ASP.NET for Developers

Michael Amundsen

DRAFT

SAMS

201 West 103rd St., Indianapolis, Indiana, 46290 USA

0-672-32038-x
Summer 2001

CHAPTER

4

Understanding Visual Basic.NET Syntax and Structure

IN THIS CHAPTER

All the examples in this book are written in Visual Basic.NET. Why, you ask, have we decided to use Visual Basic exclusively since the .NET platform supports a plethora of languages? Why not pepper the text with examples in C#, Jscript, and maybe even Eiffel? We decided to concentrate our efforts on only one language to simplify things and to keep the book to a reasonable length. While it's certainly nice to be able to develop ASP.NET applications using a number of different languages, let's face it: Most programmers prefer to program in a single language. But why have we decided to use Visual Basic? After all, isn't C# now Microsoft's *preferred* language? Quite the contrary: Visual Basic is now on equal footing to C++ and the new C#. In addition to this fact, we have chosen to use Visual Basic.NET in this book for several reasons. Visual Basic is the most popular programming language in the world. It's also by the far the most common language that existing ASP developers have used to create "classic" ASP pages. Finally, it's the language that the authors of this book cut our teeth on—the language that we personally prefer to use.

More than likely, you fall into one of three categories of Visual Basic (VB) developers:

1. You have little or no experience developing applications with Visual Basic or the VBScript scripting language.
2. You have considerable experience developing ASP applications using VBScript but little or no experience with VB proper.
3. You have considerable experience using the Visual Basic language (and perhaps VBScript as well).

This chapter attempts to introduce the Visual Basic.NET language to you regardless of which of these three groups you fall into. For VB novices, this chapter will bring you up to speed in a hurry. For VBScripters, this chapter will help you make the jump from VBScript to Visual Basic. And finally, for the savvy VB developer, this chapter will help you scope out the changes made to your trusty old language.

NOTE

This chapter and the other chapters in this book discuss and use the Visual Basic.NET language, but not the Visual Basic.NET product that's part of Visual Studio.NET. You do not have to own Visual Studio.NET to use the examples in this book.

The New Look of Visual Basic

To borrow the catch phrase of a now defunct U.S. car manufacturer, "This is not your father's Visual Basic!" While true to its heritage, Visual Basic.NET is a much-improved version of the venerable Visual Basic language that many of us have grown to love. Visual Basic has matured into a full-featured, object-oriented language. But unlike previous releases of Visual Basic, this version of VB was rebuilt from the ground up. Literally.

In moving to VB.NET, Microsoft has ditched a number of older, arcane features like GoSub and default properties, and totally reworked features such as arrays and data types. Other native features like the MsgBox function and the Cxxx convert functions have been demoted. These demoted features are still in VB.NET but Microsoft is recommending that you move to using the .NET System classes instead. Of course, depending on your experience and base of existing legacy VB applications, some of the changes may cause considerable pain. More than likely, however, you will soon grow to appreciate the redesigned VB language.

What does the new Visual Basic.NET language mean to the average ASP developer who has written thousands of lines of VBScript code but who has had little exposure to VB proper? If you find yourself in this category of developer, you may experience a short period of bewilderment, as you get accustomed to the wealth of new features offered by VB.NET, features that VBScript never offered. But soon enough, you will start to forget the limited VBScript language and grow to appreciate and even love the much more nimble and full-featured VB.NET.

NOTE

See Appendix A for more information on upgrading to VB.NET from VB 6.0 or VBScript.

Getting Started with VB

Compared to many programming languages, Visual Basic.NET is a fairly easy language to learn. Unlike the C family of languages, VB.NET prefers to use the English language rather than cryptic symbols like `&&`, `||`, and `%`. Unlike prior versions of the VB language, however, VB.NET is a full-featured object-oriented language that can hold its own when compared to C++, C#, or Java.

The remainder of this chapter consists of a walkthrough of the essential elements of the VB.NET language.

Statements and Lines

VB.NET statements can be placed on one or more lines. Unlike C++, C#, and Java, there is no statement terminator character in VB. When continuing a statement across more than one line, you must end continuation lines with a space followed by an underscore character (_).

For example, the following VB.NET statement spans two lines:

```
Function CreateFullName(LastName As String , _
 FirstName As String)
```

Comments

You can add comments to your code using the apostrophe (`'`) character. Everything to the right of an apostrophe is ignored by the VB.NET compiler:

```
x = y + 5 'Add 5 to the value of y
```

NOTE

VB.NET does not support multiline comments like some other languages.

Operators

Like any programming language, VB.NET has its assortment of operators. The most common of these operators are summarized in Table 4.1.

