property CanDuplex( ) As Boolean
```

Collate
Indicates whether the document being printed will be collated. The syntax of the Collate property is:

```
Public Property Collate( ) As Boolean
```

Copies
Indicates the number of copies to print. The syntax of the Copies property is:

```
Public Property Copies( ) As Short
```

DefaultPageSettings
Indicates the default page settings for this printer. The syntax of the DefaultPageSettings property is:

```
Public ReadOnly Property DefaultPageSettings( ) As _
System.Drawing.Printing.PageSettings
```

The PageSettings class was described in the previous section.

Duplex
Indicates whether the print job is to print on both sides of the paper. The syntax of the Duplex property is:

Public Property Duplex( ) As System.Drawing.Printing.Duplex
The Duplex type is an enumeration with the following values:

Simplex
The document will print only on one side of each page.

Horizontal
The document will print using both sides of each page.

Vertical
The document will print using both sides of each page, and the second side will be inverted to work with vertical binding.

Default
The document will print using the printer's default duplex mode.

FromPage
Specifies the first page to print if the PrintRange property is set to SomePages. The syntax of the FromPage property is:

```
Public Property FromPage( ) As Integer
```

InstalledPrinters
Indicates the names of the printers installed on the computer. This list includes only the printers physically connected to the machine (if any), not necessarily all printers set up in the Control Panel. The syntax of the InstalledPrinters property is:

```
Public Shared ReadOnly Property InstalledPrinters( ) As _
System.Drawing.Printing.PrinterSettings.StringCollection
```

The StringCollection class is a collection of strings. It can be iterated using code such as this:

```
' Assume pts is of type PrinterSettings.
    Dim str As String
        For Each str In pts.InstalledPrinters
            Console.WriteLine(str)
        Next
```

IsDefaultPrinter
Indicates whether this printer is the user's default printer. The syntax of the IsDefaultPrinter property is:

```
Public ReadOnly Property IsDefaultPrinter( ) As Boolean
```

If the PrinterName property is explicitly set to anything other than Nothing, this property always returns False.

IsPlotter
Indicates whether this printer is a plotter. The syntax of the IsPlotter property is:

```
Public ReadOnly Property IsPlotter( ) As Boolean
```

IsValid
Indicates whether the PrinterName property designates a valid printer. The syntax of the IsValid property is:

```
Public ReadOnly Property IsValid( ) As Boolean
```

This property is useful if the PrinterName property is being set explicitly. If the PrinterName is set as a result of allowing the user to select a printer through the PrintDialog dialog box, this property will always be True.

LandscapeAngle
Indicates the angle (in degrees) by which portrait orientation is rotated to produce landscape orientation. The syntax of the LandscapeAngle property is:

```
Public ReadOnly Property LandscapeAngle( ) As Integer
```

This value can only be 90 or 270. If landscape orientation is not supported, this value can only be 0.

MaximumCopies
Indicates the maximum number of copies that the printer can print at one time. The syntax of the MaximumCopies property is:

```
Public ReadOnly Property MaximumCopies( ) As Integer
```

MaximumPage
Indicates the highest page number that can be entered in a PrintDialog dialog box. The syntax of the MaximumPage property is:

```
Public Property MaximumPage( ) As Integer
```

Set this value prior to calling the PrintDialog object's ShowDialog method to prohibit the user from entering a page number that is too high.

MinimumPage
Indicates the lowest page number that can be entered in a PrintDialog dialog box. The syntax of the MinimumPage property is:

```
Public Property MinimumPage( ) As Integer
```

Set this value prior to calling the PrintDialog object's ShowDialog method to prohibit the user from entering a page number that is too low.

PaperSizes
Indicates the paper sizes that are supported by this printer. The syntax of the PaperSizes property is:

```
Public ReadOnly Property PaperSizes( ) As _
System.Drawing.Printing.PrinterSettings+PaperSizeCollection
```

The PaperSizeCollection is a collection of objects of type PaperSize. The PaperSize type was described in the previous section, under the description for the PaperSize property of the PageSettings class. The PaperSizeCollection can be iterated using the following code:

```
' Assume pts is of type PrinterSettings.
    Dim pprSize As PaperSize
        For Each pprSize In pts.PaperSizes
            Console.WriteLine(pprSize.PaperName)
        Next
```

PaperSources
Indicates the paper sources that are available on the printer. The syntax of the PaperSources property is:

```
Public ReadOnly Property PaperSources( ) As _
System.Drawing.Printing.PrinterSettings+PaperSourceCollection
```

The PaperSourceCollection is a collection of objects of type PaperSource. The PaperSource type was described in the previous section, under the description for the PaperSource property of the PageSettings class. The PaperSourceCollection can be iterated using the following code:

```
' Assume pts is of type PrinterSettings.
Dim pprSource As PaperSource

ForEach pprSource In pts.PaperSources
    Console.WriteLine(pprSource.SourceName
Next
```

PrinterName
Indicates the name of the printer. The syntax of the PrinterName property is:
Public Property PrinterName( ) As String Unless a string has been explicitly assigned to the property, its value is Null.
PrinterResolutions

Indicates the resolution supported by this printer. The syntax of the PrinterResolutions property is:

```
Public ReadOnly Property PrinterResolutions( ) As _
System.Drawing.Printing.PrinterSettings.PrinterResolutionCollection
```

The PrinterResolutionCollection is a collection of objects of type PrinterResolution. The PrinterResolution type was described in the previous section, under the description for the PrinterResolution property of the PageSettings class. The PrinterResolutionCollection can be iterated using the following code:

```
' Assume pts is of type PrinterSettings.
    Dim ptrResolution As PrinterResolution
        For Each ptrResolution In pts.PrinterResolutions
            Console.WriteLine(ptrResolution.Kind.ToString( ))
        Next
```

PrintRange
Indicates the range of pages that are to be printed. The syntax of the PrintRange property is:

Public Property PrintRange( ) As System.Drawing.Printing.PrintRange

The PrintRange type is an enumeration having the following values:

AllPages
Prints the entire document.Selection
Prints only the selected portion of the document.
SomePages Prints the pages starting at the page specified in the FromPage property and ending at the page specified in the ToPage property.

SupportsColor
Indicates whether the printer supports color printing. The syntax of the SupportsColor property is:

```
Public ReadOnly Property SupportsColor( ) As Boolean
```

ToPage Specifies the final page to print if the PrintRange property is set to SomePages. The syntax of the ToPage property is:

```
Public Property ToPage( ) As Integer
```

The methods of the PrinterSettings class are:

Clone
Creates a copy of the PrinterSettings object. The syntax of the Clone method is:

```
Public NotOverridable Function Clone( ) As Object_Implements
System.ICloneable.Clone
```

GetHdevmode
Returns a handle to a Windows DEVMODE (device mode) structure corresponding to this PrinterSettings object. The GetHdevmode method has two forms:

```
Public Overloads Function GetHdevmode( ) As System.IntPtr
```

and:

```
Public Overloads Function GetHdevmode( _  ByVal pageSettings As
System.Drawing.Printing.PageSettings_    ) As System.IntPtr
```

GetHdevnames
Returns a handle to a Windows DEVNAMES structure corresponding to this PrinterSettings object. The syntax of the GetHdevnames method is:

```
Public Function GetHdevnames( ) As System.IntPtr
```

SetHdevmode
Sets properties of this PrinterSettings object based on values in the given DEVMODE structure.The syntax of the SetHdevmode method is:

```
Public Sub SetHdevmode(ByVal hdevmode As System.IntPtr)
```

SetHdevnames
Sets properties of this PrinterSettings object based on values in the given DEVNAMES structure. The syntax of the SetHdevnames method is:

```
Public Sub SetHdevnames(ByVal hdevnames As System.IntPtr)
```

4.7.7 Page Setup Dialog Box

Windows Forms provides a common dialog box for page setup Settings entered by the user in this dialog box are saved to a PageSettings object. This PageSettings object can be saved by the application and passed to the PrintDocument object prior to calling the PrintDocument object's Print method. The PrintDocument object will then use the given settings for printing. Here's the code that displays the dialog box:

```
Private Sub ShowPageSetup( )

    ' Display the page settings dialog box. This assumes that there is
    ' a class-level variable of type PageSettings called m_pgSettings.

        Dim pgSetupDlg As New PageSetupDialog( )
        pgSetupDlg.PageSettings = m_pgSettings
        pgSetupDlg.ShowDialog( )

End Sub’
```

This code depends on the existence of a class-level variable of type PageSettings called m_pgSettings. Here is a suitable definition for this variable:

```
' Private member to hold the application's page settings.
' This could be placed in an application's main form or in another
' class that is accessible to the code that will need to print.
Private m_pgSettings As New PageSettings( )
```

Note the use of the *New* keyword to ensure that the PageSettings object is instantiated as soon as the enclosing class is instantiated. All that remains is to hand the PageSettings object to the PrintDocument object when it's time to print. Here is the code:

```
Private Sub PrintTheDocument( )
    ' Create the PrintDocument object.
        Dim pd As New HelloPrintDocument( )

    ' Hand it the PageSettings object.
        pd.DefaultPageSettings = m_pgSettings

    ' Create the dialog box object.
        Dim dlg As New PrintDialog( )

    ' Pass the PrintDocument object to the dialog box object.
        dlg.Document = pd

    ' Show the dialog box. Be sure to test the result so that printing
    ' occurs only if the user clicks OK.
```

```
            If dlg.ShowDialog = DialogResult( ).OK Then
                ' Print the document.
                pd.Print( )

            End If

        End Sub
```

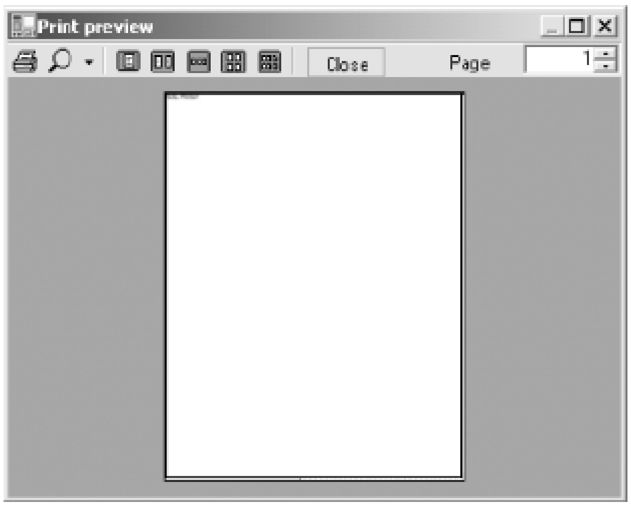
4.7.8 Print Preview

Generating a print preview is trivial. An instance of the PrintDocument object is created and passed to a PrintPreviewDialog object, whose ShowDialog method is then called to show the print preview. Here is the code:

```
    Private Sub ShowPrintPreview( )

        ' Create the PrintDocument object.
            Dim pd As New HelloPrintDocument( )

        ' Hand it the PageSettings object.
            pd.DefaultPageSettings = m_pgSettings

        ' Create the print preview dialog box object.
            Dim dlg As New PrintPreviewDialog( )

        ' Pass the PrintDocument object to the dialog box object.
            dlg.Document = pd

        ' Show the dialog box.
            dlg.ShowDialog( )

    End Sub
```

The result is shown in Figure 4-16.

Figure 4-16. Hello, Printer! in a print preview window



4.7.9 Summary of Printing

The .NET Framework hides the mechanics of printing. Applications don't have to know how to find printers, they don't have to know what a given printer's capabilities are, and they don't have to know how to issue commands that are meaningful to a given brand of printer. The Framework abstracts all of this away. However, the Framework doesn't know anything about a given application's documents. It is up to the application developer to know how to paginate the application's documents and render each page in response to each call to the PrintDocument class's OnPrintPage method.

4.4, 4.5, 4.6, 4.7 Check Your Progress
Fill in the Blanks

1. …………………….…. application permits more than one document to be open at a time.
2. ……….…………… indicates the default value of a property.
3. ………………….. objects hold the settings used when drawing lines.
4. ……………………….. is technique for making the edges of graphics figures.
5. MDI stands for …………………………………

4.8 FILES IN VB.NET

We'll also take a look at general Visual Basic file handling in this chapter. In Visual Basic .NET, file handling is largely based on the System.IO namespace, which encloses a class library that supports string, character, and file manipulation. These classes include properties, methods, and events for creating, copying, moving, and deleting files. The most commonly used classes are FileStream, BinaryReader, BinaryWriter, StreamReader, and StreamWriter, and we'll take a look at all of these classes in this chapter.

4.8.1 FileStream Class

Yyou can use the **FileStream** class to open or create files, and then use other classes, like **BinaryWriter** and **BinaryReader**, to work with the data in the file. Here's the hierarchy of the **FileStream** class:

```
Object
  MarshalByRefObject
    Stream
      FileStream
```

You can find the more notable public properties of **FileStream** objects in Table 13.7 and the more notable public methods in Table given below-

Table: Noteworthy public properties of *FileStream* objects.

Property	Means
CanRead	Determines if the stream supports reading.
CanSeek	Determines if the stream supports seeking.
CanWrite	Determines if the stream supports writing.
Handle	Gets the operating system file handle for the stream's file.
IsAsync	Determines if the stream was opened asynchronously or synchronously.
Length	Gets the length of the stream in bytes.
Name	Gets the name of the file stream passed to the constructor.
Position	Gets/sets the position in this stream.

Table: Noteworthy public methods of *FileStream* objects.

Method	Means
BeginRead	Starts an asynchronous read operation.
BeginWrite	Starts an asynchronous write operation.
Close	Closes a file, making it available in Windows to any other program.
EndRead	Waits for an asynchronous read operation to finish.

Table: Noteworthy public methods of *FileStream* objects.

Method	Means
EndWrite	Ends an asynchronous write operation, waiting until the operation has finished.
Flush	Flushes all buffers for this stream, writing any buffered data out to its target (such as a disk file).
Lock	Withholds any access to the file to other processes.
Read	Reads a block of bytes.
ReadByte	Reads a byte from the file.
Seek	Sets the current read/write position.
SetLength	Sets the length of the stream.
Unlock	Gives access to other processes to a file that had been locked.
Write	Writes a block of bytes to this stream.
WriteByte	Writes a byte to the current read/write position.

4.8.2 FileMode

When you open a file with the **FileStream** class, you specify the file mode you want to use—for example, if you want to create a new file, you use the file mode **FileMode.Creat**e. The various possible file modes are part of the **FileMode** enumeration; you can find the members of this enumeration in Table given below-

Table: Members of the *FileMode* enumeration.

Member	Means
Append	Opens a file and moves to the end of the file (or creates a new file if the specified file doesn't exist). Note that you can only use **FileMode.Append** with **FileAccess.Write**.
Create	Creates a new file; if the file already exists, it is overwritten.
CreateNew	Creates a new file; if the file already exists, an **IOException** is thrown.
Open	Opens an existing file.
OpenOrCreate	Open a file if it exists; or create a new file.
Truncate	Open an existing file, and truncate it to zero length so you can write over its data.

4.8.3 FileAccess

When you open files with the **FileStream** class, you can specify the file mode (see the previous topic) and *access*. The access indicates the way you're going to use the file- to read from, to write to, or both. To indicate the type of file access you want, you use members of the **FileAccess** enumeration. You can find the members of the **FileAccess** enumeration in Table given below-

Table: Members of the *FileAccess* enumeration.

Member	Means
Read	Gives read access to the file, which means you can read data from the file.
ReadWrite	Gives both read and write access to the file, which means you can both read and write to and from a file.

Table: Members of the *FileAccess* enumeration.	
Member	**Means**
Write	Gives write access to the file, which means you can write to the file.

4.8.4 Opening or Creating a File with the FileStream Class

When you want to open or create a file, you use the **FileStream** class, which has many constructors, allowing you to specify the file mode (for example, **FileMode.Create**), file access (such as **FileAccess.Write**), and/or the file-sharing mode (such as **FileShare.None**), like this (these are only a few of the **FileStream** constructors):

```
Dim fs As New System.IO.FileStream(String, FileMode)
Dim fs As New System.IO.FileStream(String, FileMode, FileAccess)
Dim fs As New System.IO.FileStream(String, FileMode, FileAccess, FileShare)
```

The StreamWriterReader example on the CD-ROM shows how this works—in that example, I'm creating a file named file.txt and opening it for writing with a **FileStream** object; note that I'm setting the file mode to **Create** to create this new file, and explicitly setting the file access to **Write** so we can write to the file:

```
Imports System
Imports System.IO

Public Class Form1

    Inherits System.Windows.Forms.Form
    'Windows Form Designer generated code

    Private Sub Button1_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles Button1.Click
        Dim fs As New System.IO.FileStream("file.txt", FileMode.Create, _
            FileAccess.Write)
```

4.8.5 Writing Text with the StreamWriter Class

You can use the **StreamWriter** class to write text to a file. In the topic "Opening or Creating a File with the **FileStream** Class" in this chapter, I used a **FileStream** object to create a file for writing in the StreamWriterReader example on the CD-ROM. In this topic, I can continue that example by using that **FileStream** object to create a **StreamWriter** object and use **StreamWriter** methods to write sample text to the file. Here's how this looks (note that I've discussed these methods in the In Depth section of this chapter; for example, **Write** just writes text, while **WriteLine** follows the text it writes with a carriage-return/linefeed pair):

```
Imports System
Imports System.IO

Public Class Form1
        Inherits System.Windows.Forms.Form

        'Windows Form Designer generated code

        Private Sub Button1_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles Button1.Click
            Dim fs As New System.IO.FileStream("file.txt", FileMode.Create, _
                FileAccess.Write)

            Dim w As New StreamWriter(fs)
            w.BaseStream.Seek(0, SeekOrigin.End)
            w.WriteLine("Here is the file's text.")
        w.Write("Here is more file text." & ControlChars.CrLf)
        w.WriteLine("And that's about it.")
        w.Flush()
        w.Close()

      End Sub
  End Class
```

4.8.6 Reading Text with the StreamReader Class

In the topic "Writing Text with the **StreamWriter** Class" in this chapter, I used the **StreamWriter** class to write text to a file in the StreamWriterReader example on the CD-ROM. You also can use the **StreamReader** class to read that text back in, and I do that in the **StreamWriterReader** example like this, as discussed in the In Depth section of this chapter:

```
  Imports System
  Imports System.IO

  Public Class Form1
      Inherits System.Windows.Forms.Form

      'Windows Form Designer generated code

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

        Dim fs As New System.IO.FileStream("file.txt", FileMode.Create, _
                FileAccess.Write)

        Dim w As New StreamWriter(fs)
            w.BaseStream.Seek(0, SeekOrigin.End)
            w.WriteLine("Here is the file's text.")
            w.Write("Here is more file text." & ControlChars.CrLf)
            w.WriteLine("And that's about it.")
            w.Flush()
            w.Close()
        fs = New System.IO.FileStream("file.txt", FileMode.Open, _
                FileAccess.Read)

        Dim r As New StreamReader(fs)
        r.BaseStream.Seek(0, SeekOrigin.Begin)

        While r.Peek() > -1
           TextBox1.Text &= r.ReadLine() & ControlChars.CrLf
        End While

              r.Close()

      End Sub
  End Class
```

4.8.7 Binary Class

Table: Noteworthy public properties of *BinaryWriter* objects.

Property	Means
BaseStream	Gets the underlying stream, giving you access to that stream's properties and methods.

Table: Noteworthy public methods of *BinaryWriter* objects.

Method	Means
Close	Closes the binary writer as well as the underlying stream.

Table: Noteworthy public methods of *BinaryWriter* objects.

Method	Means
Flush	Flushes the buffer of the binary writer and writes out any buffered data.
Seek	Sets the read/write position in the stream.
Write	Writes data to the stream.

4.8.8 Writing Binary Data with the BinaryWriter Class

Once you have a **FileStream** object, you can use the **BinaryWriter** class to write binary data to a file. The BinaryWriterReader example on the CD-ROM shows how to do this, and this topic is also discussed in the In Depth section of this chapter. That example uses the **BinaryWriter** class's **Write** method to write 20 **Int32** values to a file, data.dat:

```
Imports System.IO

Public Class Form1
    Inherits System.Windows.Forms.Form

        'Windows Form Designer generated code

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

          Dim fs As FileStream = New FileStream("data.dat", _
                   FileMode.OpenOrCreate)

          Dim w As BinaryWriter = New BinaryWriter(fs)

          Dim LoopIndex As Int32
          For LoopIndex = 0 To 19
             w.Write(LoopIndex)
          Next

        End Sub
End Class
```

4.8.9 Binary Reader Class

Table: Noteworthy public properties of *BinaryReader* objects.

Property	Means
BaseStream	Holds the underlying stream of the binary reader, giving you access to that stream's properties and methods.

Table: Noteworthy public methods of *BinaryReader* objects.

Method	Means
Close	Closes the binary reader as well as the underlying stream.
PeekChar	Peeks ahead and returns the next available character (but does not advance the read/write position).
Read	Reads characters from the underlying stream and advances the current position of the stream.
ReadBoolean	Reads a **Boolean** from the stream.
ReadByte	Reads the next byte from the stream.

Table: Noteworthy public methods of *BinaryReader* objects.

Method	**Means**
ReadBytes	Reads a number of bytes from the stream into a byte array.
ReadChar	Reads the next character from the stream.
ReadChars	Reads a number of characters from the stream.
ReadDecimal	Reads a decimal value from the stream.
ReadDouble	Reads an 8-byte floating-point value from the stream.
ReadInt16	Reads a 2-byte signed integer from the stream.
ReadInt32	Reads a 4-byte signed integer from the stream.
ReadInt64	Reads an 8-byte signed integer from the stream.
ReadSByte	Reads a signed byte from the stream.
ReadSingle	Reads a 4-byte floating-point value from the stream.
ReadString	Reads a string from the current stream.
ReadUInt16	Reads a 2-byte unsigned integer from the stream.
ReadUInt32	Reads a 4-byte unsigned integer from the stream.
ReadUInt64	Reads an 8-byte unsigned integer from the stream.

4.8.10 Reading Binary Data with the BinaryReader Class

If you have a **FileStream** object, you can use the **BinaryReader** class to read binary data from files. In the BinaryWriterReader example on the CD-ROM, I first write 20 **Int32** values to a file and then read them back with a **BinaryReader** object like this, using the **ReadInt32** method, and display them in a text box:

```
Imports System.IO

Public Class Form1
        Inherits System.Windows.Forms.Form

        'Windows Form Designer generated code

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

            Dim fs As FileStream = New FileStream("data.dat", _
                FileMode.OpenOrCreate)

            Dim w As BinaryWriter = New BinaryWriter(fs)
            Dim LoopIndex As Int32

            For LoopIndex = 0 To 19
                w.Write(LoopIndex)
            Next

            w.Seek(0, SeekOrigin.Begin)
            Dim r As BinaryReader = New BinaryReader(fs)

            For LoopIndex = 0 To 19
                TextBox1.Text &= r.ReadInt32() & ControlChars.CrLf
            Next
        End Sub
End Class
```

4.9 SUMMARY

In this chapter, you've seen how to work with Windows forms in creating a desktop application. A Windows form, however, is simply a container for the application's user interface components. In the next chapter, you'll round out your examination of desktop application development by looking at controls, common dialogs, and menus.

4.10 CHECK YOUR PROGRESS-*ANSWERS*

4.1 & 4.2 & 4.3
1. False
2. False
3. True
4. True

4.4 & 4.5 & 4.6 & 4.7
1. MDI
2. Default ValueAttribute
3. Pen
4. Antialiasing
5. Multiple Document Interface

4.11 QUESTIONS FOR SELF-STUDY

1. Explain Handling form Events?
2. Explain MDI Applications?
3. Write short note on graphic class?
4. Explain OnBeginPrint & OnEndPrint Methods?
5. Explain PageSetting Class?

4.12 SUGGESTED READINGS

1. Visual Basic .NET Black Book by Steven Holzner

References

Programming Visual Basic .NET by Dave Grundgeiger

◈◈◈

NOTES

NOTES

CHAPTER 5

WINDOWS FORMS II: CONTROLS, COMMON DIALOG BOXES AND MENUS

5.0 OBJECTIVES

After studying The. NET Framework chapter you will able to :

- Describe common controls and components
- State control events,form and control Layout
- Code for Common Dialog Boxes, Menus, Creating a Control in your aplication

5.1 INTRODUCTION

By themselves, one or more forms provide very little functionality to most desktop applications. For the most part, forms are valuable insofar as they serve as containers for controls. In this chapter, we'll complete our discussion of building desktop applications by focusing on the objects that forms contain—in particular, controls and components, common dialogs, and menus.

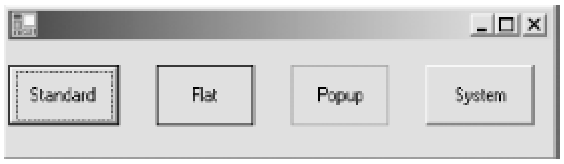
5.2 COMMON CONTROLS AND COMPONENTS

This section contains a summary of the controls and components defined in the System.Windows.Forms namespace. *Components* are classes derived from the Component class (defined in the System.ComponentModel namespace). They may or may not provide a visual interface. They are often used as elements of forms but don't have to be. *Controls* are classes derived from the Control class (defined in the System.Windows.Forms namespace). Controls generally are used to build the visual appearance of a form. The Control class itself is derived from the Component class, so controls are also components.

5.2.1 The Button Class

This class represents a button control, which is one of the most commonly used controls in Windows applications. The Button class's Click event, which it inherits from Control, is its most commonly used event.The Button class inherits two important properties from ButtonBase: FlatStyle and Image. The first determines the appearance of the button and can take any value of the FlatStyle enumeration: Flat, Popup, Standard (the default), and System. Buttons with these four settings are shown in Figure 5-1. Assigning FlatStyle.System as the value of the FlatStyle property makes the appearance of the button dependent on the operating system.

Figure 5-1. Various appearances of the Button control



The Image property allows you to embed an image into a button. The following code shows how to programmatically set the Image property of Button:
Button1.Image = New System.Drawing.Bitmap(filepath)

5.2.2 The CheckBox Class

The CheckBox class represents a checkbox control. Its appearance is determined by its Appearance property, which can take either value of the Appearance enumeration: Button or Normal (the default). The Button value is rarely used because this setting makes the checkbox look like a Button control.
The CheckBox class's Checked property can be set to True to make the checkbox checked or False to uncheck it.

5.2.3 The ComboBox Class

Both the ComboBox and ListBox classes derive from the ListControl class; therefore, the ComboBox class is very similar to the ListBox class and has properties and methods similar to those of the ListBox class. Refer to Section 5.1.9 for information on these common properties and methods.The following properties are specific to the ComboBox class:

DropDownStyle

Defines the drop-down style of the ComboBox. It can take any value of the ComboBoxStyle enumeration: DropDown (the default value), DropDownList, and Simple. Both DropDown and DropDownList require the user to click the arrow button to display the drop-down portion of the ComboBox; however, DropDown allows the user to edit the text portion of the ComboBox. Simple makes a ComboBox's text portion editable and its drop-down portion always visible.

DroppedDown

Specifies whether the drop-down portion of a ComboBox is visible.

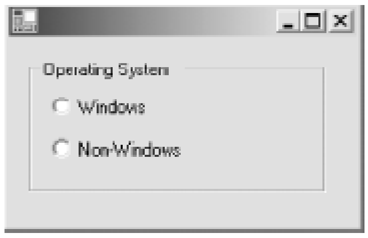
5.2.4 The DateTimePicker Class

The DateTimePicker class represents a control that allows users to select a date in the calendar, just like the MonthCalendar control. Unlike MonthCalendar, however, the DateTimePicker control only displays a box, which looks like a combo box, containing the selected date. When the user clicks the arrow, the control displays a drop-down calendar similar to the MonthCalendar control, from which the user can select a date. This drop-down portion closes as soon as the user selects a date. The user can also click on the day, date, month, or year portion of the control for editing. The DateTimePicker class has MinDate and MaxDate properties that are similar to the ones in the MonthCalendar class. To set the current date or to obtain the selected date, use the Value property of the DateTimePicker class. The selected date is readily available as a DateTime data type.

5.2.5 The GroupBox Class

As the name implies, a GroupBox control is used for grouping other controls, such as radio buttons orcheckboxes; it corresponds to the Frame control in Visual Basic 6.0. A GroupBox grouping two radio buttons is shown in Figure 5-2.

Figure 5-2. A GroupBox grouping two radio buttons



The Controls property of GroupBox represents a Control.ControlCollection class. It has methods such as Add, AddRange, Clear, GetEnumerator, and Remove, which behave exactly as do the same methods in Form.ControlCollection. For example, you can add several controls at once to a GroupBox using its AddRange method, as demonstrated by the following code that adds two radio buttons to a GroupBox named groupBox1:

```
groupBox1.Controls.AddRange(New Control( ) {radioButton1, radioButton2})
```

5.2.6 The ImageList Class

The ImageList class allows you to manage a collection of images. The most important property of this class is Images, which returns an ImageList.ImageCollection object. The ImageList.ImageCollection class has methods to add and remove images from the collection.

The Add method of the ImageList.ImageCollection class adds a bitmap image or an icon to the ImageList's image collection. The Add method has three overloads, whose signatures are given as follows:

```
Overloads Public Sub Add( ByVal value As System.Drawing.Icon)
Overloads Public Sub Add( ByVal value As System.Drawing.Image)
Overloads Public Sub Add( ByVal value As System.Drawing.Image, _
                    ByVal transparentColor as System.Drawing.Color)
```

The first overload allows you to add an icon, and the second overload is used to add an object of type System.Drawing.Image. The third overload is used to add an image and make a color of that image transparent. For example, if you have an image with a blue background color, you can make the added image transparent by passing a blue color as the second argument to the third overload of the Add method.The RemoveAt method of the ImageList.ImageCollection class allows you to remove an image.Once you instantiate an ImageList object, you can start adding images or icons. The following code,for example, adds three images and icons using three different overloads of the Add method:

```
Imports System.Drawing
Dim imageList1 As ImageList1 = New ImageList( )
    ImageList1.Images.Add(New Icon("C:\Palm.ico"))
    ImageList1.Images.Add(New Bitmap("C:\TV.bmp"))
    ImageList1.Images.Add(New Bitmap("C:\Dog.bmp"), Color.White)
```

Important properties of the ImageList class include ColorDepth and ImageSize. The ColorDepth property determines the number of colors available for the ImageList. For example, a value of 4 means that 2^4 = 16 colors are available.

The ImageSize property determines the sizes of all images in the list. By default, the value is System.Drawing.Size(16, 16).

You can assign an ImageList object to the ImageList property of controls such as Label, Button, and ListView. You can then select an image from the ImageList to be displayed on the control using the control's ImageIndex property.

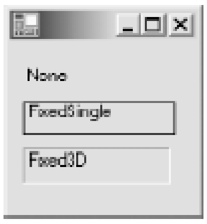
For example, the following code uses an ImageList for a button and selects the first image as the background image for the button:

```
button1.ImageList = imageList1
button1.ImageIndex = 0
```

5.2.7 The Label Class

This class represents a Label control. Its appearance is determined by two properties: BorderStyle and FlatStyle. The BorderStyle property defines the appearance of the control's border and takes any of the three members of the BorderStyle enumeration: None (the default), FixedSingle, and Fixed3D. Figure 5-3 shows three labels with three different values of BorderStyle.

Figure 5-3. Three labels with different BorderStyle values



The FlatStyle property defines the appearance of the control and can take as its value any member of the FlatStyle enumeration: Flat, Popup, Standard, and System. However, if the value of the BorderStyle property is set to None, the label's FlatStyle property can take no other value than FlatStyle.Standard. For more information on the FlatStyle property,.You normally assign a String to the label's Text property. However, you can also assign an image to its Image property. For example, the following code programmatically sets the Image property of a label:

```
Label1.Image = New System.Drawing.Bitmap(filepath)
```

Another important property is TextAlign, which determines how the label's text is aligned. This property can take any member of the ContentAlignment enumeration, including BottomCenter, BottomLeft,BottomRight, MiddleCenter, MiddleLeft, MiddleRight, TopCenter, TopLeft (the default value), and TopRight. The UseMnemonic property can be set to True so that the label accepts an ampersand character in the Text property as an access-key prefix character.

5.2.8 The LinkLabel Class

The LinkLabel class represents a label that can function as a hyperlink, which is a URL to a web site. Its two most important properties are Text and Links. The Text property is a String that defines the label of the LinkLabel object. You can specify that some or all of the Text property value is a hyperlink. For example, if the Text property has the value "Click here for more details", you can make the whole text a hyperlink, or you can make part of it (e.g., the word "here") a hyperlink. How to do this will become clear after the second property is explained. For a LinkLabel to be useful, it must contain at least one hyperlink. The Links property represents a LinkLabel.LinkCollection class of the LinkLabel object. You use the Add method of the LinkLabel.LinkCollection class to add a LinkLabel.Link object. The Add method has two overloads; the one that will be used here has the following signature:

```
Overloads Public Function Add( _ByVal start As Integer, _ByVal length As Integer, _
                                                    ByVal linkData As Object _) As Link
```

The start argument is the first character of the Text property's substring that you will turn into a hyperlink. The length argument denotes the length of the substring. The linkData argument is normally a String containing a URL, such as "www.oreilly.com". For example, if your Text property contains "Go to our web site", and you want "web site" to be the hyperlink, here is how you would use the Add method:

```
linkLabel1.Links.Add(10, 8, "www.oreilly.com")
```

10 is the position of the character w in the Text property, and **8** is the length of the substring "web site". The LinkLabel class has a LinkClicked event that you can capture so that you can run code when a LinkLabel object is clicked. The following example creates a LinkLabel object that is linked to the URL and starts and directs the default browser to that URL when the LinkLabel is clicked:

```
Dim WithEvents linkLabel1 As LinkLabel = new LinkLabel( )
    linkLabel1.Text = "Go to our web site"
    linkLabel1.Links.Add(10, 8, "www.oreilly.com")
    linkLabel1.Location = New System.Drawing.Point(64, 176)
    linkLabel1.Name = "LinkLabel1"
    linkLabel1.Size = New System.Drawing.Size(120, 16)
    ' Add to a form.
    Me.Controls.Add(linkLabel1)
Private Sub LinkLabel1_LinkClicked( _ByVal sender As Object, _
    ByVal e As LinkLabelLinkClickedEventArgs)_Handles linkLabel1.LinkClicked
    'Start the default browser and direct it to "Typed URL".
    System.Diagnostics.Process.Start(e.Link.LinkData.ToString( ))
End Sub
```

The LinkLabel class has a number of properties that are related to the appearance of a LinkLabelobject:

ActiveLinkColor

Represents the color of the LinkLabel when it is being clicked (i.e., when you press your mouse button but before you release it).
By default, the value of **ActiveLinkColor is**
System.Drawing.Color.Red.

DisabledLinkColor

Represents the color of the LinkLabel when it is disabled.

LinkColor

Represents the color of the LinkLabel in its normal condition. By default, the value is

System.Drawing.Color.Blue.

VisitedLinkColor

Represents the color of a LinkLabel that has been visited. By default, this property value is System.Drawing.Color.Purple. The LinkLabel does not automatically display in its VisitedLinkColor after it is clicked. You must change its LinkVisited property to True. Normally, you do this inside the LinkClicked event handler of the LinkLabel object. Therefore, in the previous example, if you want the LinkLabel to change color after it is clicked, you can modify its LinkClicked event handler with the following:

```
Private Sub LinkLabel1_LinkClicked( _ByVal sender As Object, _
        ByVal e As LinkLabelLinkClickedEventArgs)_Handles
                                        linkLabel1.LinkClicked
    LinkLabel1.LinkVisited = True
    ' Start the default browser and direct it to "www.oreilly.com".

    System.Diagnostics.Process.Start(e.Link.LinkData.ToString( ))
End Sub
```

LinkBehavior

Determines how the LinkLabel is displayed. This property can take any member of the LinkBehavior enumeration: AlwaysUnderline, HoverUnderline, NeverUnderline, and SystemDefault (the default value).

5.2.9 The ListBox Class

The ListBox class represents a box that contains a list of items. The following are its more important properties:

MultiColumn

This is a Boolean that indicates whether the listbox has more than one column. Its default value is False.

ColumnWidth

In a multicolumn listbox, this property represents the width of each column in pixels. By default, the value of this property is zero, which makes each column have a default width.

Items

This is the most important property of the ListBox class. It returns the ListBox.ObjectCollection class, which is basically the Items collection in the ListBox. You can programmatically add an item using the Add method or add a range of items using the AddRange method of the ListBox.ObjectCollection class. For example, the following code adds the names of vegetables and fruits to a ListBox object named listBox1:

```
listBox1.Items.AddRange(New Object( )
    {
    "apple", "avocado", "banana", "carrot", "mandarin", "orange"
    }
)
```

For more information about the ListBox.ObjectCollection class, This property determines whether multi-item selection is possible in a ListBox object. It can be assigned any member of the SelectionMode enumeration:

None, One (the default value),

MultiSimple, and MultiExtended. Both MultiSimple and MultiExtended allow the user to select more than one item. However, MultiExtended allows the use of the Shift, Ctrl, and arrow keys to make a selection.

SelectedIndex

This is the index of the selected item. The index is zero-based. If more than one item is selected, this property represents the lowest index. If no item is selected, the property returns -1.

SelectedIndices

This read-only property returns the indices to all items selected in a ListBox object in the form of a ListBox.SelectedIndexCollection object. The ListBox.SelectedIndexCollection class has a Count property that returns the number of selected indices and an Item property that returns the index number. For example, the following code returns the index number of all selected items in a ListBox control named listBox1:

```
Dim selectedIndices As ListBox.SelectedIndexCollection
    ' Obtain the selected indices.
    selectedIndices = listBox1.SelectedIndices

    ' Get the number of indices.
Dim count As Integer = selectedIndices.Count

Dim i As Integer
    For i = 0 To count - 1
        Console.WriteLine(selectedIndices(i))
    Next
```

SelectedItem

This read-only property returns the selected item as an object of type Object. You must cast the returned value to an appropriate type, which is normally String. If more than one item is selected, the property returns the item with the lowest index.

SelectedItems

This read-only property returns all items selected in a ListBox object in the form of a ListBox.SelectedObjectCollection object. The ListBox.SelectedObjectCollection class has a Count property that returns the number of items in the collection and an Item property that you can use to obtain the selected item. For example, the following code displays all the selected items of a ListBox control called listBox1:

```
Dim selectedItems As ListBox.SelectedObjectCollection
    selectedItems = listBox1.SelectedItems

Dim count As Integer = selectedItems.Count
Dim i As Integer

    For i = 0 To count - 1
        Console.WriteLine(selectedItems(i))
    Next
```

Sorted

A value of True means that the items are sorted. Otherwise, the items are not sorted. By default, the value of this property is False.

Text

This is the currently selected item's text.

TopIndex

This is the index of the first visible item in the ListBox. The value changes as the user scrolls through the items.

5.2.10 The ListBox.ObjectCollection Class

This class represents all the items in a ListBox object. It has a Count property that returns the number of items in the ListBox and an Item property that returns the item object in a certain index position. The following sample code reiterates all the items in a ListBox control named listBox1:

```
Dim items As ListBox.ObjectCollection
    items = ListBox1.Items
Dim count As Integer = items.Count
Dim i As Integer
    For i = 0 To count - 1
        Console.WriteLine(items(i))
    Next
```

In addition, the ListBox.ObjectCollection class has the following methods:

Add
Adds an item to the ListBox object. Its syntax is:

ListBox.ObjectCollection.Add(*item*)

where *item* is data of type Object that is to be added to the collection. The method returns the zero-based index of the new item in the collection.

AddRange
Adds one or more items to the ListBox object. Its most common syntax is:

ListBox.ObjectCollection.AddRange(*items( )*)

where *items* is an array of objects containing the data to be added to the ListBox.

Clear
Clears the ListBox, removing all the items. Its syntax is:

ListBox.ObjectCollection.Clear( )

Contains
Checks whether an item can be found in the list of items. Its syntax is:

ListBox.ObjectCollection.Contains(*value*)

where *value* is an Object containing the value to locate in the ListBox. The method returns True if *value* is found; otherwise, it returns False.

CopyTo
Copies all items to an object array. Its syntax is:

ListBox.ObjectCollection.CopyTo(*dest*( ), *arrayIndex*)

where *dest* is the Object array to which the ListBox items are to be copied, and *arrayIndex* is the starting position within *dest* at which copying is to begin.

IndexOf
Returns the index of a particular item. Its syntax is:

ListBox.ObjectCollection.IndexOf(*value*)

where *value* is an Object representing the item to locate in the collection. The method returns the item's index. If the item cannot be found, the method returns -1.

Insert
Inserts an item into the ListBox at the specified index position. Its syntax is:

ListBox.ObjectCollection.Insert(*index, item*)

where *index* is the zero-based ordinal position at which the item is to be inserted, and *item* is an Object containing the data to be inserted into the collection.

Remove
Removes the item that is passed as an argument to this method from the ListBox. Its syntax is:

ListBox.ObjectCollection.Remove(*value*)

where *value* is an Object representing the item to remove from the collection.

RemoveAt
Removes an item at the specified index position. Its syntax is:

ListBox.ObjectCollection.RemoveAt(*index*)

where *index* is the zero-based ordinal position in the collection of the item to be removed.

5.2.11 The ListView Class

A ListView is a container control that can hold a collection of items. Each item in a ListView can have descriptive text and an image, and the items can be viewed in four modes. The righthand pane of Windows Explorer is a ListView control. An item in a ListView is represented by an object of type ListViewItem. The various constructors of the ListViewItem class permit a ListViewItem to be constructed with a String or with a String and an index number. If an index number is used, it represents the index of the item's image in the ImageList referenced by the ListView. The following code constructs two ListViewItem objects. The first has the text "Item1" and uses the first image in the ImageList. The second has the text "Item2" and uses the second image of the ImageList:

```
    Dim listViewItem1 As ListViewItem = New ListViewItem("Item1", 0)
    Dim listViewItem2 As ListViewItem = New ListViewItem("Item2", 1)
```

Once you have references to one or more ListViewItem objects, you can add the items to your ListView object. To add an item or a group of items, you first need to reference the ListView.ListViewItemCollection collection of the ListView object. This collection can easily be referenced using the Items property of the ListView class. The ListView.ListViewItemCollection has Add and AddRange methods that you can use to add one item or a group of items. For instance, the following code uses the AddRange method to add two ListViewItem objects to a ListView object:

```
listView1.Items.AddRange(New ListViewItem( ) _{listViewItem1, listViewItem2})
```

The Add method of the ListView.ListViewItemCollection has three overloads, two of which allow you add to a ListViewItem without first creating a ListViewItem object. To add a ListViewItem object to the collection, you can use the following overload of the Add method:

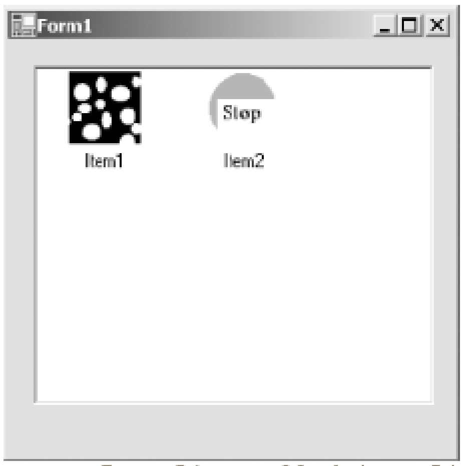
```
    Overridable Overloads Public Function Add _(ByVal value As ListViewItem) _As
        ListViewItem
```

Or,to add a String and convert it into a ListViewItem object, use the following overload:

```
    Overridable Overloads Public Function Add _(ByVal text As String) _
        As ListViewItem
```

Alternatively, you can pass a String and an image index to the third overload:Overridable Overloads Public Function Add _(ByVal text As String, _ByVal imageIndex As Integer) _As ListViewItem The following code demonstrates how to add two ListViewItem objects to a ListView. The ListView is linked to an ImageList that has two images in its collection. When the code is run, it produces something similar to Figure 5-4.

Figure 5-4. A ListView control with two ListViewItem objects



```
    ' Declare and instantiate an ImageList called imageList1.
    Dim imageList1 As ImageList = New ImageList( )
    ' Set the ColorDepth and ImageSize properties of imageList1

    imageList1.ColorDepth = ColorDepth.Depth8Bit
    imageList1.ImageSize = New System.Drawing.Size(48, 48)

    ' Add two images to imageList1.
    imageList1.Images.Add(New Icon("c:\Spotty.ico"))

        imageList1.Images.Add(New Bitmap("c:\StopSign.bmp"))

        ' Declare and instantiate two ListViewItem objects named
        ' listViewItem1 and listViewItem2.
        ' The text for listItem1 is "Item1", and the image is the first
```

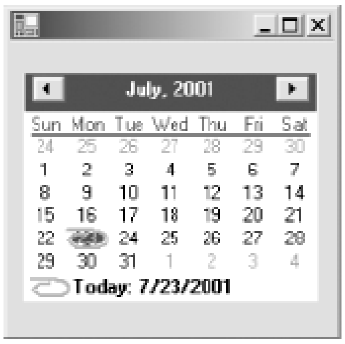
```
' image in the imageList1. ' The text for listItem1 is "Item2", and the image is the
second ' image in the imageList1.
Dim listViewItem1 As ListViewItem = New ListViewItem("Item1", 0)
Dim listViewItem2 As ListViewItem = New ListViewItem("Item2", 1)
' Declare and instantiate a ListView called listView1.
Dim listView1 As ListView = New ListView( )
' Set its properties.
listView1.View = View.LargeIcon
listView1.LargeImageList = imageList1
listView1.Location = New System.Drawing.Point(16, 16)
listView1.Name = "ListView1"
listView1.Size = New System.Drawing.Size(264, 224)
listView1.SmallImageList = Me.ImageList1
' Add listViewItem1 and listViewItem2.
listView1.Items.AddRange(New ListViewItem( ) _
{listViewItem1, listViewItem2})
' Add listView1 to the form.
Me.Controls.AddRange(New Control( ) {listView1})
```

Two properties of the ListView class tell you which item(s) are selected: SelectedIndices and SelectedItems. The first returns a ListView.SelectedIndexCollection object, and the second returns a ListView.SelectedListViewItemCollectionobject.
The ListView.SelectedIndexCollection class has a Count property that tells you how many items are selected and an Item property that returns the index of the designated item. For example, you can retrieve the index of the first selected item by passing 0 to the Item property, as follows:Index = ListView1.SelectedIndices.Item(0)
The ListView.SelectedListViewItemCollection class is very similar to listView.SelectedIndexCollection.Its Count property indicates how many items are selected. However, its Item property returns the item itself, not an index number.

5.2.12 The MonthCalendar Class

The MonthCalendar class represents a control that displays days of a month. A MonthCalendar control is shown in Figure 5-5. By default, when first displayed, the control displays the current month on the user's computer system. Users can select a day by clicking on it or select a range of dates by holding the Shift key while clicking the date at the end of the desired range. Users can also scroll backward and forward to previous or upcoming months, or they can click on the month part and more quickly select one of the 12 months. To change the year, users can click on the year part and click the scrollbar that appears.

Figure 5-5. A MonthCalendar control



Two properties determine the date range that users can select:

MinDate and MaxDate. The MinDate property is a DateTime value representing the minimum date permitted; its default is January 1, 1753.

The MaxDate property determines the maximum date allowed. By default, the value of MaxDate is December 31, 9998.

If you want your MonthCalendar to display a certain range of dates, you need to change these two properties. For instance, the following code allows the user to select a date between January 1, 1980 and December 14, 2010:

```
MonthCalendar1.MinDate = New DateTime(1980, 1, 1)
MonthCalendar1.MaxDate = New DateTime(2010, 12, 14)
```

The MonthCalendar class has a TodayDate property that represents today's date. The user selecting a new date does not automatically change the value of TodayDate. If you want the date selected by the user to be reflected as today's date, you can use the Date_Changed event handler to change its value explicitly, as shown in the following code:

```
Private Sub MonthCalendar1_DateChanged( _
    ByVal sender As System.Object, _
    ByVal e As DateRangeEventArgs) _
    Handles MonthCalendar1.DateChanged
    MonthCalendar1.TodayDate = e.Start
End Sub
```

A DateRangeEventArgs object is passed as an argument to the DateChanged event handler. Its members include a Start property, which represents the beginning of the range of selected dates, and an End property, which represents the end of the range of selected dates. The previous code simply assigns the value of the Start property to TodayDate. Later, if you need to know the value of the userselected date, you can query the TodayDate property. Note that the MonthCalendar control has a fixed size. It will ignore any attempt to change its Size property. If you need more flexibility in terms of the space it occupies, use a DateTimePicker control.

5.2.13 The Panel Class

A panel is a container that can hold other controls. Panels are typically used to group related controls in a form. Like the PictureBox class, the Panel class has a BorderStyle property that defines the panel's border and can take as its value any member of the BorderStyle enumeration: None (the default value), FixedSingle, and Fixed3D. You can add controls to a Panel object using the Add method or the AddRange method of the Control.ControlCollection class. The following code adds a button and a text box to a Panel control called panel1:

```
Dim panel1 As Panel = New Panel( )
Dim textBox1 As TextBox = New TextBox( )
Dim WithEvents button1 As Button
button1 = New Button( )
' button1
button1.Location = New System.Drawing.Point(104, 72)
button1.Name = "button1"
button1.Size = New System.Drawing.Size(64, 48)
button1.TabIndex = 0
button1.Text = "Button1"
' textBox1
textBox1.Location = New System.Drawing.Point(128, 48)
textBox1.Name = "textBox1"
textBox1.TabIndex = 1
panel1.Controls.AddRange(New Control( ) {textBox1, button1})
panel1.Location = New System.Drawing.Point(24, 24)
panel1.Name = "Panel1"
panel1.Size = New System.Drawing.Size(336, 216)
Me.Controls.Add(panel1)
```

5.2.14 The PictureBox Class

The PictureBox class represents a control to display an image. Loading an image into this control is achieved by assigning a System.Drawing.Bitmap object to its Image property, as the following code does:

```
Dim pictureBox1 As PictureBox = New PictureBox( )
pictureBox1.Image = New System.Drawing.Bitmap("c:\tv.bmp")
```

```
pictureBox1.Location = New System.Drawing.Point(72, 64)
pictureBox1.Size = New System.Drawing.Size(144, 128)
Me.Controls.Add(pictureBox1)
```

In addition, the PictureBox class has the BorderStyle and SizeMode properties. The BorderStyle property determines the PictureBox object's border and can take as its value any member of the BorderStyle enumeration: None (the default value), FixedSingle, and Fixed3D.The SizeMode property determines how the image assigned to the Image property is displayed. The SizeMode property can take any of the members of the PictureBoxSizeMode enumeration: AutoSize,
CenterImage, Normal (the default value), and StretchImage.

5.2.15 The RadioButton Class

The RadioButton class represents a radio button. When you add more than one radio button to a form, those radio buttons automatically become one group, and you can select only one button at a time. If you want to have multiple groups of radio buttons on a form, you need to use a GroupBox or Panel control to add radio buttons in the same group to a single GroupBox or Panel.
The following code shows how you can add two radio buttons to a GroupBox and then add the GroupBox to a form. Notice that you don't need to add each individual radio button to a form:

```
' Declare and instantiate a GroupBox and two radio buttons.
Dim groupBox1 As GroupBox = New GroupBox( )
Dim radioButton1 As RadioButton = new RadioButton( )
Dim radioButton2 As RadioButton = new RadioButton( )
' Set the Size and Location of each control.
groupBox1.Size = New System.Drawing.Size(248, 88)
groupBox1.Location = New System.Drawing.Point(112, 168)
radioButton1.Location = New System.Drawing.Point(16, 10)
radioButton1.Size = New System.Drawing.Size(104, 20)
radioButton2.Location = New System.Drawing.Point(16, 50)
radioButton2.Size = New System.Drawing.Size(104, 20)
' Add radioButton1 and radioButton2 to the GroupBox.
groupBox1.Controls.AddRange(New Control( ) {radioButton1, radioButton2})
' Add the GroupBox to the form.
Me.Controls.Add(groupBox1)
```

Like a checkbox, the appearance of a radio button is determined by its Appearance property, which can take one of two members of the Appearance enumeration: Normal (the default) and Button. You don't normally use Appearance.Button because it will make your radio button look like a button. The CheckAlign property determines the text alignment of the radio button. Its value is one of the members of the ContentAlignment enumeration: BottomCenter, BottomLeft, BottomRight, MiddleCenter, MiddleLeft (the default), MiddleRight, TopCenter, TopLeft, and TopRight. The Checked property takes a Boolean. Setting this property to True selects the radio button (and deselects others in the group); setting it to False unselects it.

5.2.16 The TextBox Class

This class represents a text-box control, a control that can accept text as user input. Its majorproperties are:

Multiline
This property can be set to True (indicating that multiple lines are permitted) or False (single-line mode). By default, the value for this property is False.

AcceptsReturn
If True (its default value), and if the Multiline property is also True, pressing the Enter key will move to the next line in the text box. Otherwise, pressing Enter will have the same effect as clicking the form's default button.

CharacterCasing
This property determines how characters that are input by the user appear in the text box. It can take any member of the CharacterCasing enumeration: Normal (the

default), Upper, and Lower. Setting this property to CharacterCasing.Upper translates all input characters to uppercase. Assigning CharacterCasing.Lower to this property converts all input to lowercase. CharacterCasing.Normal means that no conversion is performed on the input.

PasswordChar
This property defines a mask character to be displayed in place of each character input by the user, thereby turning the text box into a password box. This property applies only to a singleline text box (i.e., a text box whose Multiline property is False). It affects the display, but not the value of the Text property.

ScrollBars
In a multiline text box, this property determines whether scrollbars are shown on the control. The property can take any member of the ScrollBars enumeration: None (the default),Horizontal, Vertical, and Both.

TextAlign
This property determines how text is aligned in the text box. It can take any member of the HorizontalAlignment enumeration: Center, Left, and Right.

5.2.17 The Timer Class

The Timer class represents a timer that can trigger an event at a specified interval. At each interval, a Timer object raises its Tick event. You can write code in the Tick event handler that runs regularly. The Timer will raise its Tick event only after you call its Start method. To stop the timer, call its Stop method. The interval for the Timer is set by assigning a value to its Interval property. It accepts a number representing the interval in milliseconds. The following code, which prints an incremented integer from 1 to 20 to the console, shows how to use a timer:

```
Dim i As Integer
Friend WithEvents timer1 As Timer
timer1 = New Timer( )
timer1.Interval = 1000
timer1.Start( )
    ' The event handler for Tick.
    Private Sub Timer1_Tick( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles Timer1.Tick
    If i < 20 Then
        i = i + 1
        Console.WriteLine(i)
    Else
        timer1.Stop( )
    End If
End Sub
```

5.2.18 Other Controls and Components

In addition to the controls we've discussed in some detail in the preceding sections, the .NET Framework includes a number of other controls and components for Windows application development. These include:

AxHost
Wraps ActiveX controls to let them be used within Windows Forms.

CheckedListBox
The same as the ListBox control, except that checkboxes are displayed to the left of each item.The CheckedListBox control derives from the ListBox control.

ContainerControl
A container for other controls. The emphasis of this control is on managing the focus state of contained controls. For example, it has a method that can be called to force activation of a given contained control (ActivateControl) and an overridable

method that is invoked when the user tabs between contained controls (ProcessTabKey). The ContainerControl control provides automatic support for scrolling because it derives from ScrollableControl. When contained controls are anchored or docked, they anchor or dock to the edges of the containing control.

ContextMenu
Provides a method for displaying a context menu (also known as a pop-up menu). Context menus are usually displayed in response to the user right-clicking the mouse. See Section 5.5 later in this chapter for details.

DataGrid
A grid control for displaying ADO.NET data. See Chapter 8 for examples.

DomainUpDown
A Windows up-down control (also known as a spin button) that allows the user to select from a list of text values by clicking the control's up and down arrows. See also NumericUpDown, later in this list.

HScrollBar
A standard Windows horizontal scrollbar.

MainMenu
Provides a method for displaying an application's main menu. See Section 5.5 later in this chapter for details.

MenuItem
Represents a menu item within a menu. See Section 5.5 later in this chapter for details.

NotifyIcon
Provides a way to put an icon into the Windows System Tray.

NumericUpDown
A Windows up-down control (also known as a spin button) that allows the user to select from a list of numeric values by clicking the control's up and down arrows. See also DomainUpDown, earlier in this list.

PrintPreviewControl
Displays a preview image of a document to be printed. The PrintPreviewControl is not usually used directly. It is found on the PrintPreviewDialog.

ProgressBar
A standard Windows progress bar.

PropertyGrid
A user interface for viewing and setting the properties of any object.

RichTextBox
A standard Windows rich text box (also known as a rich edit control). The RichTextBox control can manipulate text in Rich Text Format (RTF).

ScrollableControl
Not used directly. The ScrollableControl control serves as the base class for controls that need to provide automatic support for scrolling. The ContainerControl control and the Panel control are both derived from ScrollableControl.

Splitter
Provides the user with a way to resize docked controls using the mouse.

StatusBar
A standard Windows status bar.

TabControl
A container that provides pages to contain other controls and tabs to click to move between the pages. The pages are instances of the TabPage control, which is derived from the Panel control.

ToolBar
A standard Windows toolbar.

TrackBar
A standard Windows trackbar (also known as a slider)

TreeView
A standard Windows tree view.

UserControl
Not used directly. UserControl serves as the base class for developer-created container controls.
.
VScrollBar
A standard Windows vertical scrollbar.

5.3 CONTROL EVENTS

Controls on a form are represented in code as fields—one field for each control. For example, when the Visual Studio .NET Windows Forms Designer is used to add a text box to a form, the following declaration is added to the form class:

```
    Private WithEvents TextBox1 As System.Windows.Forms.TextBox
```

This declaration doesn't instantiate the control; it only defines a field that can hold a reference to a control of type TextBox. The control is instantiated in the InitializeComponent subroutine, which is called in the Form class's constructor. The code that instantiates the control looks like this:

```
Me.TextBox1 = New System.Windows.Forms.TextBox( )
```

When a field declaration includes the WithEvents keyword, the parent class can handle events that the referenced object raises. To do so, the parent class must define a handler method having the appropriate signature, and the definition of the method must include a Handles clause to link the method to the appropriate event on the appropriate object. For example, here is the definition of a handler method for the Click event of TextBox1:

```
    Private Sub TextBox1_Click( _ByVal sender As Object, _
        ByVal e As System.EventArgs _) Handles TextBox1.Click
        ' ...
    End Sub
```

The event-handler method can be given any name, but it is a common convention to use a name of the form *FieldName_EventName*. The event-handler method's signature must correspond to The signature of the event being handled. By convention, event signatures have two parameters:

Sender and *e*.

The *sender* parameter is always of type Object and holds a reference to the object that raised the event. The *e* parameter is of type EventArgs—or of a type that inherits from EventArgs—and holds a reference to an object that provides any extra information needed for the event. Events that pass a generic EventArgs argument have no event information to pass. Events that pass an argument of an EventArgs-derived type pass additional information within the fields of the passed object. The correct signature for handling a specific event can be determined either by referring to the control's documentation or by using Visual Studio .NET's built-in object browser. In addition, the Visual Studio .NET Windows Forms Designer can automatically generate a handler-method declaration for any event exposed by any control on a given form.

5.4 FORM AND CONTROL LAYOUT

Windows Forms allows developers to lay out sophisticated user interfaces that are capable of intelligently resizing without writing a line of code. Previously, developers writing desktop applications for Windows typically spent a good deal of time writing resizing code to handle the placement of controls on the form when the user resized the form. The .NET platform, however, allows you to define a control's layout and size by setting a few properties.

5.4.1 The Anchor Property

The Anchor property lets a control anchor to any or all sides of its container. Each anchored side of the control is kept within a constant distance from the corresponding side of its container. When the container is resized, its anchored controls are repositioned and resized as necessary to enforce this rule. The Anchor property is defined by the Control class (in the System.Windows.Forms namespace) and so is inherited by all controls and forms. Its syntax is:

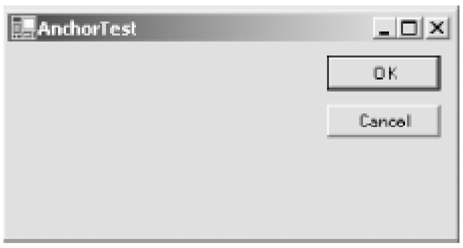
```
Public Overridable Property Anchor( ) As System.Windows.Forms.AnchorStyles
```

The AnchorStyles type is an enumeration that defines the values Left, Top, Right, Bottom, and None. To anchor a control on more than one edge, combine the values with the Or operator, as shown here:

```
' Assumes Imports System.Windows.Forms
SomeControl.Anchor = AnchorStyles.Top Or AnchorStyles.Right
```

By default, controls are anchored on the top and left sides. This means that if a form is resized, its controls maintain a constant distance from the top and left edges of the form. This behavior matches the behavior of Visual Basic 6 forms. For example, Figure 5-6 shows a common button configuration, where the OK and Cancel buttons should track the right edge of the form as the form is resized. In previous versions of Visual Basic, it was necessary to add code to the form's Resize event handler to reposition the buttons as the form was resized. In Visual Basic .NET, however, it is necessary only to set the button's Anchor property appropriately. This can be done either in Visual Studio .NET's Properties window or in code (in the form's constructor).

Figure 5-6. A sizable form with controls that should be anchored on the top and right edges

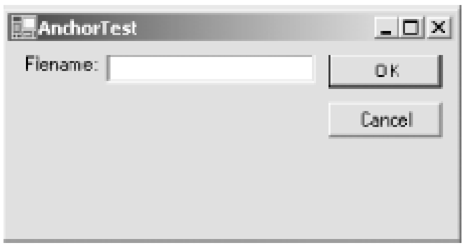


The code to anchor a button on the top and right edges looks like this:

```
' Assumes Imports System.Windows.Forms
btnOk.Anchor = AnchorStyles.Top Or AnchorStyles.Right
```

Sometimes a control should stretch as the form is resized. This is accomplished by anchoring the control to two opposite sides of the form. For example, the text box in Figure 5-7 should always fill the space between the left edge of the form and the OK button on the right side of the form.

Figure 5-7. In this form, the text box should expand and shrink to fill the available space



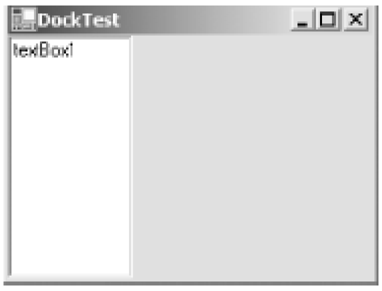
To accomplish this, lay out the form as shown, anchor the buttons on the top and right edges as already discussed, then anchor the text box on the top, left, and right edges. By default, the label in the form is already anchored on the top and left edges. The code looks like this:

```
' Assumes Imports System.Windows.Forms ' Anchor the OK and Cancel buttons on
the top and right edges.

btnOk.Anchor = AnchorStyles.Top Or AnchorStyles.Right
btnCancel.Anchor = AnchorStyles.Top Or AnchorStyles.Right
' Anchor the Filename text box on the top, left, and right edges.
' This causes the text box to resize itself as needed.

txtFilename.Anchor = AnchorStyles.Top Or AnchorStyles.Left _
Or AnchorStyles.Right
```

5.4.2 The Dock Property

The Dock property lets a control be docked to any one side of its container or lets it fill the container. Docking a control to one side of a container resembles laying out the control with three of its edges adjacent and anchored to the three corresponding edges of its container. Figure 5-8 shows a text box control docked to the left side of its form.

Figure 5-8. A TreeView docked to the left side of a form

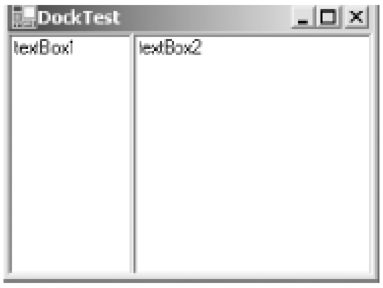


The Dock property is defined by the Control class (in the System.Windows.Forms namespace) and so is inherited by all controls and forms. Its syntax is:

```
Public Overridable Property Dock( ) As System.Windows.Forms.DockStyle
```

The DockStyle type is an enumeration that defines the values Left, Top, Right, Bottom, Fill, and None. Only one value can be used at a time. For example, a control can't be docked to both the left and right edges of its container. Setting the Dock property to DockStyle.Fill causes the control to expand to fill the space available in its container. If the control is the only docked control in the container, it expands to fill the container. If there are other docked controls in the container, the control expands to fill the space not occupied by the other docked controls. For example, in Figure 5-9, the Dock property of textBox1 is set to DockStyle.Left, and the Dock property of textBox2 is set to DockStyle.Fill.

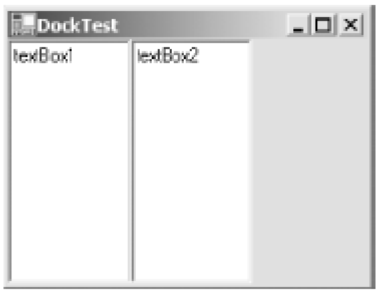
Figure 5-9. A left-docked control and a fill-docked control



5.4.2.1 Controlling dock order

Controls have an intrinsic order within their container. This is known as their *z-order* (pronounced "zee order"). The z-order of each control is unique within a given container. When two controls overlap in the same container, the control with the higher z-order eclipses the control with the lower z-order. Docking behavior is affected by z-order. When two or more controls are docked to the same side of a form, they are docked side by side. For example, the two text-box controls shown in Figure 5-10 are both docked left.

Figure 5-10. Docking two controls to the same side of a form



In such a situation, the relative position of each docked control is determined by its z-order. The control with the lowest z-order is positioned closest to the edge of the form. The control with the next higher z-order is placed next to that, and so on. In Figure 5-10, textBox1 has the lower z-order, and textBox2 has the higher z-order. So how is z-order determined? In code, z-order is determined by the order in which controls are added to their container's Controls collection. The first control added through the collection's Add method has the highest z-order; the last control added has the lowest z-order. For example, to produce the display shown in Figure 5-10, textBox2 must be added to the Controls collection first, as shown here:

```
Me.Controls.Add(Me.textBox2)
Me.Controls.Add(Me.textBox1)
```

This results in textBox1 having the lower z-order and therefore getting docked directly to the edge of the form. If the Controls collection's AddRange method is used to add an array of controls to a container, the first control in the array has the highest z-order and the last control in the array has the lowest z-order. For example, textBox1 and textBox2 in Figure 5-10 could have been added to their container using this code:

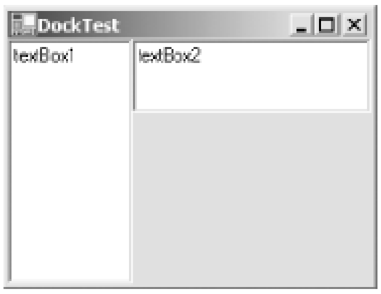
```
' Assumes Imports System.Windows.Forms
Me.Controls.AddRange(New Control( ) {Me.textBox2, Me.textBox1})
```

After controls have been added to their container's Controls collection, you can change their z-order by calling the Control class's SendToBack or BringToFront methods. The SendToBack method gives the control the lowest z-order within its container (causing it to dock closest to the edge of the form). The BringToFront method gives the control the highest z-order within its container (causing it to dock furthest from the edge of the form). For example, the following code forces the controls to dock as shown in Figure 5-10, regardless of the order in which they were added to the form's Controls collection:

```
    Me.textBox2.BringToFront( )
```

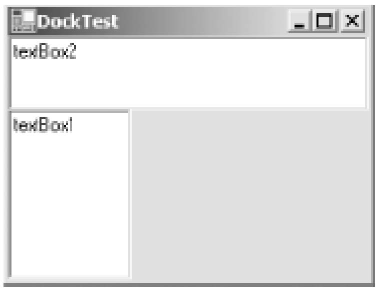
The order in which the controls are instantiated in code and the order in which their respective Dock properties are set do not affect the controls' z-orders. When you design forms using Visual Studio .NET's Windows Forms Designer, the controls added most recently have the highest z-orders and therefore dock furthest from the edges of their containers. The z-orders of the controls can be changed within the designer by right-clicking on a control and choosing "Bring to Front" or "Send to Back." When one of the controls in a container is fill-docked, that control should be last in the dock order (i.e., it should have the highest z-order), so that it uses only the space that is not used by the other docked controls. Dock order also comes into play when controls are docked to adjacent sides of a container, as shown in Figure 5-11.

Figure 5-11. Docking two controls to adjacent sides of a form



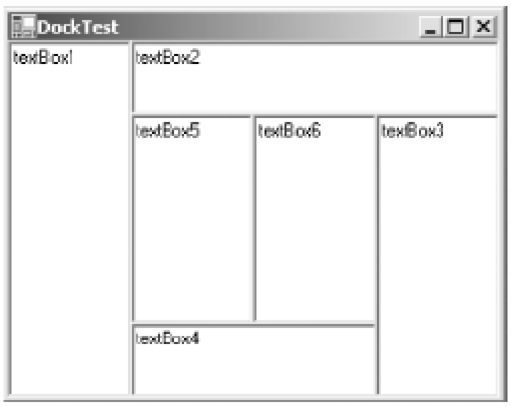
The control with the lowest z-order is docked first. In Figure 5-11, textBox1 has the lowest z-order and so is docked fully to the left side of the form. textBox2 is then docked to what remains of the top of the form. If the z-order is reversed, the appearance is changed, as shown in Figure 5-12.

Figure 5-12. Reversing the dock order



In Figure 5-12, textBox2 has the lowest z-order and so is docked first, taking up the full length of the side to which it is docked (the top). Then textBox1 is docked to what remains of the left side of the form. This behavior can be exploited to provide complex docking arrangements, such as the one shown in Figure 5-13.

Figure 5-13. Complex docking arrangements are easily created



The text boxes in Figure 5-13 are in ascending z-order (textBox1 is lowest; textBox6 is highest). The Dock properties of the controls are set like this:

```
    textBox1.Dock = DockStyle.Left
    textBox2.Dock = DockStyle.Top
```

```
textBox3.Dock = DockStyle.Right
textBox4.Dock = DockStyle.Bottom
textBox5.Dock = DockStyle.Left
textBox6.Dock = DockStyle.Fill
```

5.4.2.2 The Splitter control

By default, the internal edges of docked controls aren't draggable. To make the edge of a given docked control draggable, add a Splitter control to the form. The Splitter control must appear in the dock order immediately after the control whose edge is to be draggable, so the Splitter control's zorder must be just above that control's z-order. Further, the Splitter control must be docked to the same edge of the form as the control whose edge is to be draggable. The Splitter control provides a vertical splitter if docked left or right and a horizontal splitter if docked top or bottom. Consider again Figure 5-9. To make the edge between the two text-box controls draggable, a Splitter control must be added to the form (call it splitter1), and the z-order must be textBox2 (highest), then splitter1, and then textBox1 (lowest). This can be accomplished in the Visual Studio .NET Windows Forms Designer by adding the controls to the form in dock order (textBox1, splitter1, textBox2). As previously mentioned, if the controls have already been added to the form, their z-order can be rearranged by right-clicking the controls one-by-one in dock order and choosing "Bring To Front". The z-order can also be controlled in code, as previously discussed. It is possible to automate the task of Splitter creation. The subroutine in Example 5-1 takes as a parameter a reference to a docked control and instantiates a Splitter control that makes the given control's inside edge draggable.

Example 5-1. Dynamically creating a Splitter control

```
Shared Sub AddSplitter(ByVal ctl As Control)
    ' Create a Splitter control.
    Dim split As New Splitter( )

    ' Get the Controls collection of the given control's container

Dim controls As System.Windows.Forms.Control.ControlCollection _
    = ctl.Parent.Controls

    ' Add the Splitter to the same container as the given control.
    controls.Add(split)
    ' Move the Splitter control to be immediately prior to the given
    ' control in the Controls collection. This causes the Splitter
    ' control's z-order to be just above the given control's z-order,
    ' which in turn means that the Splitter control will be docked
    ' immediately following the given control.controls.SetChildIndex(split,
    controls.GetChildIndex(ctl))
    ' Dock the Splitter control to the same edge as the given control.
    split.Dock = ctl.Dock