Table 4.1 Common VB.NET Operators

Type	*Operator*	*Purpose*	*Example*
Math	+	Add	5 + 2 = 7
	–	Subtract	5 – 2 = 3
	*	Multiply	5 * 2 = 10
	/	Divide	5 / 3 = 2.5
	\	Integer Divide	5 \ 2 = 2
	^	Exponentiation	5^2 = 25
	Mod	Remainder after integer division	5 mod 2 = 1
String	+	Concatenate	“one” + “two” = “onetwo”
	&	Concatenate	“one” & “two” = “onetwo”
Assignment	=	Assigns the value an expression to the variable	x = 5 + 3

Type	*Operator*	*Purpose*	*Example*
	+=	Adds the value of a variable by an expression and assigns the result to the variable*	x += y
	-=	Subtracts the value of a variable by an expression and assigns the result to the variable	x -= y
	*=	Multiplies the value of a variable by an expression and assigns the result to the variable*	x *= y
	/=	Divides the value of a variable by an expression and assigns the result to the variable*	x /= y
	\=	Integer divides the value of a variable by an expression and assigns the result to the variable	x \= y
	&=	Concatenates the value of a variable by an expression and assigns the result to the variable*	x &= y

Table 4.1 Continued

Type	*Operator*	*Purpose*	*Example*
	^=	Exponentiates the value of a variable by an expression and assigns the result to the variable*	x ^= y
Comparison	=	Is equal to	If (x = y)
	<	Is less than	If (x < y)
	<=	Is less than or equal to	If (x <= y)
	>	Is greater than	If (x > y)
	>=	Is greater than or equal to	If (x >= y)
	<>	Is not equal to	If (x <> y)
	Like	Matches a pattern*	If (x Like "p??r")
	Is	Do object variables refer to same object	If (x Is y)
Logical	And	True if both expressions are true	If (x = 3 And y = 4)
	Or	True if one or both expressions are true	If (x = 3 Or y = 4)
	Not	True if the expression is False	If Not (x = 5)
	Xor	True if one expression is true, but not both*	If (x = 3 Xor y = 4)

* *This operator was introduced in VB.NET.*

You will find a number of examples that use the VB.NET operators scattered about the chapter.

Using Procedures

The basic unit of executable code in VB.NET, as in most programming languages, is the *procedure*. VB supports two basic types of procedures: the subroutine (or sub) and the function.

Subroutines

You declare a subroutine with the Sub statement. For example

```
Sub HelloWorld()
    Response.Write("Hello World")
End Sub
```

You call a sub using either of the following statements:

```
HelloWorld()
Call HelloWorld()
```

NOTE

Unlike prior versions of VB, VB.NET requires parentheses around argument lists whether or not you use the Call keyword.

Functions

Functions in VB.NET are similar in functionality to subroutines with one difference: Functions can return a value to the calling program. You create a function with the Function statement. For example, the following function returns "Hello World" to the calling code:

```
Function SayHello()
    Return "Hello World"
End Function
```

NOTE

Prior versions of VB used a different syntax for returning a value from a function.

Using Variables and Parameters

You use the `Dim`, `Private`, `Protected`, `Friend`, or `Public` statements in VB.NET to declare a variable and its data type. Which statement you use depends on where you wish to declare the variable.

To declare a variable from within a subroutine or function, you use the `Dim` statement. For example

```
Function DoSomething()
    Dim Counter As Integer
End Function
```

A variable declared using `Dim` is local to the procedure in which it is declared.

To declare a variable that's global to the entire page, you declare the variable outside of any subroutine or function using the `Private` statement. For backward compatibility, `Dim` also works in this context, but it's best to use `Private` instead.

NOTE

The `Public`, `Friend`, and `Protected` statements are discussed later in the chapter when we introduce classes.

New for VB.NET, you can both declare a variable and set its initial value in one statement. For example

```
Dim Age As Integer = 23
Private Company As String = "Microsoft"
```

VB.NET supports the data types shown in Table 4.2.

TABLE 4.2 Visual Basic.NET Data Types

Visual Basic Data Type	*.NET Runtime Data Type*	*Storage Size*	*Range of Values*
Boolean	System.Boolean	4 bytes	True or False
Byte	System.Byte	1 byte	0 to 255 (unsigned)
Char	System.Char	2 bytes	1 Unicode "" character
Date	System.DateTime	8 bytes	January 1, 0001 to December 31,9999 12:00:00 AM

Visual Basic Data Type	*.NET Runtime Data Type*	*Storage Size*	*Range of Values*
Decimal	System. Decimal	12 bytes	+/-0.0 79,228,162,514,264,337, 593,543,950,335 with no decimal point;+/- 7.922816251426433 7593543950335 with 28 places to the right of the decimal; smallest non-zero number is +/-0.000000000 0000000000000000001
Double (double-precision floating-point)	System. Double	8 bytes	-1.797693134862310.0 E308 to -4.940656 45841247E-324 for negative values; 4.94065645841247E -324 to 1.7976931 3486232E308 for positive values
Integer	System.Int32	4 bytes	-2,147,483,648 to 2,147,483,647
Long (long integer)	System.Int64	8 bytes	-9,223,372,036,854,775,80 8 to 9,223,372,036,854,775, 807
Object	System.Object	4 bytes	Any data type Depends on usage.
Short	System.Int16	2 bytes	-32,768 to 32,7670
Single (single-precision floating-point)	System.Single	4 bytes	-3.402823E38 to 0.0 -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values
String ""	System.String	10 bytes + (2 * string length)	0 to approximately 2 billion Unicode characters

You may have noticed that there is no entry for `Variant` in Table 4.2. That's because VB.NET no longer supports the `Variant` data type. However, you can use the generic `Object` type any place you would have used `Variant` in prior versions of VB. (In VB.NET, Variant is a synonym for Object.)

Unlike prior versions of VB, if you use a declare statement as shown in the following example, all three variables will be declared as integers:

```
Dim x, y, z As Integer
```

In prior versions of VB, `x` and `y` would be declared as variant variables and only `z` would be declared as an Integer.

Constants

You can use the `Const` statement to declare a constant. Like a variable, a constant holds a value; however, a constant's value is set at design time and may not change. You can include the `Private` or `Public` keyword within the `Const` statement to alter the scooping of the constant declaration. Here are a few examples:

```
Const Pi As Double = 3.14159
Private Const CmPerInch As Double = 2.54
Public Const BookTitle As String = "ASP for Developers"
```

In addition to user-defined constants, VB.NET and the .NET Framework define a number of intrinsic constants. For example, you can use the intrinsic constant `CrLf` anytime you wish to add a carriage return and line feed to a string:

```
MsgString = "An error has occurred in the program." & _
 CrLf & "Click on OK to continue or CANCEL to abort."
```

NOTE

Most of the old intrinsic constants have changed names in VB.NET. For example, in VB 6.0 and VBScript, you would use `vbCrLf` instead of `CrLf`.

Implicit and Explicit Variable Declarations

VB has always supported implicit variable declarations, which means that you are not required to declare your variables or parameters before using them. However, most professional developers agree that you should not take advantage of this VB feature unless you like bugs in your code. The issue is best demonstrated with an example:

```
Function Multiply(number1, number2)
    Return number1 * numbr2
End Function
```

The `Multiply` function will always return 0 because we misspelled one of the parameters. This happens because VB.NET implicitly declares `numbr2` and initializes it to 0 because it is used in a numeric context. You can avoid this type of hard-to-find bug by using `Option Explicit` or `Option Strict`. In this example, if you had used either of these options, VB.NET would generate a compile-time error when the page was compiled.

Option Explicit Versus Option Strict

VB has always had the `Option Explicit` declaration, which forces you to declare all your variables, but VB.NET also introduces `Option Strict`, which goes one step further. In addition to forcing you to declare all your variables, `Option Strict` restricts the types of implicit conversions that the language allows. When you use `Option Strict`, VB won't allow conversions where data loss would occur. `Option Strict` also disallows implicit conversions between numeric and string data types.

To specify `Option Explicit`, you can use the following page directive at the top of the ASP page:

```
<%@ Page Explicit="True" %>
```

To specify `Option Strict`, you can use the following page directive at the top of the ASP page:

```
<%@ Page Strict="True" %>
```

Arrays

You create arrays in VB.NET using the `Dim`, `Public`, or `Private` statements. You use parentheses to specify that you wish to declare an array rather than a scalar variable. For example, the following statement creates an array of strings:

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

Before using an array, you must specify the total number of elements in the array with the `ReDim` statement:

```
Dim Names() As String
ReDim Names(2)
Names(0) = "Mike"
Names(1) = "Paul"
```

All arrays have a lower bound of zero. The number you place between the parentheses of the `ReDim` statement designates the total number of elements the array will hold. Thus, a value of 2 as shown in this example tells VB that the array will hold two string elements, numbered 0 and 1.

NOTE

In prior versions of VB, the number you placed between the parentheses of the `ReDim` statement designated the index number of the highest element, not the total number of elements.

Multidimensional Arrays

The `Names` array in the previous example is a one-dimensional array. You can create arrays of multiple dimensions by using commas when you declare the array. For example, the following statements create a two-dimensional array of strings named `Customer` and a three-dimensional array of double-precision numbers named `Cube`:

```
Private Customer( , ) As String
Private Cube ( , , ) As Double
```

You would use the following code to specify that the `Cube` array would hold 27 elements (3 x 3 x 3 = 27):

```
ReDim Cube(3,3,3)
```

The following code sets the value of several elements in the `Cube` array:

```
Cube(0,0,0) = 23.4
Cube(0,0,1) = 14.6
Cube(2,1,0) = - 13.7
Cube(2,2,2) = 4899.231
```

In addition to using `ReDim`, when declaring an array you can specify the initial size of the array. For example:

```
Dim Names(2) As String
```

Declaring the initial dimensions of an array this way, however, does not restrict you from later resizing the array using `ReDim`.

You can also set the values of an array when you declare it. For example

```
Dim Names() As String = {"Mike", "Paul"}
```

ReDim and ReDim Preserve

Using ReDim erases any existing values of the array. However, you can use the Preserve keyword to preserve the values. For example, the following code redimensions the Colors array without using the Preserve keyword:

```
Dim i As Integer
Dim Colors(3) As String
Colors(0) = "Red"
Colors(1) = "Green"
Colors(2) = "Blue"
ReDim Colors(5)
Colors(3) = "White"
Colors(4) = "Black"

For i = 0 To UBound(Colors)
    Response.Write("<br>" & i & "=" & Colors(i))
Next
```

This produces the following output:

```
0=
1=
2=
3=White
4=Black
```

NOTE

The For Next loop will be introduced later in this chapter.

Changing the ReDim statement to the following

```
ReDim Preserve Colors(5)
```

changes the output to

```
0=Red
1=Green
2=Blue
3=White
4=Black
```

For multidimensional arrays, only the last dimension's values are preserved when using the Preserve keyword.

Checking an Array's Upper Bound

You can use the `UBound` function to determine the upper boundary of an array. `UBound` returns the largest available subscript for the array (*not* the number of elements). For example, in the following code

```
Dim intI, intJ As Integer
Private Square (10 ,5) As Double

intI = UBound(Square, 1)
intJ = UBound(Square, 2)
```

`intI` = 9 and `intJ` = 4.