End Sub
```

5.1 - 5.4 Check Your Progress

Fill in the blanks

1. ………………………… are the classes derived from the component class defined in the System CompoentModel namespace
2. …..………… property allows you to manage a collection of images.
3. The …………………………... property lets a control anchor to any or all sides of its container.
4. Controls have an intrinsic order within their container known as ………………………………..

5.5 COMMON DIALOG BOXES

There are several classes that implement common dialog boxes, such as color selection and print setup. These classes all derive from the CommonDialog class and, therefore, all inherit the ShowDialog method. The syntax of the ShowDialog method is:

Public Function Show Dialog( ) As System.Windows.Forms.DialogResult Calling the ShowDialog method causes the dialog box to be displayed modally, meaning that other windows in the application can't receive input focus until the dialog box is dismissed. The call is asynchronous, meaning that code following the call to the ShowDialog method isn't executed until the dialog box is dismissed. The return value of the ShowDialog method is of type DialogResult (defined in the System.Windows.Forms namespace). DialogResult is an enumeration that defines several values that a dialog box could return. The common dialog boxes, however, return only OK or Cancel, indicating whether the user selected the OK or Cancel button, respectively. In Visual Studio .NET's Windows Forms Designer, common dialog boxes are added to forms in much the same way that controls and nonvisual components are. Just select the desired dialog box from the Windows Forms tab of the Toolbox. As with nonvisual components, a representation of the dialog box appears within a separate pane rather than directly on the form that is being designed. The properties of the component can then be set in Visual Studio .NET's Properties window. The Windows Forms Designer creates code that declares and instantiates the dialog box, but you must add code to show the dialog box and use the values found in its properties. Alternatively, common dialog-box components can be instantiated and initialized directly in code, bypassing the Windows Forms Designer. For example, this method instantiates and shows a ColorDialog component:

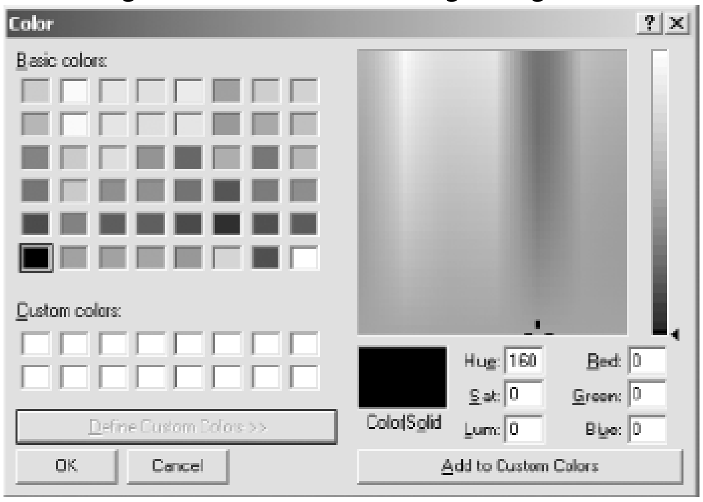
```
' Assumes Imports System.Windows.Forms
Public Class SomeClass
    Public Sub SomeMethod( )
    Dim clrDlg As New ColorDialog( )
    If clrDlg.ShowDialog = DialogResult.OK Then
    ' Do something with the value found in clrDlg.Color.
    End If
    End Sub
End Class
```

The remainder of this section briefly describes each of the common dialog-box components.

5.5.1 ColorDialog

The ColorDialog component displays a dialog box that allows the user to choose a color. After the user clicks OK, the chosen color is available in the ColorDialog object's Color property. The Color property can also be set prior to showing the dialog box. This causes the dialog box to initially display the given color. Figure 5-14 shows an example of the ColorDialog dialog box.

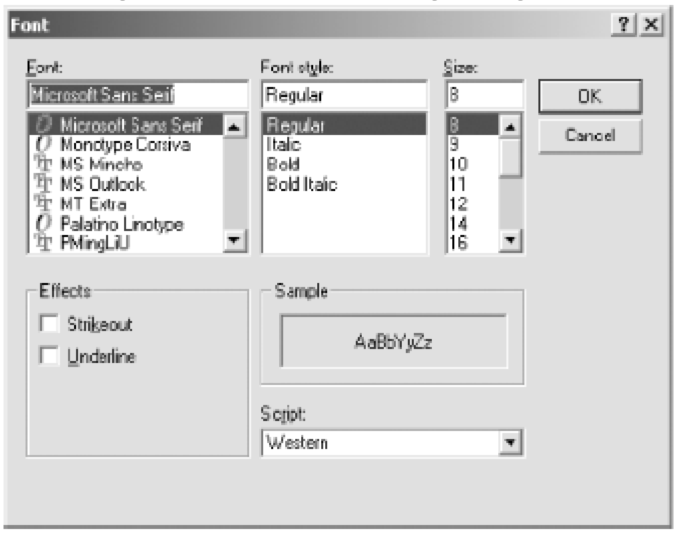
Figure 5-14. The ColorDialog dialog box



5.5.2 FontDialog

The FontDialog component displays a dialog box that allows the user to choose a font. After the user clicks OK, the chosen font is available in the FontDialog object's Font property. The Font property can also be set prior to showing the dialog box. This causes the dialog box to initially display the given font. Figure 5-15 shows an example of the FontDialog dialog box.

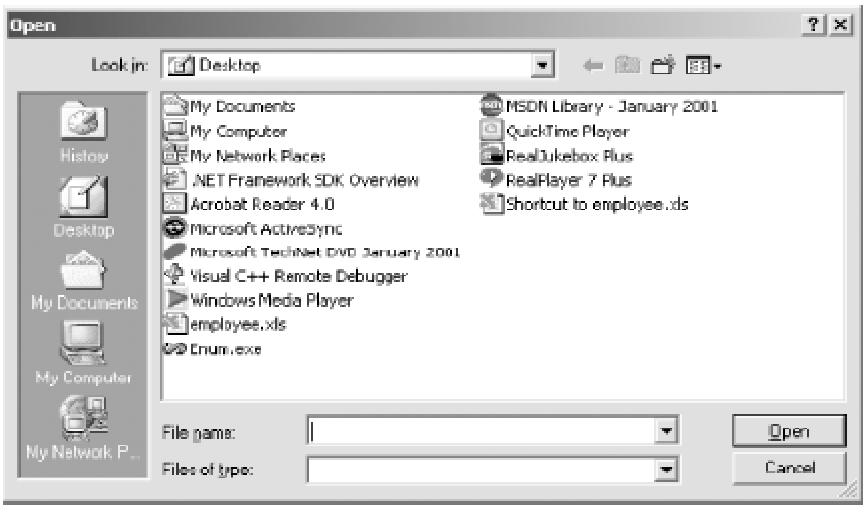
Figure 5-15. The FontDialog dialog box



5.5.3 OpenFileDialog

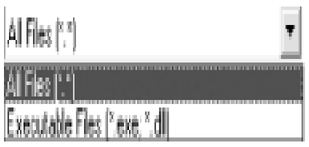
The OpenFileDialog component displays a dialog box that allows the user to choose a file to open. After the user clicks OK, the name of the file (including the path) is available in the OpenFileDialog object's FileName property. The FileName property can be set prior to showing the dialog box. This causes the dialog box to initially display the given filename. Figure 5-16 shows an example of the OpenFileDialog dialog box.

Figure 5-16. The OpenFileDialog dialog box



In most cases, your applications should set the InitialDirectory, Filter, and FilterIndex properties prior to calling ShowDialog. This is not necessary for proper functioning of the dialog box, but it will give your application a more professional look and feel. The InitialDirectory property determines which directory is shown when the dialog box first appears. The default is an empty string, which causes the dialog box to display the user's My Documents directory. The Filter property holds a String value that controls the choices in the "Files of type" drop-down list. The purpose of this drop-down list is to let the user limit the files shown in the dialog box based on filename extension. A typical example is shown in Figure 5-17.

Figure 5-17. A typical "Files of type" drop-down list



Even though the "Files of type" list can include many items and each item can represent many filename extensions, a single String property represents the whole thing. Here's how it works:

. Each item in the drop-down list is represented by a substring having two parts separated by the vertical bar character (|).

1. The first part is the description that appears in the drop-down list (e.g., "All Files (*.*)").

2. The second part is the corresponding filter (e.g., "*.*"). Taking them together and adding the vertical bar character, the first item in the list in Figure 5-17 is represented by the substring:

"All Files (*.*)|*.*"

1. If a given item has multiple filters, the filters are separated by semicolons (;). The second item in the list in Figure 5-17 is therefore represented by:

"Executable Files (*.exe; *.dll)|*.exe;*.dll"

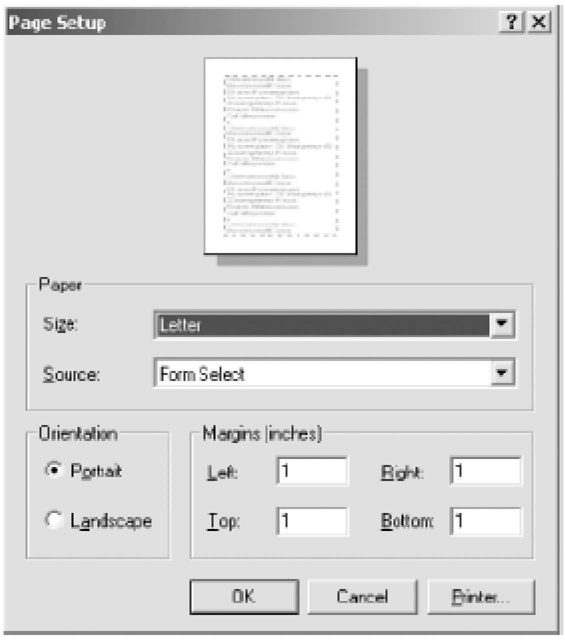
2. The value to assign to the Filter property is the concatenation of all the substrings thus attained, again separated by the vertical bar character. Therefore, the Filter property value that produced the drop-down list in Figure 5-17 is:

"All Files (*.*)|*.*|Executable Files (*.exe; *.dll)|*.exe;*.dll"
The default value of the Filter property is an empty string, which results in an empty "Files of type" drop-down list.The FilterIndex property determines which filter is in force when the dialog box is initially shown. This is a 1-based index that refers to the Filter string. For example, referring again to Figure 5-17, if the FilterIndex property is set to 1, the "All Files" item will be selected when the dialog box is shown. If the FilterIndex is set to 2, the "Executable Files" item will be shown. The default value is 1.

5.5.4 PageSetupDialog

The PageSetupDialog component displays a dialog box that allows the user to choose page settings for a document. Certain properties in the PageSetupDialog object must be set prior to showing the dialog box. After the user clicks OK, new settings can be read from the object Figure 5-18 shows an example of the PageSetupDialog dialog box.

Figure 5-18. The PageSetupDialog dialog box

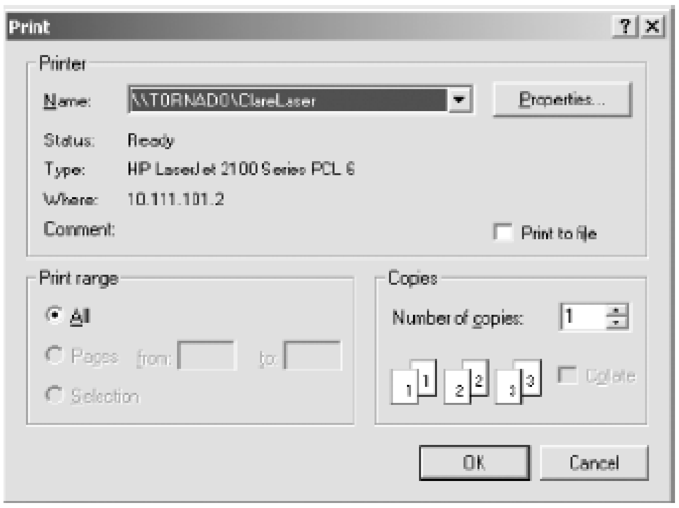


Size: Letter
Source: Form Select
(● Portrait) ○ Landscape
Left: 1
Right: 1
Top: 1
Bottom: 1

5.5.5 PrintDialog

The PrintDialog component displays a dialog box that allows the user to choose printer settings for a document. Certain properties in the PrintDialog object must be set prior to showing the dialog box. After the user clicks OK, new settings can be read from the object. See "Printing" in Chapter 4 for details. Figure 5-19 shows an example of the PrintDialog dialog box.

Figure 5-19. The PrintDialog dialog box

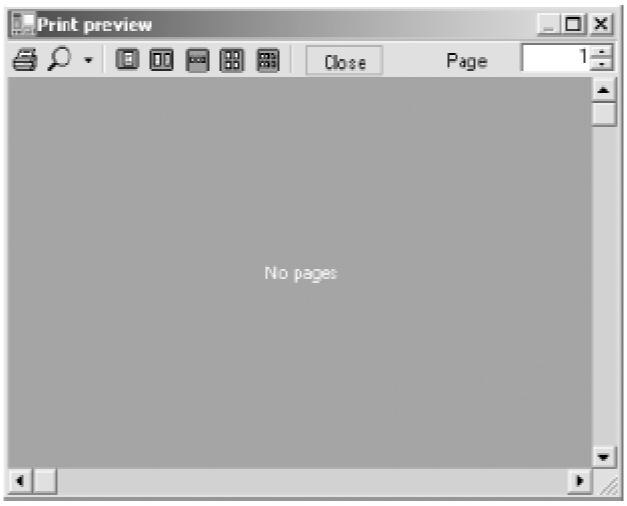


Name: \\TORNADO\ClareLaser
Status: Ready
Type: HP LaserJet 2100 Series PCL 6
Where: 10.111.101.2
Comment:
☐ Print to file
(● All) ○ Pages from: to: ○ Selection
Number of copies: 1
☐ Collate

5.5.6 PrintPreviewDialog

The PrintPreviewDialog component displays a dialog box that allows the user to view a document before printing it. Prior to showing the dialog box, the PrintPreviewDialog object must be loaded with information about the document to be printed. See "Printing" in Chapter 4 for details. The dialog box itself displays a preview of the printed version of the document, allowing the user to navigate through it. Figure 5-20 shows an example of the PrintPreviewDialog dialog box, although this example doesn't have a document loaded.

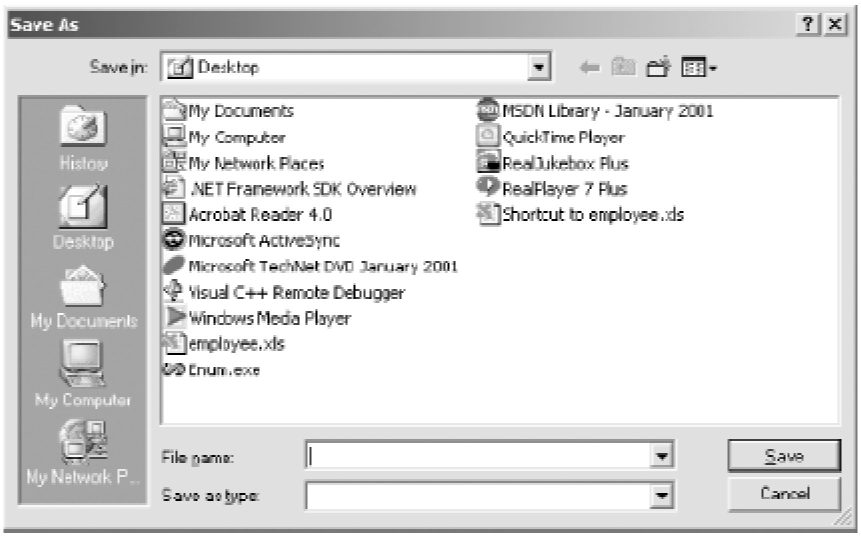
Figure 5-20. The PrintPreviewDialog dialog box



5.5.7 SaveFileDialog

The SaveFileDialog component displays a dialog box that allows the user to specify a filename to be used for saving. After the user clicks OK, the name of the file (including the path) is available in the SaveFileDialog object's FileName property (which can be set prior to showing the dialog box). This causes the dialog box to initially display the given filename. Figure 5-21 shows an example of the SaveFileDialog dialog box.

Figure 5-21. The SaveFileDialog dialog box



As with the OpenFileDialog component, most applications should set the InitialDirectory, Filter, and FilterIndex properties prior to calling ShowDialog. Their usage with the SaveFileDialog component is precisely the same as with OpenFileDialog.

5.6 MENUS

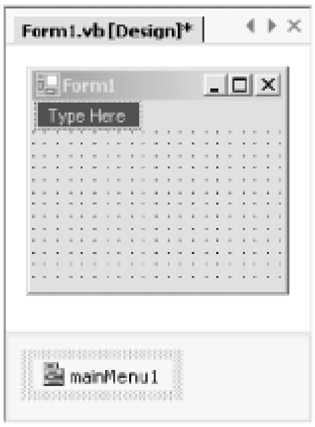
The Windows Forms library provides three components for creating and managing menus:

1. The MainMenu class manages the display of a menu across the top of a form.
2. The ContextMenu class manages the display of a context menu (also known as a pop-up menu).
3. Menu or context menu.
   Menus can be created either in the Visual Studio .NET Windows Forms Designer or programmatically.

5.6.1 Adding Menus in the Visual Studio .NET Windows Forms Designer

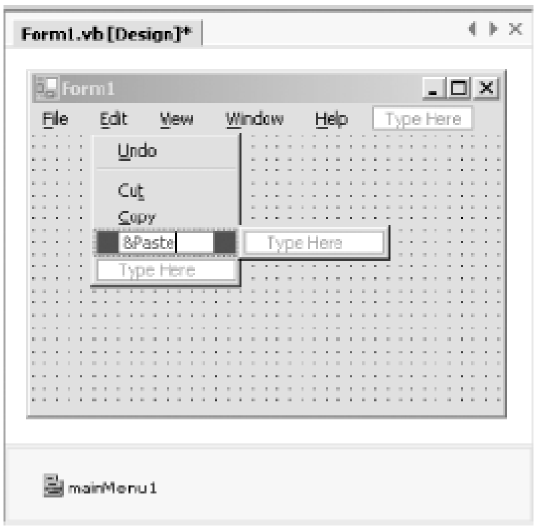
Visual Studio .NET includes an in-place WYSIWYG menu editor that is a dramatic improvement over the menu editor in Visual Basic 6. To create an application menu in the Windows Forms Designer, add a MainMenu component to the form. This causes a representation of the component to appear in the lower pane of the form's design view. When the component is selected, Visual Studio .NET adds a WYSIWYG editor to the top of the form, as shown in Figure 5-22.

Figure 5-22. In-place editing of a main menu



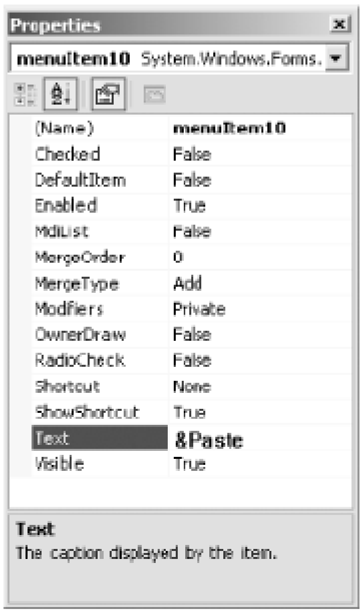
To create a menu item, type a value into the "Type Here" box. The menu editor automatically adds "Type Here" boxes to the right and beneath the box in which you are typing. As was the case in previous Windows development environments, typing an ampersand (&) within the menu name causes the character following the ampersand to be an accelerator key. Typing a single hyphen (-) within a "Type Here" box creates a menu-separator bar. Figure 5-23 shows a menu containing several items.

Figure 5-23. A menu containing several items



After a menu item has been added, its properties can be set in the Properties window. The Properties window of the Edit Paste item shown in Figure :

Figure 5-24. The Properties window display for the Edit Paste menu item shown in Figure 5-23



Properties

menuItem10 System.Windows.Forms.

(Name)	menuItem10
Checked	False
DefaultItem	False
Enabled	True
MdiList	False
MergeOrder	0
MergeType	Add
Modifiers	Private
OwnerDraw	False
RadioCheck	False
Shortcut	None
ShowShortcut	True
Text	&Paste
Visible	True

Text
The caption displayed by the item.

The properties shown in Figure 5-24 are:

Checked
Indicates whether a checkmark appears next to the menu item. The type is Boolean. The default is False.

DefaultItem
Indicates whether the menu item is the default item in its menu or submenu. Default menu items are displayed in bold. The type is Boolean. The default is False.

Enabled
Indicates whether the menu item is enabled. If this is False, the menu item is grayed and can't be selected. The type is Boolean. The default is True.

MdiList
In MDI applications, indicates whether the menu item should have subitems for each of the open MDI child forms. This property should be set to True for the Windows menu item. The type is Boolean. The default is False.

MergeOrder
In applications where two menus might be merged (such as an MDI application where a child form menu might be merged with its parent form menu), sorts the merged menu items based on this property. The type is Integer. The default is 0. See "MDI Applications" for more information.

MergeType
Indicates how two menu items having the same MergeOrder value should be merged. The value is of type MenuMerge (defined in the System.Windows.Forms namespace). The default is MenuMerge.Add. See "MDI Applications" in Chapter 4 for more information.

Modifiers
Specifies the declaration modifiers that are placed on the menu-item field declaration within the generated code. This is not actually a property of the MenuItem class. Rather, it becomes part of the field declaration in source code. The default is Private.

OwnerDraw
Indicates whether the menu item requires custom drawing. If you set this property to True, you must handle the menu item's DrawItem event. The type is Boolean. The default is False.

RadioCheck
If the Checked property is True, indicates whether to display the checkmark as a radio button instead. The type is Boolean. The default is False.

Shortcut
Specifies the shortcut key combination that invokes this menu item. The value is of type Shortcut (defined in the System.Windows.Forms namespace). The Shortcut enumeration defines a unique value for each potential shortcut key combination. The values of the Shortcut enumeration are shown in Table 5-1. The default is Shortcut.None. *Table 5-1. Values defined by the System.Windows.Forms.Shortcut*

Enumeration

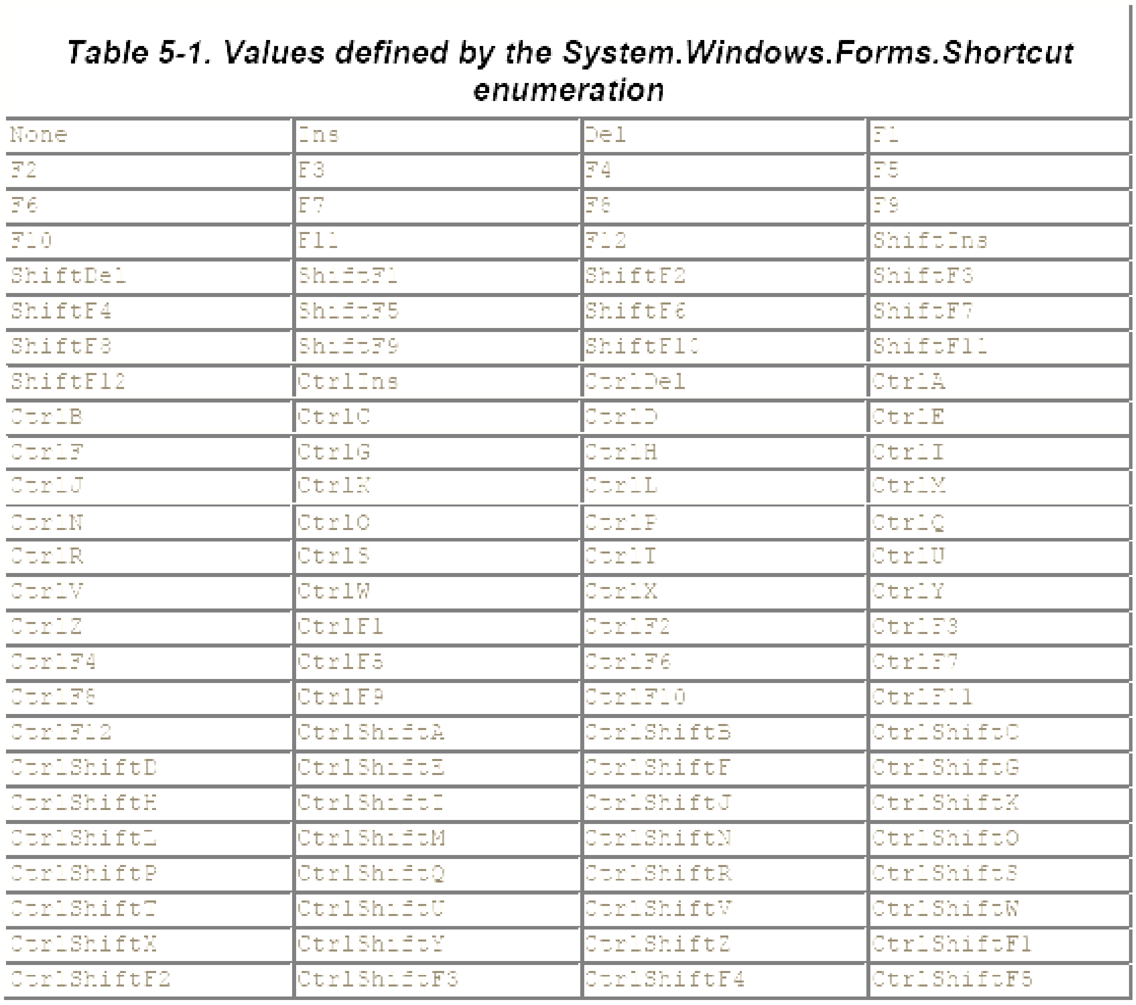
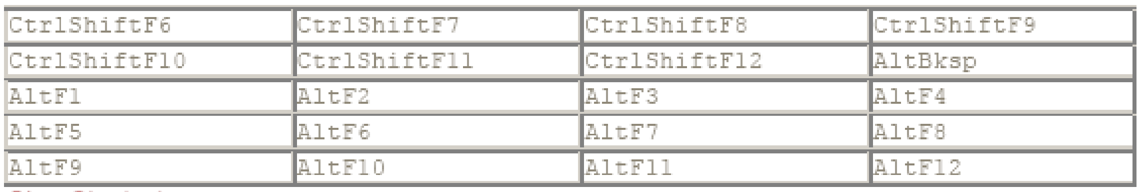



Table 5-1. Values defined by the System.Windows.Forms.Shortcut enumeration

None	Ins	Del	F1
F2	F3	F4	F5
F6	F7	F8	F9
F10	F11	F12	ShiftIns
ShiftDel	ShiftF1	ShiftF2	ShiftF3
ShiftF4	ShiftF5	ShiftF6	ShiftF7
ShiftF8	ShiftF9	ShiftF10	ShiftF11
ShiftF12	CtrlIns	CtrlDel	CtrlA
CtrlB	CtrlC	CtrlD	CtrlE
CtrlF	CtrlG	CtrlH	CtrlI
CtrlJ	CtrlK	CtrlL	CtrlM
CtrlN	CtrlO	CtrlP	CtrlQ
CtrlR	CtrlS	CtrlT	CtrlU
CtrlV	CtrlW	CtrlX	CtrlY
CtrlZ	CtrlF1	CtrlF2	CtrlF3
CtrlF4	CtrlF5	CtrlF6	CtrlF7
CtrlF8	CtrlF9	CtrlF10	CtrlF11
CtrlF12	CtrlShiftA	CtrlShiftB	CtrlShiftC
CtrlShiftD	CtrlShiftE	CtrlShiftF	CtrlShiftG
CtrlShiftH	CtrlShiftI	CtrlShiftJ	CtrlShiftK
CtrlShiftL	CtrlShiftM	CtrlShiftN	CtrlShiftO
CtrlShiftP	CtrlShiftQ	CtrlShiftR	CtrlShiftS
CtrlShiftT	CtrlShiftU	CtrlShiftV	CtrlShiftW
CtrlShiftX	CtrlShiftY	CtrlShiftZ	CtrlShiftF1
CtrlShiftF2	CtrlShiftF3	CtrlShiftF4	CtrlShiftF5
CtrlShiftF6	CtrlShiftF7	CtrlShiftF8	CtrlShiftF9
CtrlShiftF10	CtrlShiftF11	CtrlShiftF12	AltBksp
AltF1	AltF2	AltF3	AltF4
AltF5	AltF6	AltF7	AltF8
AltF9	AltF10	AltF11	AltF12

ShowShortcut
If a shortcut key combination is defined for the menu item, indicates whether the key combination should be shown on the menu. The type is Boolean. The default is True.

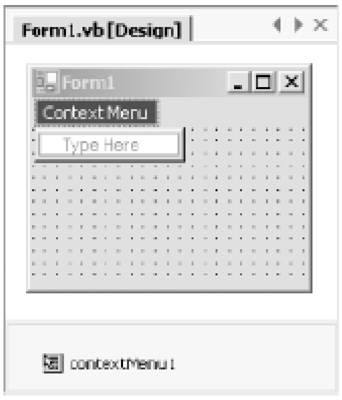
Text
Represents the text shown on the menu item. The type is String. The default is an empty string.

Visible

Indicates whether the menu item should be visible. The type is Boolean. The default is True. The Windows Forms Designer creates code that declares a field for the MainMenu object, as well as fields for the MenuItem objects that represent each menu item in the menu. The designer also creates code that instantiates the objects at runtime and sets their properties according to the values set in the IDE's Properties window. In addition, the top-level MenuItem objects are added to the MenuItems collection of the MainMenu object, and the lower-level MenuItem objects are added to the MenuItems collection of the menu of which they are submenus. Finally, the MainMenu object is assigned to the form's Menu property. To create a context menu in the Windows Forms Designer, add a ContextMenu component to the form. This causes a representation of the component to appear in the lower pane of the form's design view.

When the component is selected, Visual Studio .NET adds a WYSIWYG editor to the top of the form. Clicking on the editor drops down a "Type Here" box, as shown in Figure 5-25.

Figure 5-25. In-place editing of a context menu



Adding menu items and setting their properties work the same with context menus as with main menus. For a context menu to be displayed, it must be associated with some control on the form or with the form itself. When the user right-clicks on that control or form, the context menu is displayed. To make this association, perform these steps:

1. In the Windows Forms Designer, right-click the control or form that is to be associated with the context menu, and choose Properties.
2. In the Properties window, find the ContextMenu property, and click the arrow of the drop-down list associated with it.
3. The drop-down list displays the ContextMenu objects that are defined on the current form.

Choose the desired one. The same effect can be achieved in code by assigning the ContextMenu object to the ContextMenu property of a control or form, like this:

```
' Somewhere within the definition of a Form class.
Me.Button1.ContextMenu = Me.ContextMenu1
```

5.6.2 Programmatically Creating Menus

Main menus and context menus can be instantiated and populated directly in code. Example 5-2 shows how to create a simple main menu having File and Edit items, with the File item having an Exit item. Compile it from the command line with this command:

```
vbc filename.vb /r:System.dll,System.Drawing.dll,System.Windows.Forms.dll
/t:winexe
```

The code in Example 5-2 is similar to the code that would be created by the Windows Forms Designer if its visual menu editor were used to create this menu. The essential steps are:

1. Declare a MainMenu object to represent the menu.
2. Declare a MenuItem object for each menu item.
3. Instantiate the MainMenu object and all the MenuItem objects.

4. Set the properties of the MenuItem objects (such as the Text property) as desired.
5. Add the top-level MenuItem objects to the MenuItems collection of the MainMenu object, then add the lower-level MenuItem objects to the MenuItems collection of the MenuItem object under which they should appear.

Example 5-2. Creating a main menu in code

```
Imports System.Drawing
Imports System.Windows.Forms
    Module modMain
    <System.STAThreadAttribute( )> Public Sub Main( )
    System.Threading.Thread.CurrentThread.ApartmentState = _
    System.Threading.ApartmentState.STA
    System.Windows.Forms.Application.Run(New Form1( ))
End Sub
End Module


Public Class Form1
    Inherits Form
    ' Declare the main menu component.
    Private myMenu As MainMenu
    ' Declare the menu items that will be in the menu. Use the
    ' WithEvents keyword so that event handlers can be added later
    Private WithEvents mnuFile As MenuItem
    Private WithEvents mnuFileExit As MenuItem
    Private WithEvents mnuEdit As MenuItem

    Public Sub New( )
        ' Instantiate the menu objects.
        Programming Visual Basic .NET

        myMenu = New MainMenu( )
        mnuFile = New MenuItem( )
        mnuFileExit = New MenuItem( )
        mnuEdit = New MenuItem( )
        ' Set the properties of the menu items.
        mnuFile.Text = "&File"
        mnuFileExit.Text = "E&xit"
        mnuEdit.Text = "&Edit"
        ' Connect the menu items to each other and to the main menu.
        mnuFile.MenuItems.Add(mnuFileExit)
        myMenu.MenuItems.Add(mnuFile)
        myMenu.MenuItems.Add(mnuEdit)
        ' Connect the main menu to the form.
        Me.Menu = myMenu

    End Sub
End Class
```

5.6.3 Handling Menu Events

User interaction with a menu causes menu events to be fired. The most common menu event is the Click event of the MenuItem class, which fires when a user clicks a menu item. Here is an example of a Click event handler (this code could be added to the Form1 class of Example 5-2):

```
Private Sub mnuFileExit_Click( _ByVal sender As Object, _
ByVal e As EventArgs _) Handles mnuFileExit.Click
    Me.Close( )
End Sub
```

The events of the MenuItem class are:

Click
Fired when the menu item is chosen either by clicking it with the mouse or by pressing a shortcut key combination defined for the menu item. The syntax of the Click event is:

Public Event Click As System.EventHandler

This is equivalent to:

Public Event Click(ByVal *sender* As Object, ByVal *e* As System.EventArgs)

Disposed
Fired when the MenuItem object's Dispose method is called. The syntax of the Disposed
event is:

Public Event Disposed As System.EventHandler

This is equivalent to:

Public Event Disposed(ByVal
sender As Object, ByVal
e As System.EventArgs)

The event is inherited from the Component class.

DrawItem
Fired when the menu item needs to be drawn, when the MenuItem object's OwnerDraw
property is True. The syntax of the DrawItem event is:

Public Event DrawItem As System.Windows.Forms.DrawItemEventHandler
This is equivalent to:
Public Event DrawItem(_ByVal *sender* As Object, _
ByVal *e* As System.Windows.Forms.DrawItemEventArgs_)

The *e* parameter, of type DrawItemEventArgs, provides additional information that is needed for drawing the menu item. The properties of the DrawItemEventArgs class are:

BackColor
The background color that should be used when drawing the item. The type is Color (defined in the System.Drawing namespace).

Bounds
The bounding rectangle of the menu item. The type is Rectangle (defined in the System.Drawing namespace).

Font
The font that should be used when drawing the item. The type is Font (defined in the System.Drawing namespace).

ForeColor
The foreground color that should be used when drawing the item. The type is Color (defined in the System.Drawing namespace).

Graphics
The graphics surface on which to draw the item. The type is Graphics (defined in the System.Drawing namespace).
Index The index of the menu item within its parent menu. The type is Integer.

State
The state of the menu item. The type is DrawItemState (defined in the System.Windows.Forms namespace). DrawItemState is an enumeration that defines the values None, Selected, Grayed, Disabled, Checked, Focus, Default, HotLight, Inactive, NoAccelerator, NoFocusRect, and ComboBoxEdit.

MeasureItem

Fired prior to firing the DrawItem event when the MenuItem object's OwnerDraw property is True. The MeasureItem event allows the client to specify the size of the item to be drawn.The syntax of the MeasureItem event is:

```
Public Event MeasureItem As
System.Windows.Forms.MeasureItemEventHandler
```

This is equivalent to:

```
Public Event MeasureItem( _
ByVal sender As Object, _
ByVal e As System.Windows.Forms.MeasureItemEventArgs _ )
```

The *e* parameter, of type MeasureItemEventArgs, provides additional information needed by the event handler and provides fields that the event handler can set to communicate the item size to the MenuItem object. The properties of the MeasureItemEventArgs are:

Graphics

The graphics device upon which the menu item will be drawn. This is needed so the client can determine the scale of the device upon which the menu item will be rendered. The type is Graphics (defined in the System.Drawing namespace).

Index

The index of the menu item within its parent menu. The type is Integer.

ItemHeight

The height of the menu item. The type is Integer.

ItemWidth

The width of the menu item. The type is Integer.

Popup

Fired when the submenu is about to be displayed, when a menu item has subitems associated with it. This provides the client with an opportunity to set the menu states (checked, enabled, etc.) of the submenu items to match the current program state. The syntax of the Popup event is:

```
Public Event Popup As System.EventHandler
```

This is equivalent to:

```
Public Event Popup(ByVal sender As Object, ByVal e As System.EventArgs)
```

Select

Fired when the user places the mouse over the menu item or when the user highlights the menu item by navigating to it with the keyboard arrow keys. The syntax of the Select event is: Public Event Select As System.EventHandler This is equivalent to:

```
Public Event Select(ByVal sender As Object, ByVal e As System.EventArgs)
```

The ContextMenu class also exposes a Popup event, which is fired just before the context menu is displayed. The syntax of the Popup event is:

```
Public Event Popup As System.EventHandler
```

This is equivalent to:

```
Public Event Popup(ByVal sender As Object, ByVal e As System.EventArgs)
```

5.6.4 Cloning Menus

Sometimes menu items and their submenus need to appear on more than one menu. A common example is an application that has context menus containing some of the same functionality as the application's main menu. However, MenuItem objects don't work correctly if they are assigned to more than one menu. To provide an easy solution for developers who need to duplicate functionality on multiple menus, the MenuItem class provides the CloneMenu method. This method returns a new MenuItem object whose properties are set the same as those of the MenuItem object on which the CloneMenu method is called. If the original MenuItem object has submenus, the submenu MenuItem objects are cloned as well. Example 5-3 shows the complete code for a program that has both a main menu and a context menu. It can be compiled from the command line with this command:

vbc *filename.vb* /r:System.dll,System.Drawing.dll,System.Windows.Forms.dll /t:winexe

The code in Example 5-3 sets up a menu item labeled Format that has four options beneath it: Font, ForeColor, BackColor, and Reset. After this menu structure is set up, it is added to the MainMenu object, which is then attached to the form, as shown here:

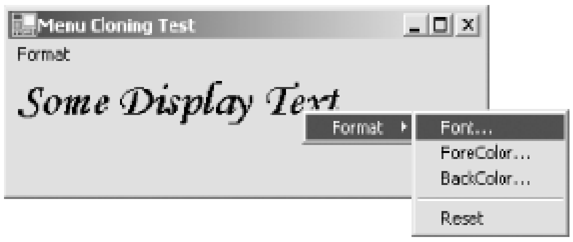
```
MainMenu1.MenuItems.Add(mnuFormat)
Menu = MainMenu1
```

An identical menu structure is then created and assigned to the ContextMenu object, which is then attached to the Label1 control, as shown here:

```
    ContextMenu1.MenuItems.Add(mnuFormat.CloneMenu( ))
    Label1.ContextMenu = ContextMenu1
```

The CloneMenu method even detects the event handlers that are defined for the MenuItems being cloned and automatically registers those handlers with the corresponding events on the newly created MenuItem objects. This means that the event handlers shown in Example 5-3 handle the events both from the main menu and from the context menu. Figure 5-26 shows the running application with the context menu displayed (the Label control was right-clicked).

Figure 5-26. The display produced by Example 5-3



Example 5-3. Cloning menus

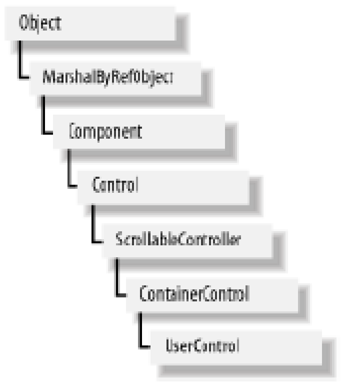
```
Imports System
Imports System.Drawing
Imports System.Windows.Forms
    Module modMain
    <STAThreadAttribute( )> Public Sub Main( )
    System.Threading.Thread.CurrentThread.ApartmentState = _
    System.Threading.ApartmentState.STA
    Application.Run(New Form1( ))
End Sub
End Module
Public Class Form1
    Inherits Form
    ' This member references the label used for displaying text.
    Private WithEvents Label1 As New Label( )
    ' These members store original font and color properties of the
    ' Label1 control.
    Private origFont As Font
    Private origForeColor As Color
    Private origBackColor As Color
    ' These members hold the MainMenu and ContextMenu objects.
    Private WithEvents MainMenu1 As New MainMenu( )
    Private WithEvents ContextMenu1 As New ContextMenu( )
    ' These members hold the MenuItem objects.
    Private WithEvents mnuFormat As New MenuItem( )
    Private WithEvents mnuFormatFont As New MenuItem( )
    Private WithEvents mnuFormatForeColor As New MenuItem( )
    Private WithEvents mnuFormatBackColor As New MenuItem( )
    Private mnuSeparator As New MenuItem( )
    Private WithEvents mnuFormatReset As New MenuItem( )
    Public Sub New( )
    MyBase.New( )
    ' Set up the Format menu.
    mnuFormat.Text = "F&ormat"
    mnuFormatFont.Text = "&Font..."
    mnuFormatForeColor.Text = "F&oreColor..."
    mnuFormatBackColor.Text = "&BackColor..."
    mnuSeparator.Text = "-"
    mnuFormatReset.Text = "&Reset"
    mnuFormat.MenuItems.AddRange(New MenuItem( ) {mnuFormatFont, _
    mnuFormatForeColor, mnuFormatBackColor, mnuSeparator, _
    mnuFormatReset})

    ' Attach the Format menu to the main menu and attach the main
    ' menu to the form.
    MainMenu1.MenuItems.Add(mnuFormat)
    Menu = MainMenu1
    ' Clone the Format menu, attach the clone to the context menu, and
    ' attach the context menu to the Label control.
    ContextMenu1.MenuItems.Add(mnuFormat.CloneMenu( ))
    Label1.ContextMenu = ContextMenu1
    ' Set up non-menu-related properties of the form and the label.
    AutoScaleBaseSize = New Size(5, 13)
    ClientSize = New Size(312, 81)
    Controls.Add(Label1)
    Menu = MainMenu1
    Name = "Form1"
    Text = "Menu Cloning Test"
    Label1.Text = "Some Display Text"
    Label1.AutoSize = True
' Save the original font and color properties of the Label1 control.
    origFont = Label1.Font
    origForeColor = Label1.ForeColor
    origBackColor = Label1.BackColor
End Sub
' Font Click event handler
Private Sub mnuFormatFont_Click( _ByVal sender As Object, _ByVal e As EventArgs
_) Handles mnuFormatFont.Click

    Dim dlg As New FontDialog( )
    dlg.Font = Label1.Font
    If dlg.ShowDialog = DialogResult.OK Then
    Label1.Font = dlg.Font
    End If
    dlg.Dispose( )
End Sub
' ForeColor Click event handler
Private Sub mnuFormatForeColor_Click( _ByVal sender As Object, _
ByVal e As EventArgs _) Handles mnuFormatForeColor.Click

    Dim dlg As New ColorDialog( )
    dlg.Color = Label1.ForeColor
    If dlg.ShowDialog = DialogResult.OK Then
    Label1.ForeColor = dlg.Color
    End If
    dlg.Dispose( )
End Sub
' BackColor Click event handler
```

```
Private Sub mnuFormatBackColor_Click( _ByVal sender As Object, _
ByVal e As EventArgs _) Handles mnuFormatBackColor.Click

Dim dlg As New ColorDialog( )
dlg.Color = Label1.BackColor
    If dlg.ShowDialog = DialogResult.OK Then
        Label1.BackColor = dlg.Color
    End If
dlg.Dispose( )
End Sub
' Resent Click event handler
Private Sub mnuFormatReset_Click( _ByVal sender As Object, _
ByVal e As EventArgs _) Handles mnuFormatReset.Click
        Label1.Font = origFont
        Label1.ForeColor = origForeColor
        Label1.BackColor = origBackColor
    End Sub
End Class
```

5.7 CREATING A CONTROL

A control is a component with a visual representation. The Windows Forms class library provides the base functionality for controls through the Control class (defined in the System.Windows.Forms namespace). All controls derive directly or indirectly from the Control class. In addition, Windows Forms provides a class called UserControl for the purpose of making it easy to write custom control classes. The derivation of the UserControl class is shown in Figure 5-27.

Figure 5-27. The derivation hierarchy of the UserControl class



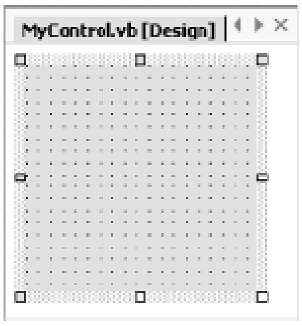
5.7.1 Building Controls from Other Controls

The easiest way to create a new control is to aggregate and modify the functionality of one or more existing controls. To do this in Visual Studio .NET's Windows Forms Designer, perform the following steps:

1. Choose Project Add User Control from the main menu.

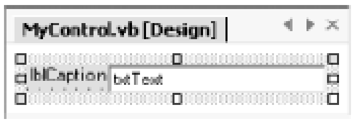
2. Type the name of the *.vb* file that will hold the code for the control, and click OK. The designer displays a blank user control in design mode, as shown in Figure 5-28.

Figure 5-28. A blank user control in Visual Studio .NET's Windows Forms Designer



3. Add controls from the Toolbox window just as you would when laying out a form. Controls that are made part of another control are called *constituent controls* . Figure 5-29 shows a user control that has two constituent controls: a Label and a TextBox.

Figure 5-29. A user control with two constituent controls



The user control shown in Figure 5-29 is a good start on a captioned text-box control—a text box that carries around its own caption. These additional steps would also be helpful:

a. Set the Label control's AutoSize property to True so the label expands to the size needed for displaying its text.
b. Dock the Label control to the left side of the user control. This allows the TextBox control to be docked to the Label control. The benefit of docking the TextBox control to the Label control is that the TextBox control will move whenever the Label control resizes itself.
c. Dock the TextBox control to fill the remainder of the space.

4. Add any appropriate properties to the user control. This could involve overriding properties inherited from base classes or creating new properties. For the captioned text-box user control, do the following:
a. Override the Text property. The purpose of this property is to allow the client environment to set and read the text displayed in the constituent TextBox control.
Here is the code:
b. Public Overrides Property Text( ) As String
c. Get
d. Return txtText.Text
e. End Get
f. Set(ByVal Value As String)
g. txtText.Text = Value
h. End Set
End Property
As can be seen from the code, the user control's Text property is simply mapped to the TextBox control's Text property.
i. Create a new property for setting the caption text. Here is the code:
j. Public Property Caption( ) As String
k. Get
l. Return lblCaption( ).Text
m. End Get
n. Set(ByVal Value As String)
o. lblCaption( ).Text = Value
p. End Set

End Property

In this case, the Caption property is mapped to the Label control's Text property.

5. Add any appropriate events to the user control. This could involve invoking base-class events or creating and invoking new events. For the captioned text-box user control, do the following:
   a. Add a handler for the constituent TextBox control's TextChanged property. Within the handler, invoke the base class's TextChanged event. Here is the code:
   b. Private Sub txtText_TextChanged(_
   c. ByVal sender As Object, _
   d. ByVal e As EventArgs _
   e.) Handles txtText.TextChanged
   f. Me.OnTextChanged(e)
      End Sub
      Notice that this code calls the OnTextChanged method, which is declared in the Control class. The purpose of this method is to fire the TextChanged event, which is also declared in the Control class (Visual Basic .NET doesn't provide a way to fire a base-class event directly.) There are On*EventName* methods for each of the events defined in the Control class.The OnTextChanged method is overridable. If you override it in your derived class, be sure that your overriding method calls MyBase.OnTextChanged. If you don't, the TextChanged event won't be fired.
   g. Declare a new event to notify the client when the user control's Caption property is changed. Name the event CaptionChanged. Add a handler for the Label control's TextChanged event and raise the CaptionChanged event from there. Here's the code:
   h. Event CaptionChanged As EventHandler
   i.
   j. Private Sub lblCaption_TextChanged(_
   k. ByVal sender As Object, _
   l. ByVal e As EventArgs _
   m.) Handles lblCaption.TextChanged
   n. RaiseEvent CaptionChanged(Me, EventArgs.Empty)
      End Sub
      Note the arguments to the CaptionChanged event. The value Me is passed as the sender of the event, and EventArgs.Empty is passed as the event arguments. The Empty field of the EventArgs class returns a new, empty EventArgs object.

6. Add any appropriate attributes to the syntax elements of the class. For example, the Caption property will benefit from having a Category attribute and a Description attribute, as shown in bold here:

8. <Category("Appearance"), _
9. **Description("The text appearing next to the textbox.")> _**
10. Public Property _
11. Caption( ) As String
12. Get
13. Return lblCaption( ).Text
14. End Get
15. Set(ByVal Value As String)
16. lblCaption( ).Text = Value
17. End Set
End Property
These attributes are compiled into the code and are picked up by the Visual Studio .NET IDE.
The Category attribute determines in which category the property will appear in the Properties window. The Description attribute determines the help text that will be displayed in the Properties window when the user clicks on that property. See "Component Attributes" in Chapter 4 for a list of attributes defined in the System.ComponentModel namespace.
This is all very similar to creating forms. As with forms, custom controls can be defined directly in code. Example 5-4 shows a complete class definition for the captioned text-box control, created without the aid of the Windows Forms Designer. It can be compiled from the command line with this command:

vbc *filename.vb* /r:System.dll,System.Drawing.dll,System.Windows.Forms.dll /t:library
(Note the /t:library switch for creating a *.dll* rather than an *.exe* file.)

Example 5-4. A custom Control class

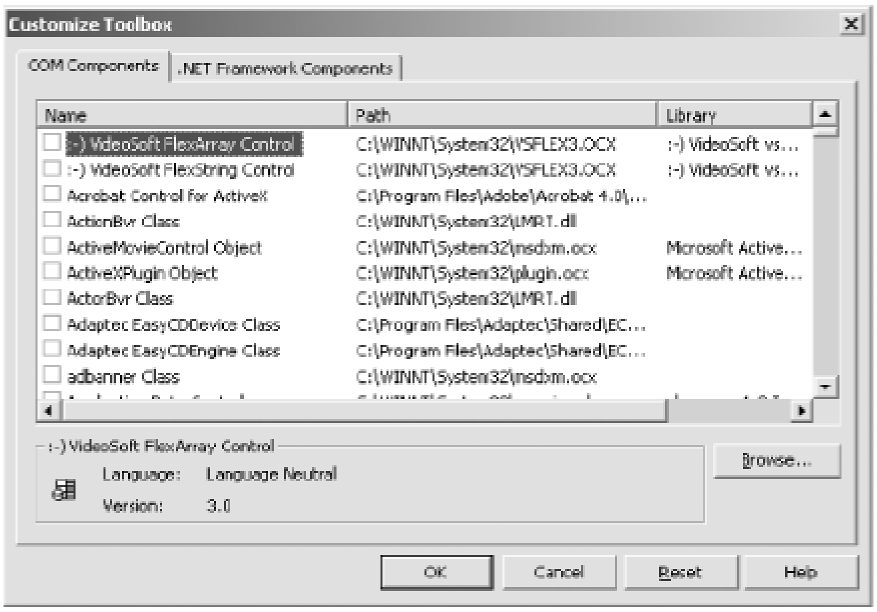
```
Imports System
Imports System.ComponentModel
Imports System.Drawing
Imports System.Windows.Forms
Public Class MyControl
Inherits UserControl
Private WithEvents lblCaption As Label
Private WithEvents txtText As TextBox
Event CaptionChanged As EventHandler
Public Sub New( )
MyBase.New( )
' Instantiate a Label object and set its properties.
lblCaption = New Label( )
With lblCaption
    .AutoSize = True
    .Dock = DockStyle.Left
    .Size = New Size(53, 13)
    .TabIndex = 0
    .Text = "lblCaption"
End With
' Instantiate a TextBox object and set its properties.
txtText = New TextBox( )
With txtText
    .Dock = DockStyle.Fill
    .Location = New Point(53, 0)
    .Size = New Size(142, 20)
    .TabIndex = 1
    .Text = "txtText"
End With
' Add the label and text box to the form's Controls collection.
Me.Controls.AddRange(New Control( ) {txtText, lblCaption})
' Set the size of the form.
Me.Size = New Size(195, 19)
End Sub
' Override the Control class's Text property. Map it to the
' constituent TextBox control's Text property.
<Category("Appearance"), _
Description("The text contained in the textbox.")> _Public Overrides Property Text(
) As String
Get
Return txtText.Text
End Get
Set(ByVal Value As String)
txtText.Text = Value
End Set
End Property
' Add a Caption property. Map it to the constituent Label
' control's Text property.
<Category("Appearance"), _
Description("The text appearing next to the textbox.")> _
Public Property _
Caption( ) As String
Get
Return lblCaption.Text
End Get
Set(ByVal Value As String)
lblCaption.Text = Value
End Set
End Property
' When the constituent TextBox control's TextChanged event is
' received, fire the user control's TextChanged event.

    Private Sub txtText_TextChanged( _ByVal sender As Object, _
    ByVal e As EventArgs _) Handles txtText.TextChanged
    Me.OnTextChanged(e)
    End Sub
' When the constituent Label control's TextChanged event is
' received, fire the user control's CaptionChanged event.

    Private Sub lblCaption_TextChanged( _ByVal sender As Object, _
    ByVal e As EventArgs _) Handles lblCaption.TextChanged
        RaiseEvent CaptionChanged(Me, EventArgs.Empty)
    End Sub
End Class
```

After compiling the code in Example 5-4, either the custom control can be added to the Visual Studio .NET toolbox and added to forms just like other controls, or it can be referenced and instantiated from an application compiled at the command line.
To add the custom control to the Visual Studio .NET toolbox:

1. Deploy the custom control's *.dll* file into the client application's *bin* directory. (The *bin* directory is a directory created by Visual Studio .NET when the client application is created.)

2. From the Visual Studio .NET menu, select Tools Customize Toolbox.
   The Customize Toolbox dialog box appears, as shown in Figure 5-30.

Figure 5-30. The Customize Toolbox dialog box



Name	Path	Library
:-) VideoSoft FlexArray Control	C:\WINNT\System32\VSFLEX3.OCX	:-) VideoSoft vs...
:-) VideoSoft FlexString Control	C:\WINNT\System32\VSFLEX3.OCX	:-) VideoSoft vs...
Acrobat Control for ActiveX	C:\Program Files\Adobe\Acrobat 4.0\...	
ActionBvr Class	C:\WINNT\System32\LMRT.dll	
ActiveMovieControl Object	C:\WINNT\System32\msdxm.ocx	Microsoft Active...
ActiveXPlugin Object	C:\WINNT\System32\plugin.ocx	Microsoft Active...
ActorBvr Class	C:\WINNT\System32\LMRT.dll	
Adaptec EasyCDDevice Class	C:\Program Files\Adaptec\Shared\EC...	
Adaptec EasyCDEngine Class	C:\Program Files\Adaptec\Shared\EC...	
adbanner Class	C:\WINNT\System32\msdxm.ocx	

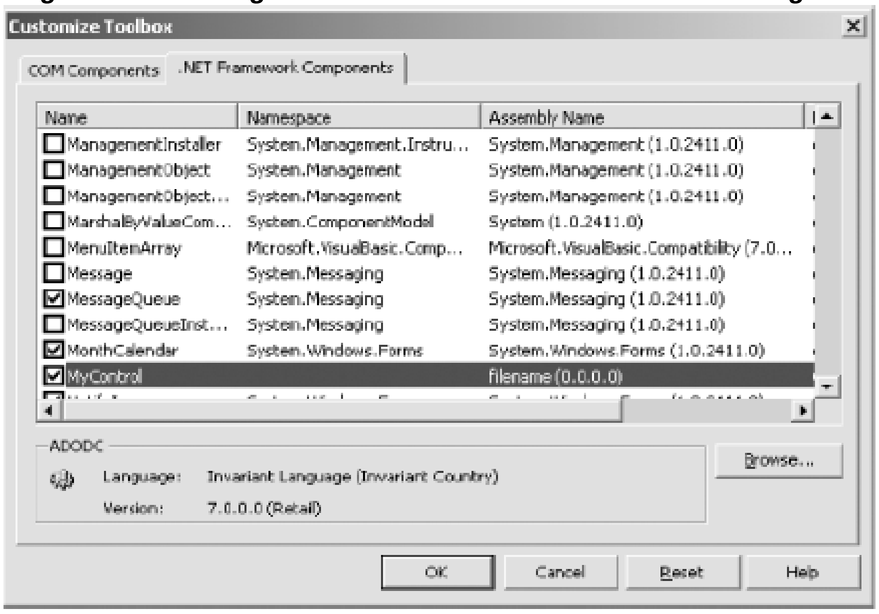
:-) VideoSoft FlexArray Control
Language: Language Neutral
Version: 3.0

3. Click the .NET Framework Components tab, then click the Browse button.

4. Browse for and select the *.dll* file compiled from the code in Example 5-4.

5. The controls in the *.dll* file are added to the Customize Toolbox dialog box, as shown in Figure 5-31. (In this case there is only one control in the *.dll* file—the MyControl control.)

Figure 5-31. Adding a control to the Customize Toolbox dialog box


Customize Toolbox

COM Components | .NET Framework Components

Name	Namespace	Assembly Name
☐ ManagementInstaller	System.Management.Instru...	System.Management (1.0.2411.0)
☐ ManagementObject	System.Management	System.Management (1.0.2411.0)
☐ ManagementObject...	System.Management	System.Management (1.0.2411.0)
☐ MarshalByValueCom...	System.ComponentModel	System (1.0.2411.0)
☐ MenuItemArray	Microsoft.VisualBasic.Comp...	Microsoft.VisualBasic.Compatibility (7.0...
☐ Message	System.Messaging	System.Messaging (1.0.2411.0)
☑ MessageQueue	System.Messaging	System.Messaging (1.0.2411.0)
☐ MessageQueueInst...	System.Messaging	System.Messaging (1.0.2411.0)
☑ MonthCalendar	System.Windows.Forms	System.Windows.Forms (1.0.2411.0)
☑ MyControl		filename (0.0.0.0)

ADODC

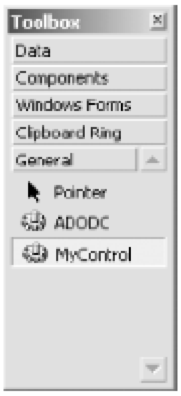
Language: Invariant Language (Invariant Country)

Version: 7.0.0.0 (Retail)

Browse... | OK | Cancel | Reset | Help

6. Ensure that a checkmark appears next to the control name, and click OK. The control should now appear on the General tab of the Toolbox window, as shown in Figure 5-32.

Figure 5-32. The Toolbox window, showing the custom control from Example 5-4



After you add the custom control to the Toolbox, the control can be added to a form just like any other control. To use the custom control from a non-Visual Studio .NET application, these steps are required:

1. Deploy the custom control's *.dll* file into the same directory as the client *.vb* file.
   the same way
2. Declare, instantiate, and use the control in client application code, in that has been done throughout this chapter for standard controls.
3. Reference the control's assembly in the compilation command.
   The code in Example 5-5 shows how to use the control. It can be compiled with this command:
   vbc MyApp.vb
   /r:System.dll,System.Drawing.dll,System.Windows.Forms.dll,MyControl.dll
   /t:winexe
   (Note that the command should be typed on a single line.)

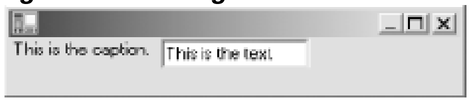
Example 5-5. Using a custom control

```
Imports System.Drawing
Imports System.Windows.Forms
    Module modMain
    <System.STAThreadAttribute( )> Public Sub Main( )
```

```
        System.Threading.Thread.CurrentThread.ApartmentState = _
        System.Threading.ApartmentState.STA
        System.Windows.Forms.Application.Run(New Form1( ))
    End Sub
    End Module
    Public Class Form1
    Inherits System.Windows.Forms.Form
    Private ctrl As New MyControl( )
        Public Sub New( )
        ctrl.Caption = "This is the caption."
        ctrl.Text = "This is the text."
        Controls.Add(ctrl)
    End Sub
    End Class
```

The resulting display is shown in Figure 5-33.

Figure 5-33. Using a custom control



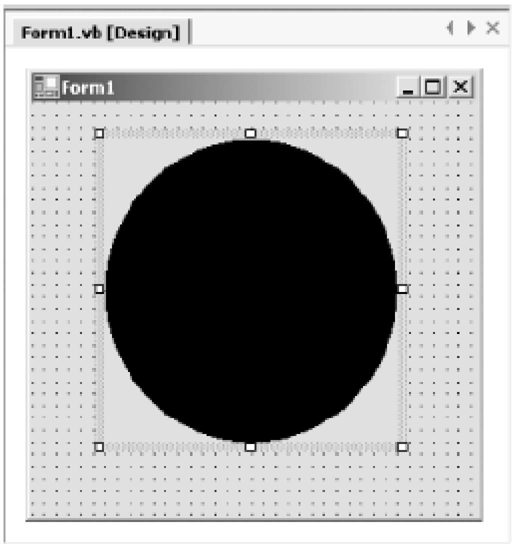
5.7.2 Building Controls That Draw Themselves

Adding constituent controls to a user control is just one way to make a custom control. Another way is to draw the user interface of the control directly onto the control's surface. Example 5-6 shows the definition of a control that draws its own (albeit simple) user interface. Figure 5-34 shows the control after it has been placed on a form in design mode in the Windows Forms Designer.

Example 5-6. A control that renders itself

```
    Public Class MyControl
        Inherits UserControl

        Protected Overrides Sub OnPaint(ByVal e As PaintEventArgs)
            e.Graphics.FillEllipse(New SolidBrush(Me.ForeColor), _
            Me.ClientRectangle)
            End Sub

        Public Sub New( )
            Me.ResizeRedraw = True
        End Sub
    End Class
```

Figure 5-34. The MyControl control as it appears on a form in the Windows Forms Designer



The control in this example overrides the OnPaint method declared in the Control class. Windows invokes the OnPaint method whenever the control needs repainting. The PaintEventArgs object passed to the OnPaint method provides information useful within the OnPaint method. The PaintEventArgs type was discussed in detail in Chapter 4 under Section 4.6. The OnPaint method in Example 5-6 draws an ellipse sized to fill the client area of the control.Setting the control's ResizeRedraw property to True causes the OnPaint method to be called whenever the control is resized. Because the appearance of the control in Example 5-6 depends on the size of the control, the control's constructor sets the ResizeRedraw property to True. When ResizeRedraw is False (the default), resizing the control does not cause the OnPaint method to be called. Although the code in Example 5-6 was built by hand, the OnPaint method can also be added (by hand) to the code built by the Windows Forms Designer. In addition, the techniques of using constituent controls and drawing directly on the user control can both be used within the same user control.

5.7.3 Building Nonrectangular Controls

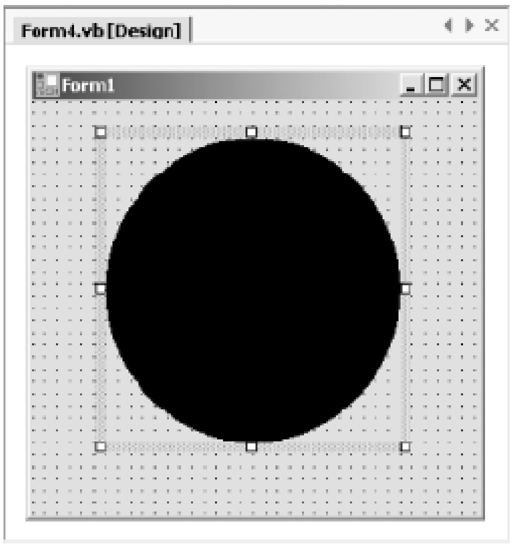
User controls are rectangular by default, but controls having other shapes can be made by setting the control's Region property. The Region property accepts a value of type Region (defined in the
System.Drawing namespace). Objects of type Region define complex areas and are commonly used for window clipping. Consider again Example 5-6 and Figure 5-34. Notice in Figure 5-34 that the grid dots on the form do not show through the background of the control. Example 5-7 shows how to clip the area of the control to match the ellipse being drawn in the OnPaint method. It is based on Example 5-6, with new code shown in bold.

Example 5-7. Clipping the area of a control

```
' Assumes Imports System.Drawing.Drawing2D
Public Class MyControl
Inherits UserControl
    Private Function CreateRegion( ) As Region
    Dim gp As New GraphicsPath( )
    gp.AddEllipse(Me.ClientRectangle)
    Dim rgn As New Region(gp)
    Return rgn
    End Function
    Protected Overrides Sub OnResize(ByVal e As EventArgs)
    Me.Region = Me.CreateRegion( )
End Sub
Protected Overrides Sub OnPaint(ByVal e As PaintEventArgs)
e.Graphics.FillEllipse(New SolidBrush(Me.ForeColor), _
Me.ClientRectangle)