Passing Parameters

You use parameters to pass information to or from a procedure without having to use global variables. For example, the `Divide` function has two parameters:

```
Function Divide(Numerator As Double, Denominator As Double) As Double
    Return Numerator / Denominator
End Function
```

The `Numerator` and `Denominator` variables in this example have been declared with the `Double` data type.

To declare the data type of the return value of a function, you add `As` *`datatype`* to the `Function` statement after the closing parentheses. The return value of the Divide function was set to Double. The following function has a return value data type of String:

```
Function SayHello() As String
    Return "Hello World"
End Function
```

When you call a procedure, you pass an argument for each parameter. You can specify the list of arguments for a VB procedure in one of two ways: by position or by name. For example

```
Response.Write(Divide(10,2))
Response.Write("<BR>")
Response.Write(Divide(Denominator:=10, Numerator:=20))
```

The above would produce the following output:

```
5
2
```

Although it takes a little more time, your code will be better documented when you pass arguments by name.

By Value Versus By Reference

Unlike prior versions of VB, by default, VB.NET passes arguments to subroutines and functions *by value*, which is a change from earlier versions of VB. This means that VB sends a copy of each argument's value to the procedure. It also means that parameters, by default, can only be used for input.

You can override this default behavior by using the `ByRef` keyword. When you use `ByRef`, VB.NET sends a pointer, or reference, to each parameter to the procedure rather than a copy of the parameter's value. Thus, you can use `ByRef` parameters to pass information back to the code that called the procedure.

Optional Parameters

VB.NET supports optional parameters. To create an optional parameter you insert the `Optional` keyword before the parameter name and you supply the parameter's default value after the data type, like this:

```
Optional parameter_name As data_type = default_value
```

The following function takes a string and makes it into an HTML heading of a level specified by the `Level` parameter. If `Level` is not specified, it is assumed to be 1:

```
Function CreateHead(Msg As String, Optional Level As Integer = 1) As String
    Dim ReturnMsg As String
    Dim HLevel As String

    If Level >= 1 And Level <=6 Then
        HLevel = Level.ToString()
        ReturnMsg = "<H" & HLevel & ">" & Msg & "</H" & HLevel & ">"
    Else
        ReturnMsg = "<P>" & Msg & "</P>"
    End If

    Return ReturnMsg
End Function
```

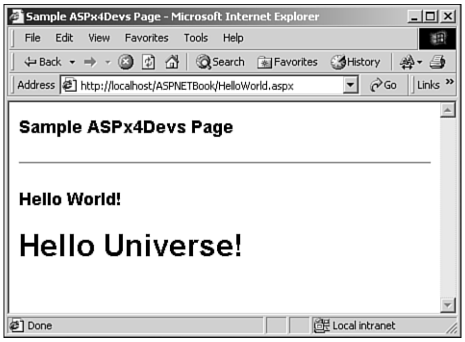
NOTE

The `If` statement will be discussed later in the chapter. In addition, the `ToString` method, which you can use to convert a number into a string, will be discussed in Chapter 5, "Working with Numbers, Strings, Dates, and Arrays in Visual Basic.NET."

The `HelloWorld.aspx` page calls `CreateHead` twice from the `Page_Load` subroutine:

```
<script language="VB" runat="server">
' ...
Sub Page_Load(Src as Object, E as EventArgs)

    If Not Page.IsPostBack Then
        DisplayMsg.Text = CreateHead("Hello World!", 3)
        DisplayMsg.Text &= CreateHead("Hello Universe!")
    End If

End Sub
' ...
</script>

<asp:label id="DisplayMsg" runat="server" />
```

The first time the code calls `CreateHead` with the phrase "Hello World" and `Level` is equal to 3. The second time the code calls `CreateHead` with the phrase "Hello Universe" and `Level`is not specified, which is interpreted to mean a heading level of 1. This produces a page like the one shown in Figure 4.1.


Sample ASPx4Devs Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
History
Address http://localhost/ASPNETBook/HelloWorld.aspx
Go
Links
Done
Local intranet

Sample ASPx4Devs Page

Hello World!

Hello Universe!

FIGURE 4.1
This sample page illustrates the use of optional parameters.

Every parameter to the right of an optional parameter must also be optional.

Using Branching and Looping Structures

More than likely, you'll want to be able to conditionally branch in your code based on the value of a variable or an expression. Or perhaps you'll want to repeatedly loop through a section of code. VB.NET supports several branching and looping structures.

Branching in VB.NET

You can branch in your code using the `If...Then...Else` statement or the `Select Case` statement.

The `If...Then...Else` Statement

You use the `If...Then...Else` statement (or simply the `If` statement) to conditionally execute a piece of code based on the value of some expression. The simplest form of the `If` statement contains an `If` clause without any `Else` clause. Such a statement was used in an earlier example (from `HelloWorld.aspx`):

```
If Not Page.IsPostBack Then
    DisplayMsg.Text = CreateHead("Hello World!", 3)
    DisplayMsg.Text &= CreateHead("Hello Universe!")
End If
```

In this example, the two assignment statements are executed only when `Page.IsPostBack` is `False`. Otherwise, no statements are executed.

NOTE

The `Page` object and the IsPostBack property are discussed in Chapter 7, “Introducing ASP.NET Web Forms.”