End Sub

    Public Sub New( )
        Me.ResizeRedraw = True
        End Sub
    End Class
```

The CreateRegion method in Example 5-7 creates a region in the shape of an ellipse sized to fill the client area of the control. To create nonrectangular regions, you must instantiate a GraphicsPath object, use the drawing methods of the GraphicsPath class to define a complex shape within the GraphicsPath object, and then instantiate a Region object, initializing it from the GraphicsPath object. Example 5-7 calls the GraphicsPath class's AddEllipse method to create an ellipse within the GraphicsPath object. Additional methods could be called to add more shapes (including line-drawn shapes) to the GraphicsPath object. The OnResize method in Example 5-7 ensures that the control's Region property is reset every time the control is resized. Figure 5-35 shows the control after it has been placed on a form in design mode in the Windows Forms Designer. Note that the grid dots now show through the corners of the control. This clipped area is no longer considered part of the control. If the user clicks on this area, the click passes through to the object underneath the control (in this case, the form).

Figure 5-35. A control with its Region property set to clip everything outside of the ellipse



5.8 COM INTEROPERABILITY

5.8.1 Interoperability

The Component Object Model (COM) allows an object to expose its functionality to other components and to host applications. While COM objects have been fundamental to Windows programming for many years, applications designed for the common language runtime (CLR) used by the .NET Platform offer many advantages.

.NET Platform applications will eventually replace those developed with COM. Until then, you may need to use or create COM objects with Visual Studio .NET. Interoperability with COM, or COM interop, enables you to use existing COM objects while transitioning to the .NET Platform at your own pace.

5.8.1.1 Managed Code and Data

Code developed for the .NET Platform is referred to as managed code, and contains metadata that is used by the common language runtime (CLR). Data used by .NET applications is called managed data because the .NET runtime manages data-related tasks such as allocating and reclaiming memory, and type checking. By default, Visual Basic .NET uses managed code and data, but you can access the unmanaged code and data of COM objects using interop assemblies.

5.8.1.2 Assemblies

An assembly is the primary building block of a .NET Framework application. It is a collection of functionality that is built, versioned, and deployed as a single implementation unit containing one or more files. Each assembly contains an assembly manifest.

5.8.1.3 Type Libraries and Assembly Manifests

Type libraries describe characteristics of COM objects, such as member names and data types. Assembly manifests perform the same function for .NET applications. They include information about:

- Assembly identity, version, culture, and digital signature
- Files that make up the assembly implementation
- Types and resources that make up the assembly, including those that are exported from it
- Compile-time dependencies on other assemblies
- Permissions required for the assembly to run properly

5.8.1.4 Importing and Exporting Type Libraries

Visual Studio .NET contains a utility, Tlbimp, that lets you import information from a type library into a .NET application. You can generate type libraries from assemblies by using the Tlbexp utility.

For information on Tlbimp and Tlbexp, see Type Library Importer (Tlbimp.exe) and Type Library Exporter (Tlbexp.exe).

5.8.1.5 Interop Assemblies

Interop assemblies are .NET assemblies that act as a bridge between managed and unmanaged code, mapping COM object members to equivalent .NET managed members. Interop assemblies created by Visual Basic .NET handle many of the details of working with COM objects, such as interoperability marshaling.

5.8.1.6 Interoperability Marshaling

All .NET applications share a set of common types that allow interoperability of objects, regardless of the programming language used. The parameters and return values of COM objects sometimes use data types that are different than those used in managed code. Interoperability marshaling is the process of packaging parameters and return values into equivalent data types as they move to and from COM objects. For more information.

5.8.2 COM in .Net

In Visual Basic .NET, adding references to COM objects that have type libraries is similar to doing so in previous versions of Visual Basic. However, Visual Basic .NET adds the creation of an interop assembly to the procedure. References to the members of the COM object are routed to the interop assembly and then forwarded to the actual COM object. Responses from the COM object are routed to the interop assembly and forwarded to your .NET application.

5.8.3 To add references to COM objects

1. On the Project menu, select Add Reference and then click the COM tab.
2. Select the component you want to use from the list of COM objects.
3. To simplify access to the interop assembly, add an Imports statement to the top of the class or module in which you will use the COM object.

5.5- 5.8 Check Your Progress
Fill in the blanks
1. The ………………. Component displays a dialog box that allows the user to choose a Colour.
2. The …………….. provides three components for creating and managing menus.
3. COM means ………………………….. Model.
4. Before using any pre written code we have to add reference of …………………………….. in our assembly.

5.9 SUMMARY

The Windows Forms architecture is broad and deep. This chapter has presented everything you need to know to get started developing GUI desktop applications, but there is more power waiting for you after you assimilate what's here. Once you master the material in this chapter, you can write complex GUI applications. You'll also be able to tackle and understand additional types and functionality documented in the online .NET reference material.

5.10 CHECK YOUR PROGRESS - *ANSWERS*

5.1 - 5.4
1. Components 2. ImageList 3. Anchor4. z-order.

5.5 - 5.7
1. ColorDialog 2. Windows Forms library 3. Component Object Model
4. DLL

5.11 QUESTIONS FOR SELF-STUDY

1. Explain ListBox. ObjectCollection Class?
2. Write short note on Splitter control?
3. Write short note on common dialog box?
4. Explain Cloning Menu?
5. describe steps to add reference of COM in your program.

5.12 SUGGESTED READINGS

1. Visual Basic .NET Black Book by Steven Holzner

References

Programming Visual Basic .NET by Dave Grundgeiger

NOTES

CHAPTER 6

ADO.NET:DEVELOPING DATABASE APPLICATIONS

6.0 OBJECTIVES

After studying The. NET Framework chapter you will able to :

- Describe universal Data Access
- State managed Providers for database connectivity
- Write code for connecting to a SQL Server Database with authentication
- Connecting to an OLE DB DataSource
- Write code for reading Data into a Dataset in your application
- State relations between Data tables in a Data Set.
- Describe the Dataset's XML capabilities
- Explain binding a Dataset to a windows forms Data Grid.
- State binding a Dataset to a Web Forms Data Grid,typed datasets,Reading Data using a data-reader
- Execute store procedures through a sqlCommand object.

6.1 INTRODUCTION

Many software applications benefit from storing their data in database management systems. A database management system is a software component that performs the task of storing and retrieving large amounts of data. Examples of database management systems are Microsoft SQL Server and Oracle Corporation's Oracle. Microsoft SQL Server and Microsoft Access both include a sample database called Northwind. The Northwind database is used in the examples throughout this chapter. All examples in this chapter assume that the following declaration appears in the same file as the code:

```
Imports System.Data
```

Examples that use SQL Server also assume this declaration:

```
Imports System.Data.SqlClient
```

and examples that use Access assume this declaration:
Imports System.Data.OleDb

6.2 A BRIEF HISTORY OF UNIVERSAL DATA ACCESS

Database management systems provide APIs that allow application programmers to create and access databases. The set of APIs that each manufacturer's system supplies is unique to that manufacturer. Microsoft has long recognized that it is inefficient and error prone for an applications programmer to attempt to master and use all the APIs for the various available database management systems. What's more, if a new database management system is released, an existing application can't make use of it without being rewritten to understand the new APIs. What is needed is a common database API. Microsoft's previous steps in this direction included Open Database Connectivity (ODBC), OLE DB, and ADO (not to be confused with ADO.NET). Microsoft has made improvements with each new technology. With .NET, Microsoft has released a new mechanism for accessing data: ADO.NET. The name is a carryover from Microsoft's ADO (ActiveX Data Objects) technology, but it no longer stands for ActiveX

Data Objects—it's just ADO.NET. To avoid confusion, I will refer to ADO.NET as ADO.NET and to ADO as *classic* ADO. If you're familiar with classic ADO, be careful—ADO.NET is not a descendant, it's a new technology. In order to support the Internet evolution, ADO.NET is highly focused on disconnected data and on the ability for anything to be a source of data. While you will find many concepts in ADO.NET to be similar to concepts in classic ADO, it is not the same.

6.3 MANAGED PROVIDERS

When speaking of data access, it's useful to distinguish between providers of data and consumers of data. A *data provider* encapsulates data and provides access to it in a generic way. The data itself can be in any form or location. For example, the data may be in a typical database management system such as SQL Server, or it may be distributed around the world and accessed via web services. The data provider shields the data consumer from having to know how to reach the data. In ADO.NET, data providers are referred to as ***managed providers*.**
A *data consumer* is an application that uses the services of a data provider for the purposes of storing, retrieving, and manipulating data. A customer-service application that manipulates a customer database is a typical example of a data consumer. To consume data, the application must know how to access one or more data providers.
ADO.NET is comprised of many classes, but five take center stage:

Connection
Represents a connection to a data source.

Command
Represents a query or a command that is to be executed by a data source.

DataSet
Represents data. The DataSet can be filled either from a data source (using a DataAdapter object) or dynamically.

DataAdapter
Used for filling a DataSet from a data source.

DataReader
Used for fast, efficient, forward-only reading of a data source.
With the exception of DataSet, these five names are not the actual classes used for accessing data sources. Each managed provider exposes classes specific to that provider. For example, the SQL Server managed provider exposes the SqlConnection, SqlCommand, SqlDataAdapter, and SqlDataReader classes. The DataSet class is used with all managed providers. Any data-source vendor can write a managed provider to make that data source available to ADO.NET data consumers. Microsoft has supplied two managed providers in the **.NET Framework:**

SQL Server and OLE DB. The examples in this chapter are coded against the SQL Server managed provider, for two reasons. The first is that I believe that most programmers writing data access code in Visual Basic .NET will be doing so against a SQL Server database. Second, the information about the SQL Server managed provider is easily transferable to any other managed provider.

6.4 CONNECTING TO A SQL SERVER DATABASE

To read and write information to and from a SQL Server database, it is necessary first to establish a connection to the database. This is done with the SqlConnection object, found in the System.Data.SqlClient namespace. Here's an example:

```
' Open a database connection.
Dim strConnection As String = _
"Data Source=localhost;Initial Catalog=Northwind;" _
& "Integrated Security=True"
Dim cn As SqlConnection = New SqlConnection(strConnection)
cn.Open( )
```

This code fragment instantiates an object of type SqlConnection, passing its constructor a connection string. Calling the SqlConnection object's Open method opens the connection. A connection must be open for data to be read or written, or for commands to be executed. When you're finished accessing the database, use the Close method to close the connection:

```
'Close the database connection.
cn.Close( )
```

The connection string argument to the SqlConnection class's constructor provides information that allows the SqlConnection object to find the SQL Server database. The connection string shown in the earlier code fragment indicates that the database is located on the same machine that is running the code snippet (Data Source=localhost), that the database name is Northwind (Initial Catalog=Northwind), and that the user ID that should be used for logging in to SQL Server is the current Windows login account (Integrated Security=True). Table 6-1 shows the valid SQL Server connection string settings.

Table 6-1. SQL Server connection string settings**Setting Default Value Description**

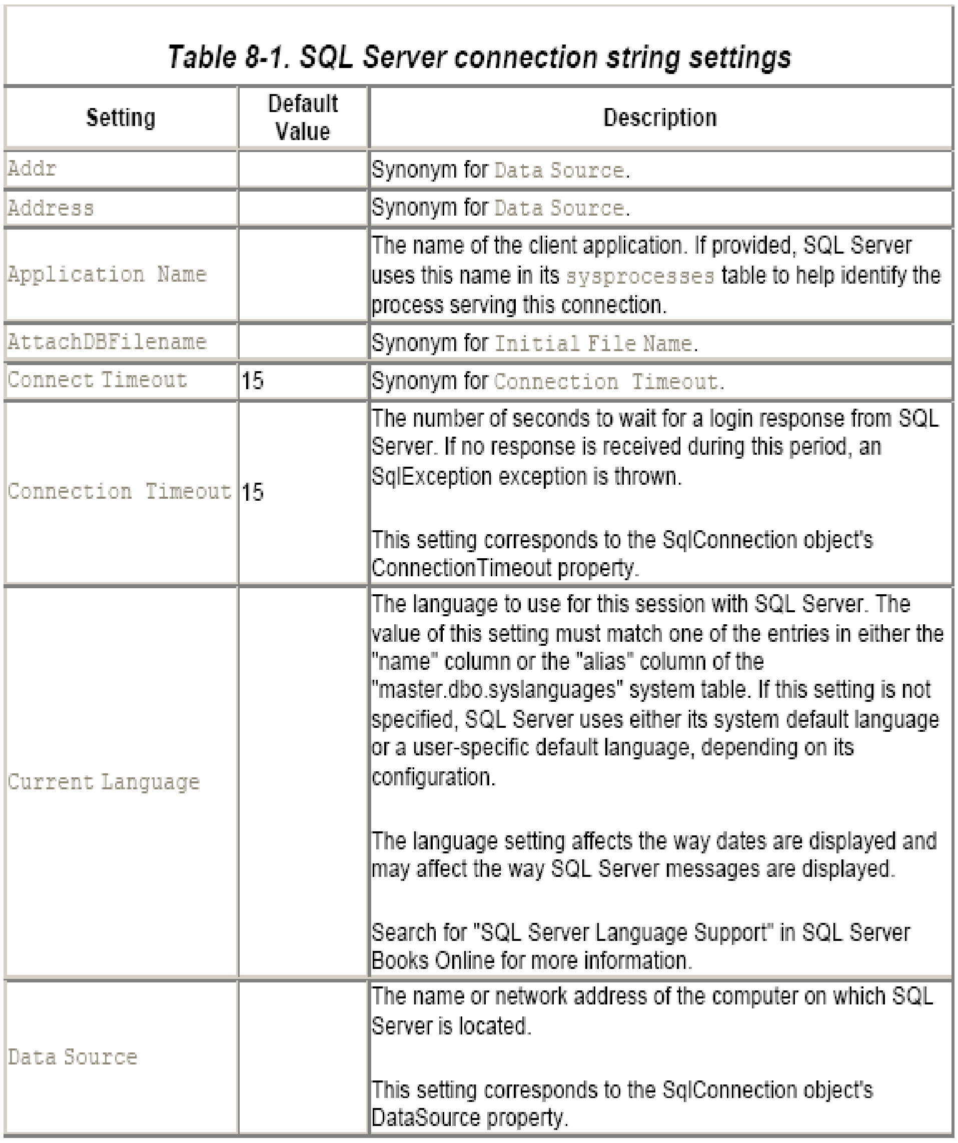
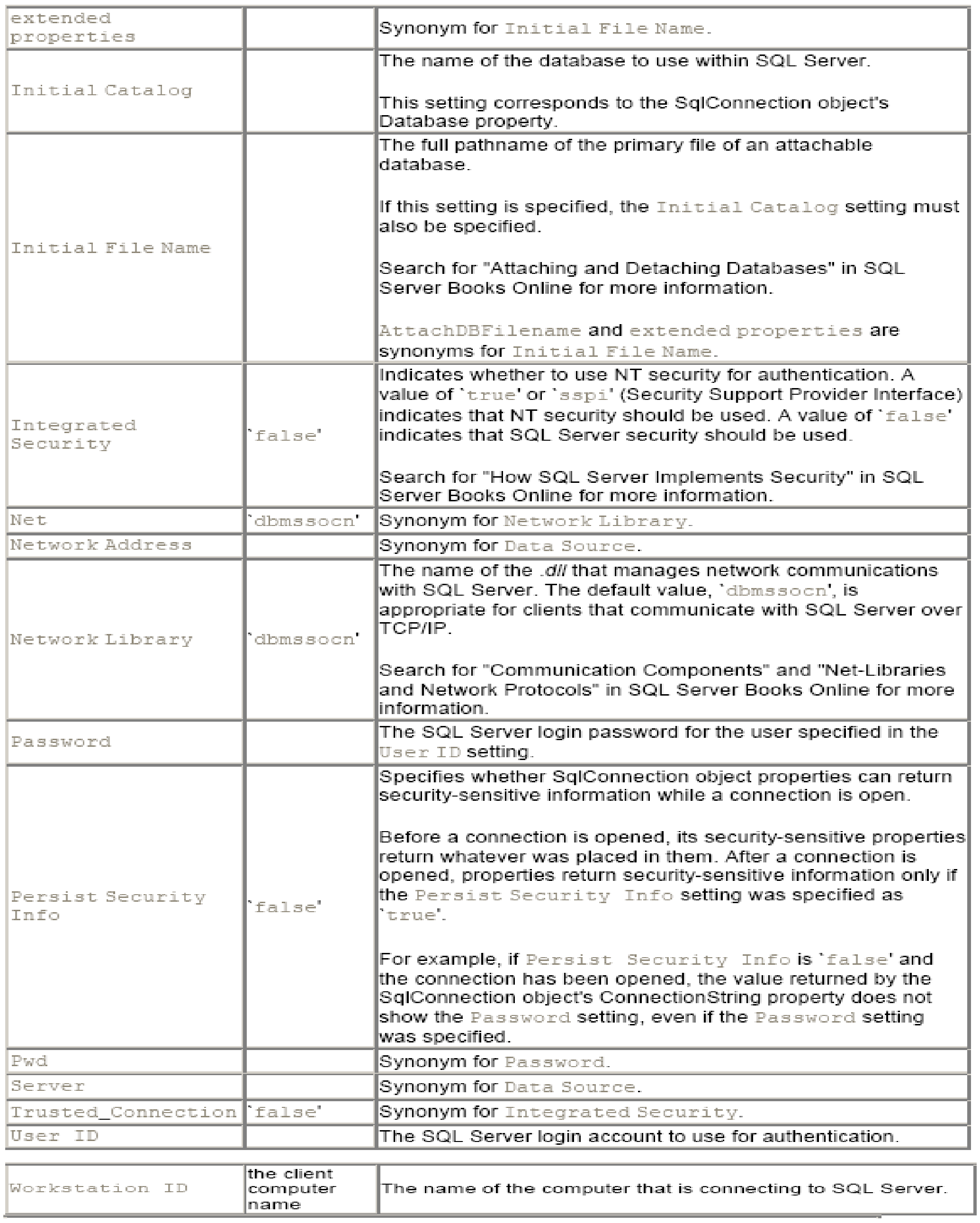


Table 8-1. SQL Server connection string settings

Setting	Default Value	Description
`Addr`		Synonym for `Data Source`.
`Address`		Synonym for `Data Source`.
`Application Name`		The name of the client application. If provided, SQL Server uses this name in its `sysprocesses` table to help identify the process serving this connection.
`AttachDBFilename`		Synonym for `Initial File Name`.
`Connect Timeout`	15	Synonym for `Connection Timeout`.
`Connection Timeout`	15	The number of seconds to wait for a login response from SQL Server. If no response is received during this period, an SqlException exception is thrown. This setting corresponds to the SqlConnection object's ConnectionTimeout property.
`Current Language`		The language to use for this session with SQL Server. The value of this setting must match one of the entries in either the "name" column or the "alias" column of the "master.dbo.syslanguages" system table. If this setting is not specified, SQL Server uses either its system default language or a user-specific default language, depending on its configuration. The language setting affects the way dates are displayed and may affect the way SQL Server messages are displayed. Search for "SQL Server Language Support" in SQL Server Books Online for more information.
`Data Source`		The name or network address of the computer on which SQL Server is located. This setting corresponds to the SqlConnection object's DataSource property.



extended properties		Synonym for Initial File Name.
Initial Catalog		The name of the database to use within SQL Server. This setting corresponds to the SqlConnection object's Database property.
Initial File Name		The full pathname of the primary file of an attachable database. If this setting is specified, the Initial Catalog setting must also be specified. Search for "Attaching and Detaching Databases" in SQL Server Books Online for more information. AttachDBFilename and extended properties are synonyms for Initial File Name.
Integrated Security	`false'	Indicates whether to use NT security for authentication. A value of `true' or `sspi' (Security Support Provider Interface) indicates that NT security should be used. A value of `false' indicates that SQL Server security should be used. Search for "How SQL Server Implements Security" in SQL Server Books Online for more information.
Net	`dbmssocn'	Synonym for Network Library.
Network Address		Synonym for Data Source.
Network Library	`dbmssocn'	The name of the *.dll* that manages network communications with SQL Server. The default value, `dbmssocn', is appropriate for clients that communicate with SQL Server over TCP/IP. Search for "Communication Components" and "Net-Libraries and Network Protocols" in SQL Server Books Online for more information.
Password		The SQL Server login password for the user specified in the User ID setting.
Persist Security Info	`false'	Specifies whether SqlConnection object properties can return security-sensitive information while a connection is open. Before a connection is opened, its security-sensitive properties return whatever was placed in them. After a connection is opened, properties return security-sensitive information only if the Persist Security Info setting was specified as `true'.  For example, if Persist Security Info is `false' and the connection has been opened, the value returned by the SqlConnection object's ConnectionString property does not show the Password setting, even if the Password setting was specified.
Pwd		Synonym for Password.
Server		Synonym for Data Source.
Trusted_Connection	`false'	Synonym for Integrated Security.
User ID		The SQL Server login account to use for authentication.
Workstation ID	the client computer name	The name of the computer that is connecting to SQL Server.

SQL Server Authentication

Before a process can access data that is located in a SQL Server database, it must log in to SQL Server. The SqlConnection object communicates with SQL Server and performs this login based on information provided in the connection string. Logging in requires *authentication*. Authentication means proving to SQL Server that the process is acting on behalf of a user who is
authorized to access SQL Server data. SQL Server recognizes two methods of authentication:

- SQL Server Authentication, which requires the process to supply a username and password that have been set up in SQL Server by an administrator. Beginning with SQL Server 2000, this method of authentication is no longer recommended (and is disabled by default).
- Integrated Windows Authentication, in which no username and password are provided. Instead, the Windows NT or Windows 2000 system on which the process is running communicates the user's Windows login name to SQL Server. The Windows user must be set up in SQL Server by an administrator in order for this to work.

To use SQL Server Authentication:

1. (SQL Server 2000 only) Enable SQL Server Authentication. In Enterprise Manager, right-click on the desired server, click Properties, and then click the Security tab. Select "SQL Server and Windows" and click OK.
2. The network administrator sets up a login account using Enterprise Manager, specifying that the account will use SQL Server Authentication and supplying a password. Programming books (including this one) typically assume the presence of a user named "sa" with an empty password, because this is the default system administrator account set up on every SQL Server installation (good administrators change the password, however).
3. The network administrator assigns rights to this login account as appropriate.
4. The data access code specifies the account and password in the connection string passed to the SqlConnection object. For example, the following connection string specifies the "sa" account with a blank password:

   "Data Source=SomeMachine; Initial Catalog=Northwind; User ID=sa; Password="

   To use Integrated Windows Authentication:

1. The network administrator sets up the login account using Enterprise Manager, specifying that the account will use Windows Authentication and supplying the Windows user or group that is to be given access.
2. The network administrator assigns rights to this login account as appropriate.
3. The data access code indicates in the connection string that Integrated Windows Security should be used, as shown here:

"Data Source= SomeMachine; Initial Catalog=Northwind; Integrated Security=True" When using Integrated Windows Authentication, it is necessary to know what Windows login account a process will run under and to set up appropriate rights for that login account in SQL Server Enterprise Manager. A program running on a local machine generally runs under the login account of the user that started the program. A component running in Microsoft Transaction Server (MTS) or COM+ runs under a login account specified in the MTS or COM+ Explorer. Code that is embedded in an ASP.NET web page runs under a login account specified in Internet Information Server (IIS). Consult the documentation for these products for information on specifying the login account under which components run. Consult the SQL Server Books Online for information on setting up SQL Server login accounts and on specifying account privileges.

6.5 CONNECTING TO AN OLE DB DATA SOURCE

OLE DB is a specification for wrapping data sources in a COM-based API so that data sources can be accessed in a polymorphic way. The concept is the same as ADO.NET's concept of managed providers. OLE DB predates ADO.NET and will eventually be superseded by it. However, over the years, OLE DB providers have been written for many data sources, including Oracle, Microsoft Access, Microsoft Exchange, and others, whereas currently only one product—SQL Server—is natively supported by an ADO.NET managed provider. To provide immediate support in ADO.NET for a wide range of data sources, Microsoft has supplied an ADO.NET managed provider for OLE DB. That means that ADO.NET can work with any data source for which there is an OLE DB data provider. Furthermore, because there is an OLE DB provider that wraps ODBC (an even older data-access technology), ADO.NET can work with virtually all legacy data, regardless of the source. Connecting to an OLE DB data source is similar to connecting to SQL Server, with a few differences: the OleDbConnection class (from the System.Data.OleDb namespace) is used instead of the SqlConnection class, and the connection string is slightly different. When using the OleDbConnection class, the connection string must specify the OLE DB provider that is to be used as well as additional information that tells the OLE DB provider where the actual data is. For example, the following code opens a connection to the Northwind sample database in Microsoft Access:

```
'Open a connection to the database.
Dim strConnection As String =
"Provider=Microsoft.Jet.OLEDB.4.0;DataSource="_&"C:\Program    Files\Microsoft
Office\Office\Samples\Northwind.mdb"
```

```
Dim cn As OleDbConnection = New OleDbConnection(strConnection)
cn.Open( )
```

Similarly, this code opens a connection to an Oracle database:

```
' Open a connection to the database.
Dim strConnection As String=_"Provider=MSDAORA.1;User ID=MyID;Password=MyPassword;"_&"DataSource=MyDatabaseService.MyDomain.com"
Dim cn As OleDbConnection = New OleDbConnection(strConnection)
cn.Open( )
```

The values of each setting in the connection string, and even the set of settings that are allowed in the connection string, are dependent on the specific OLE DB provider being used. Refer to the documentation for the specific OLE DB provider for more information.
Table 6-2 shows the provider names for several of the most common OLE DB providers.

Table 6-2. Common OLE DB provider names **Data source OLE DB provider name**

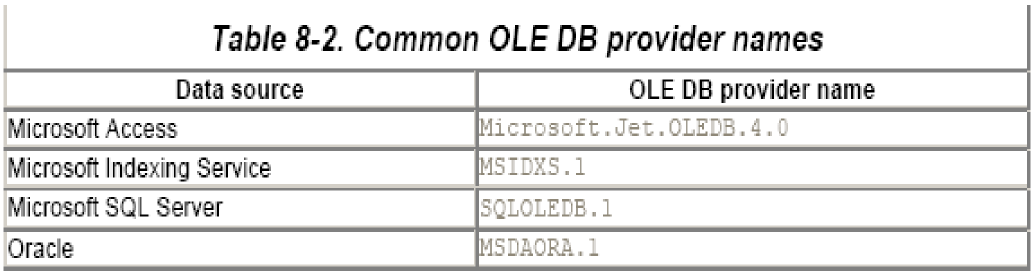


Table 8-2. Common OLE DB provider names

Data source	OLE DB provider name
Microsoft Access	Microsoft.Jet.OLEDB.4.0
Microsoft Indexing Service	MSIDXS.1
Microsoft SQL Server	SQLOLEDB.1
Oracle	MSDAORA.1

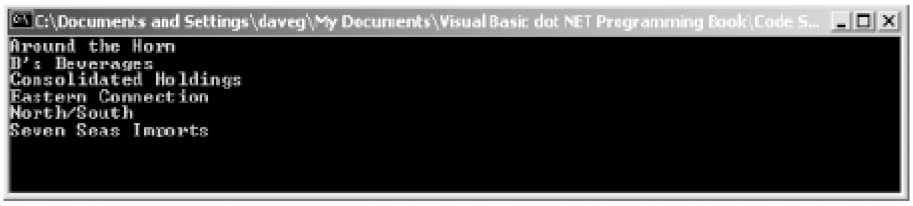
6.1 - 6.5 Check Your Progress

1. A ………….……. is an application that uses the services of a data provider for the purpose of storing retrieving and manipulating data.
2. The ………………………. object communicates with SQL Server.
3. The …………………………… class encapsulates a set of tables and the relations between those tables.

6.6 READING DATA INTO A DATASET

The DataSet class is ADO.NET's highly flexible, general-purpose mechanism for reading and updating data. Example 6-1 shows how to issue a SQL SELECT statement against the SQL Server Northwind sample database to retrieve and display the names of companies located in London. The resulting display is shown in Figure 6-1.

Figure 6-1. The output generated by the code in Example 6-1


C:\Documents and Settings\daveg\My Documents\Visual Basic dot NET Programming Book\Code S...

```
Around the Horn
B's Beverages
Consolidated Holdings
Eastern Connection
North/South
Seven Seas Imports
```

Example 6-1. Retrieving data from SQL Server using a SQL SELECT

Statement

```
' Open a connection to the database.
Dim strConnection As String = _
"Data Source=localhost; Initial Catalog=Northwind;" _
& "Integrated Security=True"
Dim cn As SqlConnection = New SqlConnection(strConnection)
cn.Open( )
' Set up a data set command object.
Dim strSelect As String = "SELECT * FROM Customers WHERE City = 'London'"
Dim dscmd As New SqlDataAdapter(strSelect, cn)
' Load a data set.
Dim ds As New DataSet( )
dscmd.Fill(ds, "LondonCustomers")
' Close the connection.
cn.Close( )
' Do something with the data set.
Dim dt As DataTable = ds.Tables.Item("LondonCustomers")
Dim rowCustomer As DataRow
For Each rowCustomer In dt.Rows
Console.WriteLine(rowCustomer.Item("CompanyName"))
Next
```

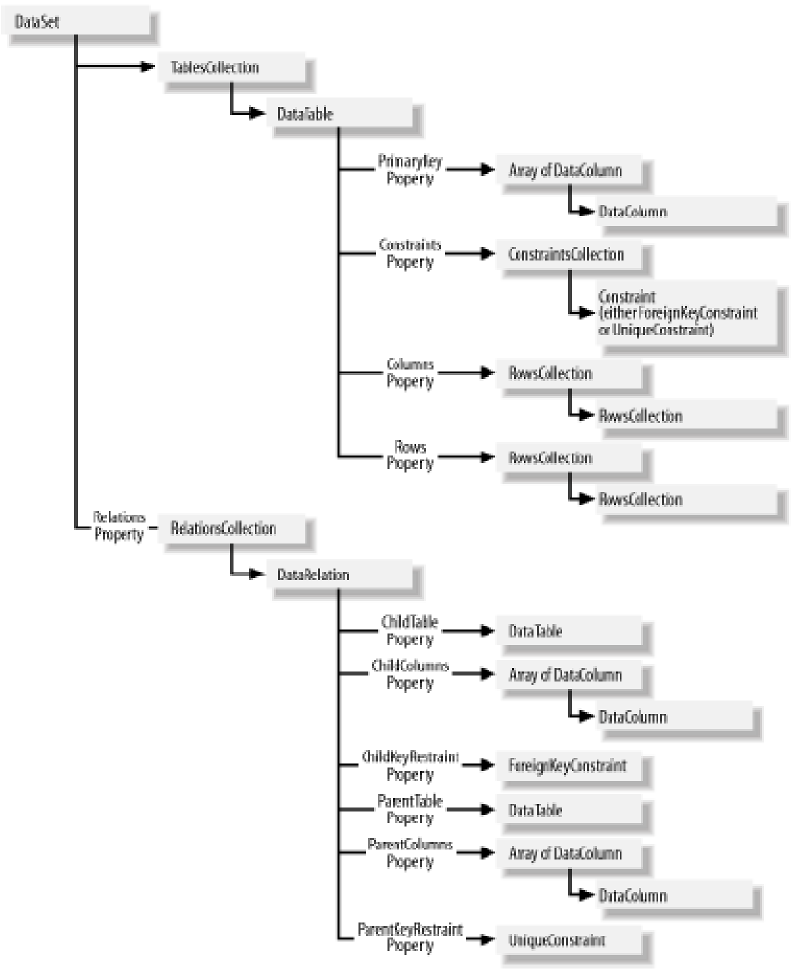
The code in **Example 6-1** performs the following steps to obtain data from the database:

1. Opens a connection to the database using a SqlConnection object.
2. Instantiates an object of type SqlDataAdapter in preparation for filling a DataSet object. In Example 6-1, a SQL SELECT command string and a Connection object are passed to the SqlDataAdapter object's constructor.
3. Instantiates an object of type DataSet and fills it by calling the SqlDataAdapter object's Fill method.

6.6.1 The DataSet Class

The DataSet class encapsulates a set of tables and the relations between those tables. Figure 6-2 shows a class model diagram containing the DataSet and related classes. The DataSet is always completely disconnected from any data source. In fact, the DataSet has no knowledge of the source of its tables and relations. They may be dynamically created using methods on the DataSet, or they may be loaded from a data source. In the case of the SQL Server managed provider, a DataSet can be loaded from a SQL Server database using an SqlDataAdapter object. This is what was done in Example 6-1.

Figure 6-2. A class model diagram for the DataSet and related classes



After a DataSet is loaded, its data can be changed, added to, or deleted, all without affecting the data source. Indeed, a database connection does not need to be maintained during these updates. When ready, the updates can be written back to the database by establishing a new connection and calling the SqlDataAdapter object's Update method. Examples of writing updates to a database are shown later in this chapter.Navigating the DataSet In this section you'll learn how to find specific data in a DataSet object, how to make changes to that data, and how to write those changes back to a database.

6.6.2 Finding Tables

The DataSet object's Tables property holds a TablesCollection object that contains the DataTable objects in the DataSet. The following code loops through all the tables in the DataSet and displays their names:

```
' Iterate through the tables in the DataSet ds.
Dim dt As DataTable
    For Each dt In ds.Tables
        Console.WriteLine(dt.TableName)
    Next
```

This code does the same thing, using a numeric index on the TablesCollection object:

```
' Iterate through the tables in the DataSet ds.
Dim n As Integer
    For n = 0 To ds.Tables.Count - 1
        Console.WriteLine(ds.Tables(n).TableName)
    Next
```

The TablesCollection object can also be indexed by table name. For example, if the DataSet ds
contains a table named "Categories", this code gets a reference to it:
Dim dt As DataTable = ds.Tables("Categories")

6.6.3 Finding Rows

The DataTable object's Rows property holds a DataRowCollection object that in turn holds the table's DataRow objects. Each DataRow object holds the data for that particular row. The following code loops through all the rows in the DataTable and displays the value of the first column (column 0) in the row:

```
' Iterate through the rows.
Dim row As DataRow
    For Each row In dt.Rows
        Console.WriteLine(row(0))
    Next
```

This code does the same thing, using a numeric index on the RowsCollection object:

```
' Iterate through the rows.
Dim n As Integer
    For n = 0 To dt.Rows.Count - 1
        Console.WriteLine(dt.Rows(n)(0))
    Next
```

To assist with locating specific rows within a table, the DataTable class provides a method called Select. The Select method returns an array containing all the rows in the table that match the given criteria. The syntax of the Select method is: Public Overloads Function Select(_
ByVal *filterExpression* As String, _ByVal *sort* As String, _ByVal *recordStates* As System.Data.DataViewRowState_)As System.Data.DataRow( )

The parameters of the Select method are:

filterExpression
This parameter gives the criteria for selecting rows. It is a string that is in the same format as the WHERE clause in an SQL statement.

sort
This parameter specifies how the returned rows are to be sorted. It is a string that is in the same format as the ORDER BY clause in an SQL statement.

recordStates
This parameter specifies the versions of the records that are to be retrieved. Record versions are discussed in Section 6.5.6. The value passed in this parameter must be one of the values given by the System.Data.DataViewRowState enumeration. Its values are:

CurrentRows
Returns the current version of each row, regardless of whether it is unchanged, new, or modified.

Deleted
Returns only rows that have been deleted.

ModifiedCurrent
Returns only rows that have been modified. The values in the returned rows are the current values of the rows.

ModifiedOriginal
Returns only rows that have been modified. The values in the returned rows are the original values of the rows.

New
Returns only new rows.

None
Returns no rows.

OriginalRows
Returns only rows that were in the table prior to any modifications. The values in the returned rows are the original values.

Unchanged
Returns only unchanged rows.
These values can be combined using the And operator to achieve combined results. For example, to retrieve both modified and new rows, pass this value:
DataViewRowState.ModifiedCurrent And DataViewRowState.New
The return value of the Select method is an array of DataRow objects.
The Select method is overloaded. It has a two-parameter version that is the same as the full version, except that it does not take a *recordStates* parameter:

Public Overloads Function Select(_ByVal *filterExpression* As String, _ByVal *sort* As String _) As System.Data.DataRow( )
Calling this version of the Select method is the same as calling the full version with a

recordStates
value of DataViewRowState.CurrentRows.
Similarly, there is a one-parameter version that takes only a *filterExpression*:

Public Overloads Function Select(_ByVal *filterExpression* As String _) As System.Data.DataRow( )

This is the same as calling the three-parameter version with *sort* equal to "" (the empty string) and *recordStates* equal to DataViewRowState.CurrentRows.Lastly, there is the parameterless version of Select:

Public Overloads Function Select( ) As System.Data.DataRow( )
This is the same as calling the three-parameter version with *filterExpression* and *sort* equal to "" (the empty string) and *recordStates* equal to

DataViewRowState.CurrentRows. As an example of using the Select method, this line of code returns all rows whose Country column contains the value "Mexico":

```
Dim rows( ) As DataRow = dt.Select("Country = 'Mexico'")
```

Because the *sort* and *recordStates* parameters were not specified, they default to "" (the empty string) and DataViewRowState.CurrentRows, respectively.

6.6.3.1 The Select method versus the SQL SELECT statement

If an application is communicating with a database over a fast, persistent connection, it is more efficient to issue SQL SELECT statements that load the DataSet with only the desired records, rather than to load the DataSet with a large amount of data and then pare it down with the DataTable's Select method. The Select method is useful for distributed applications that might not have a fast connection to the database. Such an application might load a large amount of data from the database into a DataSet object, then use several calls to the DataTable's Select method to locally view and process the data in a variety of ways. This is more efficient in this case because the data is moved across the slow connection only once, rather than once for each query.

6.6.4 Finding Column Values

The DataRow class has an Item property that provides access to the value in each column of a row. For example, this code iterates through all the columns of a row, displaying the value from each column (assume that row holds a reference to a DataRow object):

```
' Iterate through the column values.

Dim n As Integer
    For n = 0 To row.Table.Columns.Count - 1
        Console.WriteLine(row(n))
    Next
```

Note the expression used to find the number of columns: row.Table.Columns.Count. The DataRow object's Table property holds a reference to the DataTable object of which the row is a part. As will be discussed shortly, the Table object's Columns property maintains a collection of column definitions for the table. The Count property of this collection gives the number of columns in the table and therefore in each row.The DataRow object's Item property is overloaded to allow a specific column value to be accessed by column name. The following code assumes that the DataRow row contains a column named "Description". The code displays the value of this column in this row:Console.WriteLine(row("Description"))

6.6.5 Finding Column Definitions

The DataTable object's Columns property holds a ColumnsCollection object that in turn holds the definitions for the columns in the table. The following code iterates through the columns in the table and displays their names:

```
' Iterate through the columns.
    Dim column As DataColumn
        For Each column In dt.Columns
            Console.WriteLine(column.ColumnName)
        Next
```

This code does the same thing, using a numeric index on the ColumnsCollection object:

```
' Iterate through the columns.
Dim n As Integer
    For n = 0 To dt.Columns.Count - 1
        Console.WriteLine(dt.Columns(n).ColumnName)
    Next
```

The ColumnsCollection object can also be indexed by column name. For example, if DataTable dt contains a column named "Description", this code gets a reference to the associated DataColumn
object:

```
Dim column As DataColumn = dt.Columns("Description")
```

6.6.6 Changing, Adding, and Deleting Rows

To change data in a DataSet, first navigate to a row of interest and then assign new values to one or more of its columns. For example, the following line of code assumes that row is a DataRow object that contains a column named "Description". The code sets the value of the column in this row to be "Milk and cheese":

```
row("Description") = "Milk and cheese"
```

Adding a new row to a table in a DataSet is a three-step process:
1. Use the DataTable class's NewRow method to create a new DataRow. The method takes no parameters.
2. Set the values of the columns in the row.
3. Add the new row to the table.

For example, assuming that dt is a DataTable object, and that the table has columns named
"CategoryName" and "Description", this code adds a new row to the table:

```
' Add a row.
    Dim row As DataRow = dt.NewRow( )
        row("CategoryName") = "Software"
        row("Description") = "Fine code and binaries"
        dt.Rows.Add(row)
```

The DataRow object referenced by row in this code can be indexed by the names "CategoryName" and "Description" because the DataRow object was created by the DataTable object's NewRow method and so has the same schema as the table. Note that the NewRow method does not add the row to the table. Adding the new row to the table must be done explicitly by calling the DataRowCollection class's Add method through the DataTable class's Rows property. Deleting a row from a table is a one-liner. Assuming that row is a reference to a DataRow, this line deletes the row from its table:

```
row.Delete( )
```

When changes are made to a row, the DataRow object keeps track of more than just the new column values. It also keeps track of the row's original column values and the fact that the row has been changed. The Item property of the DataRow object is overloaded to allow you to specify the desired version of the data that you wish to retrieve. The syntax of this overload is:

Public Overloads ReadOnly Property Item(_ByVal *columnName* As String, _ByVal *version* As System.Data.DataRowVersion _) As Object

The parameters are:

columnName
The name of the column whose value is to be retrieved.

version
The version of the data to retrieve. This value must be a member of the System.Data.DataRowVersion enumeration. Its values are:
Current Retrieve the current (changed) version.

Default
Retrieve the current version if the data has been changed, the original version if not.
Original Retrieve the original (unchanged) version.

Proposed
Retrieve the proposed change. Proposed changes are changes that are made after a call to a DataRow object's BeginEdit method but before a call to its EndEdit or CancelEdit methods.For more information. For example, after making some changes in DataRow row, the following line displays the original version of the row's Description column:

```
    Console.WriteLine(row("Description", DataRowVersion.Original))
    The current value of the row would be displayed using any of the following lines:

    Console.WriteLine(row("Description", DataRowVersion.Current))
    Console.WriteLine(row("Description", DataRowVersion.Default))
    Console.WriteLine(row("Description"))
```

Calling the DataSet object's AcceptChanges method commits outstanding changes. Calling the DataSet object's RejectChanges method rolls records back to their original versions. The code shown in this section affects only the DataSet object, not the data source. To propagate these changes, additions, and deletions back to the data source, use the Update method of the SqlDataAdapter class, as described in Section 6.5.7. If there are relations defined between the DataTables in the DataSet, it may be necessary to call the DataRow object's BeginEdit method before making changes. For more information, see Section 6.6 later in this chapter.

6.6.7 Writing Updates Back to the Data Source

Because DataSets are always disconnected from their data sources, making changes in a DataSet never has any effect on the data source. To propagate changes, additions, and deletions back to a data source, call the SqlDataAdapter class's Update method, passing the DataSet and the name of the table that is to be updated. For example, the following call to Update writes changes from the DataTable named Categories back to the SQL Server table of the same name:
da.Update(ds, "Categories") Before using the Update method, however, you should understand how an SqlDataAdapter object performs updates. To change, add, or delete records, an SqlDataAdapter object must send SQL UPDATE, INSERT, or DELETE statements, respectively, to SQL Server. The forms of these statements either can be inferred from the SELECT statement that was provided to the SqlDataAdapter object or can be explicitly provided to the sqlDataAdapter object. Example 6-2 shows an example of allowing an SqlDataAdapter object to infer the SQL UPDATE, INSERT, and DELETE statements required for applying updates to a database.

Example 6-2. Allowing an SqlDataAdapter object to infer SQL UPDATE, INSERT, and DELETE statements from a SELECT statement

```
    ' Open a database connection.
    Dim strConnection As String = _"Data Source=localhost;Initial Catalog=Northwind;"
    _& "Integrated Security=True"

    Dim cn As SqlConnection = New SqlConnection(strConnection)
    cn.Open( )
        ' Create a data adapter object and set its SELECT command.
        Dim strSelect As String = _"SELECT * FROM Categories"
        Dim da As SqlDataAdapter = New SqlDataAdapter(strSelect, cn)
        ' Set the data adapter object's UPDATE, INSERT, and DELETE
        ' commands. Use the SqlCommandBuilder class's ability to auto-
        ' generate these commands from the SELECT command.
    Dim autogen As New SqlCommandBuilder(da)
        ' Load a data set.
    Dim ds As DataSet = New DataSet( )
        da.Fill(ds, "Categories")
        ' Get a reference to the "Categories" DataTable.
    Dim dt As DataTable = ds.Tables("Categories")
        ' Modify one of the records.
    Dim row As DataRow = dt.Select("CategoryName = 'Dairy Products'")(0)
    row("Description") = "Milk and stuff"
        ' Add a record.
```

```
row = dt.NewRow( )
row("CategoryName") = "Software"
row("Description") = "Fine code and binaries"
dt.Rows.Add(row)
' Delete a record.
row = dt.Select("CategoryName = 'MyCategory'")(0)
row.Delete( )
' Update the database.
da.Update(ds, "Categories")
' Close the database connection.
cn.Close( )
```

Note the following in Example 6-2:

1. A SqlDataAdapter object is constructed with an argument of "SELECT * FROM Categories". This initializes the value of the SqlDataAdapter object's SelectCommand property.
2. A SqlCommandBuilder object is constructed with the SqlDataAdapter object passed as an argument to its constructor. This step hooks the SqlDataAdapter object to the SqlCommandBuilder object so that later, when the SqlDataAdapter object's Update method is called, the SqlDataAdapter object can obtain SQL UPDATE, INSERT, and DELETE commands from the SqlCommandBuilder object.
3. The SqlDataAdapter object is used to fill a DataSet object. This results in the DataSet object containing a DataTable object, named "Categories", that contains all the rows from the Northwind database's Categories table.
4. One record each in the table is modified, added, or deleted.
5. The SqlDataAdapter object's Update method is called to propagate the changes back to the database. Step 5 forces the SqlCommandBuilder object to generate SQL statements for performing the database update, insert, and delete operations.When the Update method is called, the SqlDataAdapter object notes that no values have been set for its UpdateCommand, InsertCommand, and DeleteCommand properties, and therefore queries the SqlCommandBuilder object for these commands. If any of these properties had been set on the SqlDataAdapter object, those values would have been used instead. The SqlCommandBuildObject can be examined to see what commands were created. To see the commands that are generated in Example 6-2, add the following lines anywhere after the declaration and assignment of the autogen variable:

```
Console.WriteLine("UpdateCommand:" &autogen.GetUpdateCommand.CommandText)
Console.WriteLine("InsertCommand: " &
    autogen.GetInsertCommand.CommandText)

Console.WriteLine("DeleteCommand:" &
    autogen.GetDeleteCommand.CommandText)
```

The auto-generated UPDATE command contains the following text
(note that line breaks have been added for clarity in the book):

```
UPDATE Categories
SET CategoryName = @p1 , Description = @p2 , Picture = @p3
WHERE ((CategoryID = @p4) AND ((CategoryName IS NULL AND @p5 IS
NULL) OR (CategoryName = @p6)) )
```

Similarly, the INSERT command is:

```
INSERT INTO Categories( CategoryName , Description , Picture )
VALUES ( @p1 , @p2 , @p3)
```

And the DELETE command is:

```
DELETE FROM Categories
WHERE (
(CategoryID = @p1)
AND
((CategoryName IS NULL AND @p2 IS NULL) OR (CategoryName = @p3)) )
```

Note the use of formal parameters (@p0, @p1, etc.) in each of these statements. For each row that is to be changed, added, or deleted, the parameters are replaced with values from the row, and the resulting SQL statement is issued to the database. The choice of which value from the row to use for which parameter is controlled by the SqlCommand object's Parameters property. This property contains an SqlParameterCollection object that in turn contains one SqlParameter object for each formal parameter. The SqlParameter object's ParameterName property matches the name of the formal parameter (including the "@"), the SourceColumn property contains the name of the column from which the value is to come, and the SourceVersion property specifies the version of the value that is to be used. Row versions were discussed in Section 6.5.6.
If desired, a DataSet object's UpdateCommand, InsertCommand, and DeleteCommand properties can be set directly. Example 6-3 sets the value of UpdateCommand and then performs an update using this command.

Example 6-3. Setting a DataSet object's UpdateCommand property

```
' Open a database connection.
Dim strConnection As String = _
"Data Source=localhost;Initial Catalog=Northwind;" _
& "Integrated Security=True"
Dim cn As SqlConnection = New SqlConnection(strConnection)
cn.Open( )
' Set up a data adapter object.Dim da As SqlDataAdapter = New
SqlDataAdapter("SELECT * FROM Categories",cn)

   ' Create an UPDATE command.
   ' This is the command text.
   ' Note the parameter names: @Description and @CategoryID.
Dim strUpdateCommand As String = _"UPDATE Categories" _
& " SET Description = @Description" _& " WHERE CategoryID = @CategoryID"

'Create a SqlCommand object and assign it to the UpdateCommand
property.da.UpdateCommand=New
SqlCommand(strUpdateCommand, cn)

' Set up parameters in the SqlCommand object.
Dim param As SqlParameter
'
' @CategoryID
param = da.UpdateCommand.Parameters.Add( _
New SqlParameter("@CategoryID", SqlDbType.Int))
param.SourceColumn = "CategoryID"
param.SourceVersion = DataRowVersion.Original
'
' @Description

param = da.UpdateCommand.Parameters.Add( _
New SqlParameter("@Description", SqlDbType.NChar, 16))
param.SourceColumn = "Description"
param.SourceVersion = DataRowVersion.Current

' Load a data set.

Dim ds As DataSet = New DataSet( )
da.Fill(ds, "Categories")
' Get the table.

Dim dt As DataTable = ds.Tables("Categories")

'Get a row.
Dim row As DataRow = dt.Select("CategoryName = 'Dairy Products'")(0)
' Change the value in the Description column.

row("Description") = "Milk and stuff"
```

```
' Perform the update.
da.Update(ds, "Categories")
' Close the database connection.
cn.Close( )
```

6.7 RELATIONS BETWEEN DATATABLES IN A DATASET

The DataSet class provides a mechanism for specifying relations between tables in a DataSet. The DataSet class's Relations property contains a RelationsCollection object, which maintains a collection of DataRelation objects. Each DataRelation object represents a parent/child relationship between two tables in the DataSet. For example, there is conceptually a parent/child relationship between a Customers table and an Orders table, because each order must belong to some customer. Modeling this relationship in the DataSet has these benefits:

- The DataSet can enforce relational integrity.
- The DataSet can propagate key updates and row deletions.
- Data-bound controls can provide a visual representation of the relation.

Example 6-4 loads a Customers table and an Orders table from the Northwind database and then creates a relation between them. The statement that actually creates the relation is shown in bold.

Example 6-4. Creating a DataRelation between DataTables in a DataSet

```
' Open a database connection.
Dim strConnection As String = _"Data Source=localhost;Initial Catalog=Northwind;" _&
"Integrated Security=True"

Dim cn As SqlConnection = New SqlConnection(strConnection)
cn.Open( )
' Set up a data adapter object.

Dim strSql As String = "SELECT * FROM Customers" _& " WHERE City = 'Buenos
Aires' AND Country = 'Argentina'"

Dim da As SqlDataAdapter = New SqlDataAdapter(strSql, cn)
' Load a data set.

Dim ds As DataSet = New DataSet( )
da.Fill(ds, "Customers")

' Set up a new data adapter object.
strSql = "SELECT Orders.*" _& " FROM Customers, Orders" _& " WHERE
(Customers.CustomerID = Orders.CustomerID)" _& " AND (Customers.City = 'Buenos
Aires')" _& " AND (Customers.Country = 'Argentina')"

    da = New SqlDataAdapter(strSql, cn)
    ' Load the data set.
    da.Fill(ds, "Orders")
    ' Close the database connection.
    cn.Close( )
' Create a relation.
    ds.Relations.Add("CustomerOrders", _
    ds.Tables("Customers").Columns("CustomerID"), _
    ds.Tables("Orders").Columns("CustomerID"))
```

As shown in Example 6-4, the DataRelationCollection object's Add method creates a new relation between two tables in the DataSet. The Add method is overloaded. The syntax used in Example 6-4 is:

```
    Public Overloads Overridable Function Add( _ByVal name As String,     _ByVal
    parentColumn As System.Data.DataColumn, _ByVal
        childColumn As System.Data.DataColumn _) As      System.Data.DataRelation
```

The parameters are:

name
The name to give to the new relation. This name can be used later as an index to the RelationsCollection object.

parentColumn
The DataColumn object representing the parent column.

childColumn
The DataColumn object representing the child column.
The return value is the newly created DataRelation object. Example 6-4 ignores the return value.

6.8 THE DATASET'S XML CAPABILITIES

The DataSet class has several methods for reading and writing data as XML, including:

GetXml
Returns a string containing an XML representation of the data in the DataSet object.

GetXmlSchema
Returns a string containing the XSD schema for the XML returned by the GetXml method.

WriteXml
Writes the XML representation of the data in the DataSet object to a Stream object, a file, a TextWriter object, or an XmlWriter object. This XML can either include or omit the corresponding XSD schema.

WriteXmlSchema
Writes the XSD schema for the DataSet to a Stream object, a file, a TextWriter object, or an XmlWriter object.

ReadXml
Reads the XML written by the WriteXml method.

ReadXmlSchema
Reads the XSD schema written by the WriteXmlSchema method.
Example 6-5 shows how to write a DataSet to a file as XML using the **WriteXml method.**

Example 6-5. Saving a DataSet to a file as XML

```
' Open a database connection.
Dim strConnection As String = _
"Data Source=localhost;Initial Catalog=Northwind;" _
& "Integrated Security=True"
Dim cn As SqlConnection = New SqlConnection(strConnection)
cn.Open( )
' Set up a data adapter object.
Dim strSql As String = "SELECT * FROM Customers" _
& " WHERE CustomerID = 'GROSR'"
Dim da As SqlDataAdapter = New SqlDataAdapter(strSql, cn)
' Load a data set.
Dim ds As DataSet = New DataSet("MyDataSetName")
da.Fill(ds, "Customers")
' Set up a new data adapter object.
strSql = "SELECT Orders.*" _
& " FROM Customers, Orders" _
& " WHERE (Customers.CustomerID = Orders.CustomerID)" _
& " AND (Customers.CustomerID = 'GROSR')"
da = New SqlDataAdapter(strSql, cn)
' Load the data set.
da.Fill(ds, "Orders")
```

```
' Close the database connection.
cn.Close( )
' Create a relation.
ds.Relations.Add("CustomerOrders", _
ds.Tables("Customers").Columns("CustomerID"), _
ds.Tables("Orders").Columns("CustomerID"))
' Save as XML.
ds.WriteXml("c:\temp.xml")
```

The majority of the code in Example 6-5 simply loads the DataSet with data. Actually writing the XML is done with the DataSet's WriteXml method at the end of Example 6-5. The contents of the file thus created are shown in Example 6-6. Some lines in Example 6-6 have been wrapped for printing in this book.

Example 6-6. The file produced by the code in Example 6-5

```
<?xml version="1.0" standalone="yes"?>
<MyDataSetName>
<Customers>
<CustomerID>GROSR</CustomerID>
<CompanyName>GROSELLA-Restaurante</CompanyName>
<ContactName>Manuel Pereira</ContactName>
<ContactTitle>Owner</ContactTitle>
<Address>5th Ave. Los Palos Grandes</Address>
<City>Caracas</City>
<Region>DF</Region>
<PostalCode>1081</PostalCode>
<Country>Venezuela</Country>
<Phone>(2) 283-2951</Phone>
<Fax>(2) 283-3397</Fax>
</Customers>
<Orders>
<OrderID>10268</OrderID>
<CustomerID>GROSR</CustomerID>
<EmployeeID>8</EmployeeID>
<OrderDate>1996-07-30T00:00:00.0000000-05:00</OrderDate>
<RequiredDate>1996-08-27T00:00:00.0000000-05:00</RequiredDate>
<ShippedDate>1996-08-02T00:00:00.0000000-05:00</ShippedDate>
<ShipVia>3</ShipVia>
<Freight>66.29</Freight>
<ShipName>GROSELLA-Restaurante</ShipName>
<ShipAddress>5th Ave. Los Palos Grandes</ShipAddress>
<ShipCity>Caracas</ShipCity>
<ShipRegion>DF</ShipRegion>
<ShipPostalCode>1081</ShipPostalCode>
<ShipCountry>Venezuela</ShipCountry>
</Orders>
<Orders>
<OrderID>10785</OrderID>
<CustomerID>GROSR</CustomerID>
<EmployeeID>1</EmployeeID>
<OrderDate>1997-12-18T00:00:00.0000000-06:00</OrderDate>
<RequiredDate>1998-01-15T00:00:00.0000000-06:00</RequiredDate>
<ShippedDate>1997-12-24T00:00:00.0000000-06:00</ShippedDate>
<ShipVia>3</ShipVia>
<Freight>1.51</Freight>
<ShipName>GROSELLA-Restaurante</ShipName>
<ShipAddress>5th Ave. Los Palos Grandes</ShipAddress>
<ShipCity>Caracas</ShipCity>
<ShipRegion>DF</ShipRegion>
<ShipPostalCode>1081</ShipPostalCode>
<ShipCountry>Venezuela</ShipCountry>
</Orders>
</MyDataSetName>
```

The syntax of this overloaded version of the WriteXml function is:

Public Overloads Sub WriteXml(ByVal *fileName* As String)
The *fileName* parameter specifies the full path of a file into which to write the XML..The XML document written by the DataSet class's WriteXml method can be read back into a DataSet object using the ReadXml method. Example 6-7 reads back the file written

Example 6-7. Recreating a DataSet object from XML