This example, also shown earlier in the chapter, includes an `Else` clause. Any statements in the `Else` block are executed when the `If` expression is `False`:

```
If Level >= 1 And Level <=6 Then
    HLevel = Level.ToString()
    ReturnMsg = "<H" & HLevel & ">" & Msg & "</H" & HLevel & ">"
Else
    ReturnMsg = "<P>" & Msg & "</P>"
End If
```

An `If` statement may also contain one or more `Else If` clauses that allow you to test additional scenarios as shown in this example:

```
If DateTime.Today <= #12/31/2000# Then
    RegPrice = 895
ElseIf DateTime.Today <= #01/31/2001# Then
    RegPrice = 995
ElseIf DateTime.Today <= #02/28/2001# Then
    RegPrice = 1095
```

```
Else
    RegPrice = 1195
End If
```

The `Select...Case` Statement

You can also use the `Select...Case` statement for branching in VB.NET. The `Select...Case` statement is useful when you wish to check the value of an expression against a list of possible values and execute a different set of code for each value.

This example checks the value of the integer variable `PayMethod` against a list of possible values. The `Select...Case` statement sets the value of two string variables to various values depending on the value of `PayMethod`.

```
Select Case PayMethod
Case 1
    PayMethText = "Visa"
    SubmitText = "Complete Order and Bill My Credit Card"
Case 2
    PayMethText = "Mastercard"
    SubmitText = "Complete Order and Bill My Credit Card"
Case 3
    PayMethText = "American Express"
    SubmitText = "Complete Order and Bill My Credit Card"
Case 4
    PayMethText = "Company PO"
    SubmitText = "Complete Order"
Case 5
    PayMethText = "Check"
    SubmitText = "Complete Order"
Case Else
    PayMethText = "Error"
    SubmitText = "Illegal Payment Method: please correct."
End Select
```

Notice the `Case Else` clause, which is executed if none of the other cases is true.

Looping in VB.NET

You can loop using the `Do...Loop` statement, the `While...End While` statement, the `For...Next` statement, or the `For...Each` statement.

The Do...Loop Statement

You can use the `Do...Loop` statement (or simply `Do` loop) to execute a set of statements repeatedly, either while some condition is true or until some condition becomes true.

For example, the following code from `titles.aspx` fills a dropdownlist control with records from the titles table of the pubs sample SQL Server database. We have used a `Do` loop to move through each of the records returned by the query and added them to the dropdownlist's `ListItem` collection. (See Chapter 10, "Designing Advanced User Interfaces with Web For List Controls and Custom Web Controls," for more on Web Form list controls and Chapter 15, "Accessing SQL Server Data with the SQL Managed Provider," for more on using ADO.NET with the SQL Managed Provider.)

```
Sub FillList()
    Dim ConnectString As String = _
     "server=localhost;uid=sa;pwd=;database=pubs"
    Dim SQL As String
    Dim PubsCnx As SQLConnection
    Dim TitlesQry As SQLCommand
    Dim TitlesRdr As SQLDataReader
    Dim TitleItem As ListItem

    PubsCnx = New SQLConnection(ConnectString)
    PubsCnx.Open()
    SQL = "SELECT title, title_id FROM titles ORDER BY title"
    TitlesQry = New SQLCommand(SQL, PubsCnx)
    TitlesQry.Execute(TitlesRdr)

    Do While TitlesRdr.Read()
        TitleItem = New ListItem(TitlesRdr("title"), TitlesRdr("title_id"))
        TitleList.Items.Add(TitleItem)
    Loop
End Sub
```

The `Do...Loop` statement from the `FillList` subroutine uses the `SQLDataReader`'s `Read` method to retrieve the next record returned by the query. `Read` advances the current record pointer and returns `True` if it was able to successfully retrieve a record or `False` if there are no more records to retrieve.

This `Do...Loop` statement could have been rewritten by reversing the logic of the condition:

```
Do Until Not TitlesRdr.Read()
    TitleItem = New ListItem(TitlesRdr("title"), TitlesRdr("title_id"))
    TitleList.Items.Add(TitleItem)
Loop
```

The VB.NET `Do...Loop` statement also supports testing the condition at the bottom of the loop. For example, the following loop squares itself until the result exceeds 100:

```
Dim x As Integer = 2
Do
    x = x*x
Loop Until x > 100
```

You can use the `Exit Do` statement to terminate a `Do` loop early. For example, the following code (from `DotSearch.aspx`) searches for the position of the first period in a string, exiting the `Do` loop when a period is found:

```
Dim Phrase As String = "My mother is very, very smart. My father is too."
Dim Length As Integer = Len(Phrase)
Dim i As Integer = 1

Do While i<=Length
    If Phrase.Chars(i) = "." Then
        Exit Do
    End if
    i += 1
Loop
' Add one to i because the first character
' in a string is character 0 in VB.
Label1.Text = "The first period in '" & Phrase & _
 "' was found at character " & i + 1 & "."
```

NOTE

This example was provided for demonstration purposes only. You should use the .NET Framework `String.IndexOf` method to search for strings within other strings.

The `While...End While` Statement

The `While...End While` statement (or simply the `While` loop) is very similar to the `Do...Loop` statement. You can use it to execute a set of statements repeatedly, while some condition is true. For example

```
While i<=Length
    If Mid(Phrase, i, 1) = "." Then
        Exit While
    End if
    i += 1
End While
```

As shown in this example, you can use the `Exit While` statement to terminate a While loop early.

NOTE

The `While...Exit While` statement replaces the old `While...Wend` statement of earlier versions of VB.

The `For...Next` Statement

You can use the `For...Next` statement (or simply the `For` loop) to repeatedly execute a block of statements a specified number of times. While similar in concept to the `Do` and `While` loops, the `For` loop differs in that it automatically increments a counter variable for you.

The `For` loop is especially useful for iterating through the items in an array. For example, the following code iterates through all of the elements in the `Colors` array and displays them on the page:

```
For i = 0 To UBound(Colors)
    Response.Write("<br />" & i & "=" & Colors(i))
Next
```

The `For...Each` Statement

The `For...Each` statement is a special kind of `For...Next` loop that is useful for iterating through members of a collection. A collection is an ordered set of items, usually objects, that you can refer to and manipulate as a unit. For example, when working with ADO.NET, you can work with the `Errors` collection of `Error` objects.

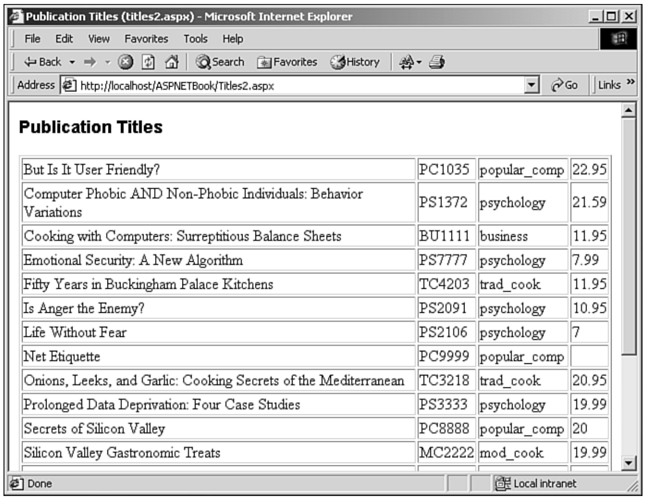
NOTE

You can use the `Collection` object in VB.NET to create your own collections.

For example, the following function (from `titles2.aspx`) returns an HTML table containing a row for each record in the titles table of the SQL Server Pubs database. The `DataSet`'s `Table` object contains a collection of rows and each row contains a collection of columns. The `DisplayTitles` function employs two nested `For...Each` loops to iterate through the `TitlesSet` dataset. (Datasets and ADO.NET are explained in more detail in Chapter 15 and Chapter 16, "Accessing Non-SQL Server Data with the OLE DB Managed Provider.")

```
Function DisplayTitles()
    Dim ConnectString As String = _
     "server=localhost;uid=sa;pwd=;database=pubs"
    Dim SQL As String
    Dim PubsCnx As SQLConnection
    Dim TitlesQry As SQLDataSetCommand
    Dim TitlesSet As DataSet
    Dim TitleItem As ListItem
    Dim row As DataRow
    Dim col As DataColumn
    Dim Msg As String
    PubsCnx = New SQLConnection(ConnectString)
    PubsCnx.Open()
    SQL = "SELECT title, title_id, type, price FROM titles ORDER BY title"
    TitlesQry = New SQLDataSetCommand(SQL, PubsCnx)
    TitlesSet = New DataSet
    TitlesQry.FillDataSet(TitlesSet, "titles")

    Msg &= "<TABLE border=""1"">"
    For Each row In TitlesSet.Tables("titles").Rows
        Msg &= "<TR>"
        For Each col In TitlesSet.Tables("titles").Columns
            Msg &= "<TD>" & FixNull(row(col), "&nbsp;") & "</TD>"
        Next
        Msg &= "</TR>"
    Next
    Msg &= "</TABLE>"
    Return Msg
End Function
```

The `titles2.aspx` page is shown in Figure 4.2.


Publication Titles (titles2.aspx) - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost/ASPNETBook/Titles2.aspx
Go Links
Done
Local intranet

Publication Titles

But Is It User Friendly?	PC1035	popular_comp	22.95
Computer Phobic AND Non-Phobic Individuals: Behavior Variations	PS1372	psychology	21.59
Cooking with Computers: Surreptitious Balance Sheets	BU1111	business	11.95
Emotional Security: A New Algorithm	PS7777	psychology	7.99
Fifty Years in Buckingham Palace Kitchens	TC4203	trad_cook	11.95
Is Anger the Enemy?	PS2091	psychology	10.95
Life Without Fear	PS2106	psychology	7
Net Etiquette	PC9999	popular_comp	
Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean	TC3218	trad_cook	20.95
Prolonged Data Deprivation: Four Case Studies	PS3333	psychology	19.99
Secrets of Silicon Valley	PC8888	popular_comp	20
Silicon Valley Gastronomic Treats	MC2222	mod_cook	19.99

FIGURE 4.2
Nested `For...Next` *loops are used to display titles from the Pubs database.*

Creating Objects

Prior versions of VB lacked many object-oriented programming (OOP) features that other languages such as C++, Java, and FoxPro have had for years. Fortunately, VB.NET includes strong support for OOP.

OOP Primer

Class, subclass, inheritance, constructor, polymorphism: Object-oriented programming uses lots of fancy new terms that undoubtedly confuse the non-OOP programmer. In this section, you'll find a 10-minute primer of OOP terminology.

Objects and Classes

Objects are things that you want to represent in your code. Another way to think of an object is as a grouping of properties, methods, and events that are logically tied together. You work with an object by manipulating its properties and methods and reacting to its events.