```
Dim ds As New DataSet( )
ds.ReadXml("c:\temp.xml")
```

The XML created by the WriteXml method contains only data—no schema information. The ReadXml method is able to infer the schema from the data. To explicitly write the schema information, use the WriteXmlSchema method. To read the schema back in, use the ReadXmlSchema method. The GetXml and GetXmlSchema methods work the same as the WriteXml and WriteXmlSchema methods, except that each returns its result as a string rather than writing it to a file.

6.9 BINDING A DATASET TO A WINDOWS FORMS DATAGRID

DataSet and DataTable objects can be bound to Windows Forms DataGrid objects to provide an easy way to view data. This is done by calling a DataGrid object's SetDataBinding method, passing the object that is to be bound to the grid. The syntax of the SetDataBinding method is:

Public Sub SetDataBinding(_ByVal *dataSource* As Object, _ByVal *dataMember* As String _)

The parameters are:

dataSource
The source of the data to show in the grid. This can be any object that exposes the System.Collections.IList or System.Data.IListSource interfaces, which includes the DataTable and DataSet classes discussed in this chapter.

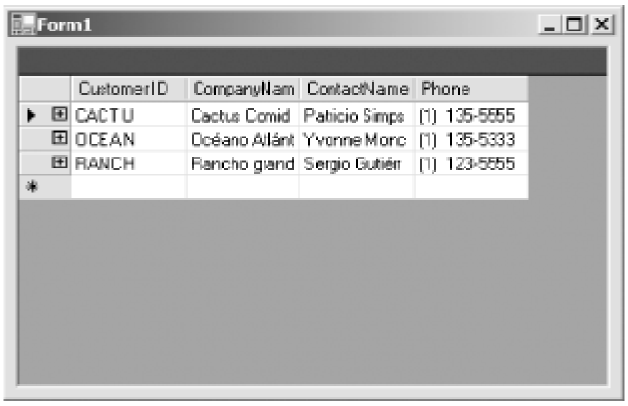
dataMember
If the object passed in the *dataSource* parameter contains multiple tables, as a DataSet object does, the *dataMember* parameter identifies the table to display in the DataGrid. If a DataTable is passed in the *dataSource* parameter, the *dataMember* parameter should contain either Nothing or an empty string. Example 6-8 shows how to bind a DataSource object to a DataGrid.

The DataSource object contains a Customers table and an Orders table, and a relation between them. The call to the DataGrid object's SetDataBinding method specifies that the Customers table should be shown in the grid.Figure 6-3 shows the resulting DataGrid display.

Example 6-8. Creating a DataSet and binding it to a Windows Forms DataGrid

```
' Open a database connection.
Dim strConnection As String = _
"Data Source=localhost;Initial Catalog=Northwind;" _
& "Integrated Security=True"
Dim cn As SqlConnection = New SqlConnection(strConnection)
cn.Open( )
' Set up a data adapter object.
Dim strSql As String = _
"SELECT CustomerID, CompanyName, ContactName, Phone FROM Customers" _
& " WHERE City = 'Buenos Aires' AND Country = 'Argentina'"
Dim da As SqlDataAdapter = New SqlDataAdapter(strSql, cn)
' Load a data set.
Dim ds As DataSet = New DataSet( )
da.Fill(ds, "Customers")
' Set up a new data adapter object.
strSql = _
"SELECT Orders.OrderID, Orders.CustomerID, Orders.OrderDate," _
& " Orders.ShippedDate" _
& " FROM Customers, Orders" _
& " WHERE (Customers.CustomerID = Orders.CustomerID)" _
& " AND (Customers.City = 'Buenos Aires')" _
& " AND (Customers.Country = 'Argentina')"
da = New SqlDataAdapter(strSql, cn)
' Load the data set.
da.Fill(ds, "Orders")
' Close the database connection.
cn.Close( )
' Create a relation.
ds.Relations.Add("CustomerOrders", _
ds.Tables("Customers").Columns("CustomerID"), _
ds.Tables("Orders").Columns("CustomerID"))
' Bind the data set to a grid.
' Assumes that grid contains a reference to a
' System.WinForms.DataGrid object.
grd.SetDataBinding(ds, "Customers")
```

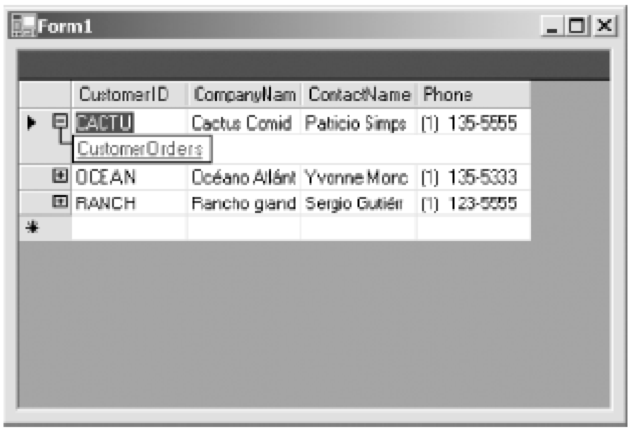
Figure 6-3. The display generated by the code in Example 6-8



CustomerID	CompanyNam	ContactName	Phone
CACTU	Cactus Comid	Paticio Simps	(1) 135-5555
OCEAN	Océano Atlánt	Yvonne Monc	(1) 135-5333
RANCH	Rancho grand	Sergio Gutiérr	(1) 123-5555

Note in Figure 6-3 that each row in this DataGrid has a "+" icon. The reason is that the DataGrid object has detected the relation between the Customers table and the Orders table. Clicking on the "+" reveals all of the relations for which the Customers table is the parent. In this case, there is only one, as shown in Figure 6-4.

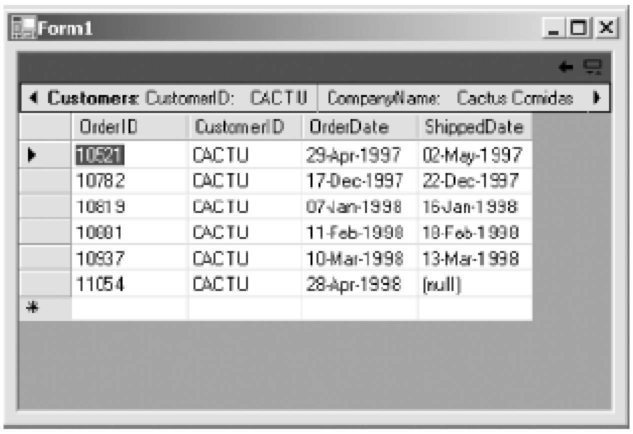
Figure 6-4. Clicking the "+" reveals relations



CustomerID	CompanyNam	ContactName	Phone
CACTU	Cactus Comid	Paticio Simps	(1) 135-5555
CustomerOrders			
OCEAN	Océano Atlánt	Yvonne Monc	(1) 135-5333
RANCH	Rancho grand	Sergio Gutiérr	(1) 123-5555

The name of the relation in the display is a link. Clicking on this link loads the grid with the child table in the relation, as shown in Figure 6-5.

Figure 6-5. The Orders table



Customers: CustomerID: CACTU CompanyName: Cactus Comidas

	OrderID	CustomerID	OrderDate	ShippedDate
▸	10521	CACTU	29-Apr-1997	02-May-1997
	10782	CACTU	17-Dec-1997	22-Dec-1997
	10819	CACTU	07-Jan-1998	16-Jan-1998
	10881	CACTU	11-Feb-1998	18-Feb-1998
	10937	CACTU	10-Mar-1998	13-Mar-1998
	11054	CACTU	28-Apr-1998	(null)
*				

While the child table is displayed, the corresponding row from the parent table is displayed in a header (shown in Figure 6-5). To return to the parent table, click the left-pointing triangle in the upper-right corner of the grid.

6.10 BINDING A DATASET TO A WEB FORMS DATAGRID

Example 6-9 shows how to bind a DataTable object to a Web Forms DataGrid object. Figure 6-6 shows the resulting display in a web browser.

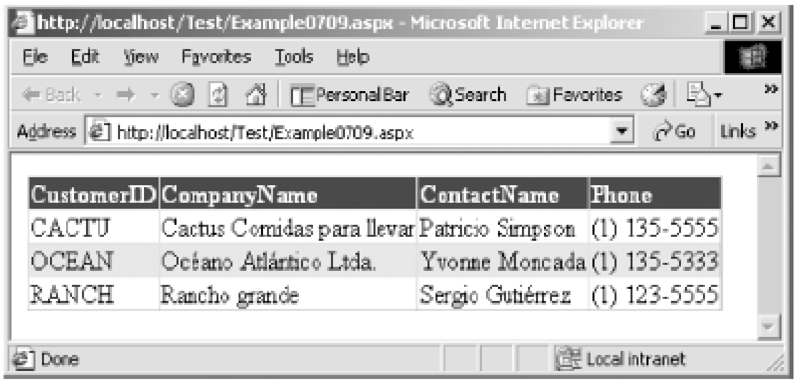
6.6 - 6.9 Check your Progress

1. Each DataRelation object represents a ………….....………………….. relationship between two tables in the DataSet.
2. Data set is …………….………….. table in your program.
3. Data grid has to be bind with ………………..…….. before displaying information.

Example 6-9. Creating a DataTable and binding it to a Web Forms DataGrid

```
<%@ Page Explicit="True" Strict="True" %>
<script language="VB" runat="server">

    Protected Sub Page_Load(ByVal Sender As System.Object, _
        ByVal e As System.EventArgs)
        If Not IsPostback Then ' True the first time the browser hits the
        page.
        ' Bind the grid to the data.
        grdCustomers.DataSource = GetDataSource( )
        grdCustomers.DataBind( )
        End If
    End Sub ' Page_Load

    Protected Function GetDataSource( ) As System.Collections.ICollection
        ' Open a database connection.
        Dim strConnection As String = _
        "Data Source=localhost;Initial Catalog=Northwind;" _
        & "Integrated Security=True"
        Dim cn As New System.Data.SqlClient.SqlConnection(strConnection)
        cn.Open( )
        ' Set up a data adapter object.
        Dim strSql As String = _
        "SELECT CustomerID, CompanyName, ContactName, Phone" _
        & " FROM Customers" _
        & " WHERE City = 'Buenos Aires' AND Country = 'Argentina'"
        Dim da As New System.Data.SqlClient.SqlDataAdapter(strSql, cn)
        ' Load a data set.
        Dim ds As New System.Data.DataSet( )
        da.Fill(ds, "Customers")
        ' Close the database connection.
        cn.Close( )
        ' Wrap the Customers DataTable in a DataView object.
        Dim dv As New System.Data.DataView(ds.Tables("Customers"))
        Return dv

    End Function ' GetDataSource
</script>
<html>
    <body>
        <asp:DataGrid id=grdCustomers runat="server" ForeColor="Black">
            <AlternatingItemStyle BackColor="Gainsboro" />
            <FooterStyle ForeColor="White" BackColor="Silver" />
            <ItemStyle BackColor="White" />
            <HeaderStyle Font-Bold="True" ForeColor="White"
            BackColor="Navy" />
        </asp:DataGrid>
    </body>
</html>
```

Figure 6-6. The display generated by the code in Example 6-9


http://localhost/Test/Example0709.aspx - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Address http://localhost/Test/Example0709.aspx
Go Links
Done
Local intranet

CustomerID	CompanyName	ContactName	Phone
CACTU	Cactus Comidas para llevar	Patricio Simpson	(1) 135-5555
OCEAN	Océano Atlántico Ltda.	Yvonne Moncada	(1) 135-5333
RANCH	Rancho grande	Sergio Gutiérrez	(1) 123-5555

Note the following:

- Unlike the Windows Forms DataGrid class, the Web Forms DataGrid class has no SetDataBinding method. Instead, set the Web Forms DataGrid's DataSource property and then call the DataGrid's DataBind method.
- Unlike the Windows Forms DataGrid class, the Web Forms DataGrid class's DataSource property can't directly consume a DataTable or DataSet. Instead, the data must be wrapped in a DataView or DataSetView object. The properties and methods of the DataView and DataSetView classes provide additional control over how data is viewed in a bound DataGrid.DataView and DataSetView objects can be used by either Windows Forms or Web Forms
DataGrids, but they are mandatory with Web Forms DataGrids.
- The DataGrid's DataSource property can consume any object that exposes the System.Collections.ICollection interface.

Typed DataSets

There is nothing syntactically wrong with this line of code:
Dim dt As System.Data.DataTable = ds.Tables("Custumers")
However, "Custumers" is misspelled. If it were the name of a variable, property, or method, it would cause a compile-time error (assuming the declaration were not similarly misspelled). However, because the compiler has no way of knowing that the DataSet ds will not hold a table called Custumers, this typographical error will go unnoticed until runtime. If this code path is not common, the error may go unnoticed for a long time, perhaps until after the software is delivered and running on thousands of client machines. It would be better to catch such errors at compile time. Microsoft has

provided a tool for creating customized DataSet-derived classes. Such classes expose additional properties based on the specific schema of the data that an object of this class is expected to hold. Data access is done through these additional properties rather than through the generic Item properties. Because the additional properties are declared and typed, the Visual Basic .NET compiler can perform compile-time checking to ensure that they are used correctly. Because the class is derived from the DataSet class, an object of this class can do everything that a regular DataSet object can do, and it can be used in any context in which a DataSet object is expected. Consider again Example 6-1, shown earlier in this chapter. This fragment of code displays the names of the customers in the Northwind database that are located in London. Compare this to Example 6-10, which does the same thing but uses a DataSet-derived class that is specifically designed for this purpose.

Example 6-10. Using a typed DataSet

```
' Open a database connection.
Dim strConnection As String = _
"Data Source=localhost;Initial Catalog=Northwind;" _
& "Integrated Security=True"
Dim cn As SqlConnection = New SqlConnection(strConnection)
cn.Open( )
' Set up a data adapter object.
Dim strSelect As String = "SELECT * FROM Customers WHERE City = 'London'"
Dim da As New SqlDataAdapter(strSelect, cn)
' Load a data set.
Dim ds As New LondonCustomersDataSet( )
da.Fill(ds, "LondonCustomers")
' Close the database connection.
cn.Close( )
' Do something with the data set.
Dim i As Integer
For i = 0 To ds.LondonCustomers.Count - 1
Console.WriteLine(ds.LondonCustomers(i).CompanyName)
```

Next Note that in Example 6-10, ds is declared as type LondonCustomersDataSet, and this class has properties that relate specifically to the structure of the data that is to be loaded into the DataSet.
However, before the code in Example 6-10 can be written, it is necessary to generate the LondonCustomersDataSet and related classes. First, create an XML schema file that defines the desired schema of the DataSet. The easiest way to do this is to write code that loads a generic DataSet object with data having the right schema and then writes that schema using the DataSet class's WriteXmlSchema method. Example 6-11 shows how this was done with the LondonCustomers DataSet.

Example 6-11. Using the WriteXmlSchema method to generate an XML schema

```
    ' This code is needed only once. Its purpose is to create
    ' an .xsd file that will be fed to the xsd.exe tool.
    ' Open a database connection.
    Dim strConnection As String = _
    "Data Source=localhost;Initial Catalog=Northwind;" _
    & "Integrated Security=True"
    Dim cn As SqlConnection = New SqlConnection(strConnection)
    cn.Open( )
    ' Set up a data adapter object.
    Dim strSelect As String = "SELECT * FROM Customers WHERE City = 'London'"
    Dim da As New SqlDataAdapter(strSelect, cn)
    ' Load a data set.
    Dim ds As New DataSet("LondonCustomersDataSet")
    da.Fill(ds, "LondonCustomers")
    ' Close the database connection.
    cn.Close( )
    ' Save as XSD.
ds.WriteXmlSchema("c:\LondonCustomersDataSet.xsd")
```

Next, run Microsoft's *XML Schema Definition Tool* (*xsd.exe*) against the XML schema file you just generated. Here is the command line used for the LondonCustomers DataSet:

```
xsd /d /l:VB LondonCustomersDataSet.xsd
```

The /d option indicates that a custom DataSet and related classes should be created. The :VB
option specifies that the generated source code should be written in Visual Basic .NET (the tool is also able to generate C# source code). With this command line, the tool generates a file named *LondonCustomersDataSet.vb*, which contains the source code. Finally, add the generated *.vb* file to a project and make use of its classes.

6.11 READING DATA USING A DATAREADER

As you have seen, the DataSet class provides a flexible way to read and write data in any data source. There are times, however, when such flexibility is not needed and when it might be better to optimize data-access speed as much as possible. For example, an application might store the text for all of its drop-down lists in a database table and read them out when the application is started. Clearly, all that is needed here is to read once through a result set as fast as possible. For needs such as this, ADO.NET has DataReader classes. Unlike the DataSet class, DataReader classes are connected to their data sources. Consequently, there is no generic DataReader class. Rather, each managed provider exposes its own DataReader class, which implements the System.Data.IDataReader interface. The SQL Server managed provider exposes the SqlDataReader class (in the System.Data.SqlClient namespace). DataReader classes provide sequential, forward-only, read-only access to data. Because they are optimized for this task, they are faster than the DataSet class. Example 6-12 shows how to read through a result set using an SqlDataReader object.

Example 6-12. Using a SqlDataReader object

```
' Open a database connection.
Dim strConnection As String = _
"Data Source=localhost;Initial Catalog=Northwind;" _
& "Integrated Security=True"
Dim cn As SqlConnection = New SqlConnection(strConnection)
cn.Open( )
' Set up a command object.
Dim strSql As String = "SELECT * FROM Customers" _
& " WHERE Country = 'Germany'"
Dim cmd As New SqlCommand(strSql, cn)
' Set up a data reader.
Dim rdr As SqlDataReader
rdr = cmd.ExecuteReader( )
' Use the data.
Do While rdr.Read
Console.WriteLine(rdr("CompanyName"))
Loop
' Close the database connection.
cn.Close( )
```

Opening a connection to the database is done the same as when using a DataSet object. However, with a DataReader object, the connection must remain open while the data is read. Instead of an SqlDataAdapter object, an SqlCommand object is used to hold the command that will be executed to select data from the database. The SqlCommand class's ExecuteReader method is called to execute the command and to return an SqlDataReader object. The SqlDataReader object is then used to read through the result set. Note the Do While loop in Example 6-12, repeated here:

```
Do While rdr.Read
Console.WriteLine(rdr("CompanyName"))
```

Loop Developers who are used to coding against classic ADO will note that this loop appears to lack a "move to the next row" statement. However, it is there. The SqlDataReader class's Read method performs the function of positioning the SqlDataReader object onto the next row to be read. In classic ADO, a RecordSet object was initially positioned on the first row of the result set. After reading each record, the RecordSet object's MoveNext method had to be called to position the RecordSet onto the next row in the result set. Forgetting to call MoveNext was a common cause of infinite loops. Microsoft removed this thorn as follows:

 The DataReader object is initially positioned just prior to the first row of the result set (and therefore has to be repositioned before reading any data).
 The Read method repositions the DataReader to the next row, returning True if the DataReader is positioned onto a valid row and False if the DataReader is positioned past the last row in the result set. These changes result in tight, easy-to-write loops such as the one in Example 6-12. The DataReader provides an Item property for reading column values from the current row. The Item property is overloaded to take either an integer that specifies the column number, which is zero-based, or a string that specifies the column name. The Item property is the default property of the SqlDataReader class, so it can be omitted. For example, this line:

```
Console.WriteLine(rdr("CompanyName"))
```

is equivalent to this line:

```
Console.WriteLine(rdr.Item("CompanyName"))
```

6.12 Executing Stored ProceduresThrough a SqlCommand Object

To execute a stored procedure, set an SqlCommand object's CommandText property to the name of the stored procedure to be executed, and set the CommandType property to the constant CommandType.StoredProcedure (defined in the System.Data namespace). Then call the ExecuteNonQuery method. Example 6-13 does just that.

6.10 - 6.12 Check Your Progress

1. …………………….. classes provide sequential forward-only read-only access to data.
2. Stored procedure are pre written ……………………. in SQL.
3. Data reader is …………………………….. type of object.

Example 6-13. Executing a parameterless stored procedure

```
' Open a database connection.
Dim strConnection As String = _
"Data Source=localhost;Initial Catalog=Northwind;" _
& "Integrated Security=True"
Dim cn As SqlConnection = New SqlConnection(strConnection)
cn.Open( )
' Set up a command object. (Assumes that the database contains a
' stored procedure called "PurgeOutdatedOrders".)
Dim cmd As New SqlCommand("PurgeOutdatedOrders", cn)
cmd.CommandType = CommandType.StoredProcedure
' Execute the command.
cmd.ExecuteNonQuery( )
' Close the database connection.
cn.Close( )
```

Example 6-13 assumes for the sake of demonstration that the database contains a stored procedure called "PurgeOutdatedOrders". If you would like to have a simple stored procedure that works with Example 6-13, use this one:

```
CREATE PROCEDURE PurgeOutdatedOrders AS
DELETE FROM Orders
WHERE OrderDate < '04-Jul-1990'
AND ShippedDate IS NOT NULL
```

See your SQL Server documentation for information on how to create stored procedures.
Some stored procedures have parameters, and some have a return value. For these stored procedures, the SqlCommand class provides the Parameters property. The Parameters property contains a reference to an SqlParameterCollection object. To pass parameters to a stored procedure and/or to read the return value of a stored procedure, add SqlParameter objects to this collection. Example 6-14 calls a stored procedure that takes a single argument.

Example 6-14. Executing a parameterized stored procedure

```
' Open a database connection.
Dim strConnection As String = _
```

```
"Data Source=localhost;Initial Catalog=Northwind;" _
& "Integrated Security=True"
Dim cn As SqlConnection = New SqlConnection(strConnection)
cn.Open( )
' Set up a command object. (Assumes that the database contains a
' stored procedure called "PurgeOutdatedOrders2".)
Dim cmd As New SqlCommand("PurgeOutdatedOrders2", cn)
cmd.CommandType = CommandType.StoredProcedure
' Set up the @BeforeDate parameter for the stored procedure.
Dim param As New SqlParameter("@BeforeDate", SqlDBType.DateTime)
param.Direction = ParameterDirection.Input
param.Value = #7/4/1990#
cmd.Parameters.Add(param)
' Execute the command.
cmd.ExecuteNonQuery( )
' Close the database connection.
cn.Close( )
```

Example 6-14 assumes for the sake of demonstration that the database contains a stored procedure called "PurgeOutdatedOrders2". If you would like to have a simple stored procedure that works with Example 6-14, use this one:

```
CREATE PROCEDURE PurgeOutdatedOrders2
@BeforeDate datetime
AS
DELETE FROM Orders
WHERE OrderDate < @BeforeDate
AND ShippedDate IS NOT NULL
```

See your SQL Server documentation for information on how to create stored procedures.
The steps taken in Example 6-14 are:

1. Open a connection to the database.
2. Instantiate an SqlCommand object using this constructor:
3. Public Overloads Sub New(_
4. ByVal *cmdText* As String, _
5. ByVal *connection* As System.Data.SqlClient.SqlConnection _)

   The ***cmdText*** parameter specifies the name of the stored procedure, and the *connection*
   parameter specifies the database connection to use.

6. Set the SqlCommand object's CommandType property to CommandType.StoredProcedure to indicate that the *cmdText* parameter passed to the constructor is the name of a stored procedure.
7. Create an SqlParameter object to pass a value in the PurgeOutdatedOrders2 stored procedure's *@BeforeDate* parameter. This is done as follows:
   a. Instantiate an SqlParameter object using this constructor:
   b. Public Overloads Sub New(_
   c. ByVal *parameterName* As String, _
   d. ByVal *dbType* As System.Data.SqlClient.SqlDbType _)

The *parameterName* parameter specifies the name of the stored procedure parameter and should match the name as given in the stored procedure. The *dbType* parameter specifies the SQL Server data type of the parameter. This parameter can take any value from the SqlDbType enumeration.

   e. Set the SqlParameter object's Direction property to ParameterDirection.Input. This indicates that a value will be passed from the application to the stored procedure.
   f. Set the Value property of the SqlParameter object.
   g. Add the SqlParameter object to the SqlCommand object's Parameters collection by calling the SqlParameterCollection object's Add method.

6. Execute the stored procedure.

Note the SqlParameter class's Direction property. Setting this property to the appropriate value from the ParameterDirection enumeration (declared in the System.Data namespace), can make a SqlParameter object an input parameter, an output parameter, an in/out parameter, or the stored procedure's return value. The values in the ParameterDirection enumeration are:

Input
The parameter provides a value to the stored procedure.

InputOutput
The parameter provides a value to the stored procedure and receives a new value back from the stored procedure.
Output
The parameter receives a value back from the stored procedure.

ReturnValue
The parameter receives the stored procedure's return value.

6.13 SUMMARY

In this chapter, you learned about Microsoft's data-access technology, ADO.NET. You learned how to connect to a database, how to read data with either a DataSet object or a DataReader object, how to navigate and change data in a DataSet, how to use the DataSet's XML capabilities, how to generate typed DataSets, and how to execute stored procedures using an SqlCommand object.

6.14 CHECK YOUR PROGRESS-*ANSWERS*

6.1- 6.5
1. Data consumer
2. SqlConnection
3. DataSet

6.6 - 6.9
1. Parent / child
2. Temprary Table
3. Dataset

6.10 - 6.12
1. DataReader
2. Procedure
3. Forward Only stream

6.15 QUESTIONS FOR SELF-STUDY

1. Write steps of connecting to a SQL Server Database?
2. Explain Binding a Dataset to a Web Forms Data Grid?
3. Describe dataset and its working in details.
4. Describe ADO.Net architecture with it’s working

6.16 SUGGESTED READINGS

1. Programming Visual Basic .NET by Dave Grundgeiger Publisher: O'Reilly

◈◈◈

NOTES

Math Functions In .Net

Math functions are provided by the members of the Math class (defined in the System namespace). All members of the Math class are shared, so it is not necessary to instantiate the class before accessing its members. Members are simply accessed through the class name. For example, the following line computes the cosine of 45:
Dim result As Double = Math.Cos(45)
The Math class exposes two constants:

E
The base of natural logarithms.

PI
The ratio of the circumference of a circle to its diameter.

The methods of the Math class are as follows. Note that the trigonometric functions consider all angle values to be in radians.

Abs
Computes the absolute value of a number.

Acos
Computes the angle whose cosine is the given number.

Asin
Computes the angle whose sine is the given number.

Atan
Computes the angle whose tangent is the given number.

Atan2
Computes the angle whose tangent is equal to the quotient of the two given numbers.

Ceiling
Computes the smallest whole number greater than or equal to the given number.

Cos
Computes the cosine of a number.

Cosh
Computers the hyperbolic cosine of a number.

Exp
Computes e raised to a given power.

Floor
Computes the largest whole number less than or equal to a given number.

IEEERemainder
Calculates the remainder in the division of two numbers.

Log
Calculates the logarithm of a number (either the natural logarithm or in a given base).

Log10
Calculates the base 10 logarithm of a number.

Max
Returns the larger of two numbers.

Min
Returns the smaller of two numbers.

Pow
Raises a given number to a given power.

Round
Rounds a number to either a whole number or a specified decimal place.

Sign
Returns -1, 0, or 1 to indicate whether the argument is negative, zero, or positive, respectively.

Sin
Calculates the sine of a number.

Sinh
Calculates the hyperbolic sine of a number.

Sqrt
Calculates the square root of a number.

Tan
Calculates the tangent of a number.

Tanh
Calculates the hyperbolic tangent of a number.

◈◈◈