A *class* is a template or schema for creating an object. At design time you create the class that serves as the template for creating objects at runtime. An object is thus an instance of a class. And that's one of the neat things about using classes: You can have as many instances of a class as you want, and VB automatically keeps each object's data independent of each other object's data. Another neat thing about classes is that they encapsulate the implementation of the object into a neat package. *Encapsulation* allows you to separate the implementation of the class (the code inside of the class that makes it work) from its interface (the public properties, methods, and events of the class).

Inheritance and Polymorphism

One of the big additions to VB.NET is its support for inheritance. *Inheritance* allows you to create classes that are descendants of another class. When a class inherits from another class, the original class is termed the base class(also sometimes referred to as the superclass or parent class) and the class that inherits from the base class is called the *derived class* (also sometimes referred to as the subclass or child class).

VB.NET supports the *overriding* of a base class's methods with alternate implementations. *Polymorphism* is the ability of different classes to support the properties and methods with the same name but with different implementations. VB.NET's support for overriding allows your classes to support polymorphism.

Creating a Class

You use the `Class` statement in VB.NET to create a class. Public variables of the class become properties of the class and public subroutines or functions of the class become methods.

Let's illustrate with an example: Say that your Web site supports an ASP.NET conference called "ASP.NET—The Conference." To simplify the registration process, you decide to create a `Register` class. The `Register` class will have two properties that describe the attendee, `CustomerFirstName` and `CustomerLastName`, and two methods, `Execute` and `Cancel`:

```
Class Register
    Public CustomerFirstName As String
    Public CustomerLastName As String

    Public Function Execute() As String
        Return "<BR /><font color=""green"">Register.Execute</font>"
    End Function

    Public Function Cancel() As String
        Return "<BR /><font color=""red"">Register.Cancel</font>"
    End Function
End Class
```

To create an object from a class, you use the `New` operator just like when creating VB and .NET Framework objects. The following code declares `Reg` as a member of the `Register` class and instantiates `Reg`:

```
Dim Reg As Register

Reg = New Register
```

The above can be simplified as follows:

```
Dim Reg As Register = New Register
```

The following code sets two properties of the `Reg` object:

```
Reg.CustomerFirstName = "Bill"
Reg.CustomerLastName = "Gates"
```

The following code invokes the `Execute` method of `Reg` setting the `Text` property of a label control to its return value:

```
Label1.Text = Reg.Execute()
```

While you could also have programmed the registration process with a series of functions and subs, the beauty of using classes is that you encapsulate all of the details associated with registering an attendee, wrapping it all up into a class with a very simple interface. Of course, either way, you have to write the code that does the work. But by coding the registration process using classes, you benefit from improved reusability, manageability, and extensibility.

Using Property Procedures

There are actually two ways to create properties of classes in VB.NET. You've already seen one way to create properties: create public variables of the class module. A second way to create properties is to use the `Property` statement. Using a `Property` statement to create properties offers several advantages over using public variables. Using a `Property` statement lets you

- Create read-only and write-only properties
- Run code when a property value is read or written

The `Property` statement must follow a certain format. For read/write properties, it should look like this:

```
Property property_name As data_type
    Get
        property_name = some_value
    End Get
    Set
        some_variable = Value
    End Set
End Property
```

The `Value` keyword within a `Set` block retrieves the value to which the consumer of the class has set the property.

Read-only properties must include the `ReadOnly` keyword and have no `Set` block:

```
ReadOnly Property property_name As data_type
    Get
        property_name = some_value
    End Get
End Property
```

Write-only properties must include the `WriteOnly` keyword and have no `Get` block:

```
WriteOnly Property property_name As data_type
    Set
        some_variable = Value
    End Set
End Property
```

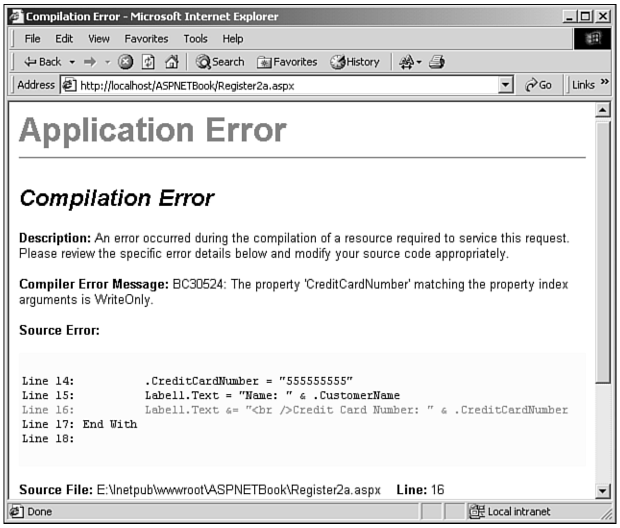
The following class (from `Register2.aspx`) uses `Property` statements to create four properties. The `CustomerFirstName` and `CustomerLastName` properties are read/write. `CustomerName` is read-only and `CreditCardNumber` is write-only:

```
Class Register
    Private CustFname As String
    Private CustLName As String
    Private FullName As String
    Private CreditCard As String

    Public Property CustomerFirstName As String
        Get
            CustomerFirstName = CustFName
        End Get
        Set
            CustFName = Value
            FullName = CustFName & " " & CustLName
        End Set
    End Property

    Public Property CustomerLastName As String
        Get
            CustomerLastName = CustLName
        End Get
        Set
            CustLName = Value
            FullName = CustFName & " " & CustLName
        End Set
    End Property

    Public ReadOnly Property CustomerName As String
        Get
            CustomerName = FullName
        End Get
    End Property

    Public WriteOnly Property CreditCardNumber As String
        Set
            CreditCard = Value
        End Set
    End Property

    Public Function Execute() As String
        Return "<BR /><font color=""green"">Register.Execute</font>"
    End Function

    Public Function Cancel() As String
        Return "<BR /><font color=""red"">Register.Cancel</font>"
    End Function
End Class
```

NOTE

The methods in these demonstration examples do nothing but return some text.

You'll get a compile error if you try to write to a read-only property or read a write-only property. The `Register2a.aspx` page illustrates what happens when trying to read a write-only property (see Figure 4.3).


Compilation Error - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost/ASPNETBook/Register2a.aspx
Go Links
Done
Local intranet

Application Error

Compilation Error

Description: An error occurred during the compilation of a resource required to service this request. Please review the specific error details below and modify your source code appropriately.

Compiler Error Message: BC30524: The property 'CreditCardNumber' matching the property index arguments is WriteOnly.

Source Error:

```
Line 14:          .CreditCardNumber = "555555555"
Line 15:          Label1.Text = "Name: " & .CustomerName
Line 16:          Label1.Text &= "<br />Credit Card Number: " & .CreditCardNumber
Line 17: End With
Line 18:
```

Source File: E:\Inetpub\wwwroot\ASPNETBook\Register2a.aspx **Line:** 16

FIGURE 4.3
Attempting to read a write-only property causes a compilation error.

NOTE

In prior versions of VB you used `Property` procedures instead of the `Property` statement to create properties.

Inheritance

VB.NET allows you to create classes that derive from other classes using the `Inherits` statement.

Let's say that you decide to branch out and now run a conference on XML in addition to ASP.NET, but the registration process for XML is different enough that you'll need to make some changes to the process. To handle the new show, you create two subclasses of `Register`: `ASPRegister` and `XMLRegister`. For `XMLRegister` you add an additional property, `UserGroupAffiliation`, because the XML conference will give special discounts to qualified user group members. Without inheritance you'd have to either change the base class or copy and paste code from class to class. Inheritance, however, allows you to start with the existing `Register` class and extend it with additional properties and methods. But there's no need to re-create the properties and methods that `XMLRegister` has in common with the base class.

In the following code from `Register3.aspx`, the `ASPRegister` and `XMLRegister` classes are derived from the `Register` base class:

```
Class Register
    Public CustomerFirstName As String
    Public CustomerLastName As String
    Public CreditCardNumber As String
    Public CreditCardExpires As String

    Public Function Execute() As String
        Return "<BR /><font color=""green"">Register.Execute</font>"
    End Function

    Public Function Cancel() As String
        Return "<BR /><font color=""red"">Register.Cancel</font>"
    End Function
End Class

Class ASPRegister
    Inherits Register
End Class

Class XMLRegister
    Inherits Register
    Public UserGroupAffiliation As String
End Class
```

The following code from `Register3.aspx` instantiates both `ASPRegister` and `XMLRegister` objects:

```
Sub Page_Load(Src as Object, E as EventArgs)
    Dim ASPReg As ASPRegister
    Dim XMLReg As XMLRegister

    ASPReg = New ASPRegister
    With ASPReg
```

```
        .CustomerFirstName = "Bill"
        .CustomerLastName = "Gates"
        .CreditCardNumber = "55555555555"
        .CreditCardExpires = "01/02"
        Label1.Text = .Execute()
    End With

    XMLReg = New XMLRegister
    With XMLReg
        .CustomerFirstName = "Steve"
        .CustomerLastName = "Balmer"
        .CreditCardNumber = "55555555556"
        .CreditCardExpires = "05/03"
        .UserGroupAffiliation = "Microsoft Windows 2000 UG"
        Label2.Text = .Execute()
    End With

    ASPReg = Nothing
    XMLReg = Nothing
End Sub
```

When you're done using an object, you should explicitly destroy it by setting it to the VB.NET keyword of `Nothing`.

Accessibility of Inherited Properties and Methods

When creating classes you use the `Public` keyword to create publicly exposed properties and methods of the object. If you don't want a class method or property to be public, however, you have a few other choices: `Private`, `Protected`, `Friend`, or `Protected Friend`.

Use the `Private` keyword to make a property or method private to the class. Private properties and methods can be accessed anywhere inside of the class, but they can't be accessed by derived classes or other classes on the page or assembly (assemblies are discussed in Chapter 6, "Using Assemblies in Visual Basic.NET").

You use the `Protected` keyword to create a property or method that is accessible from any code inside the class and from code inside of any derived classes.

Use the `Friend` keyword to create a property or method that is accessible from any code—including derived classes—in the current page or assembly. Friend properties and methods are not accessible from derived classes that live in other assemblies.

Finally, you can use the `Protected Friend` keyword to create a property or method that's available from any code in the current page or assembly *and* from any derived classes, no matter where they reside.

Overriding Methods

If you wish to override a method of the base class, you use the `Overrides` keyword.

Let's say you add a third conference on Linux, but this conference is different from the other two because registration is free. So you create a third subclass, `LinuxRegister`, but this time you wish to change the Execute method so you override the `Execute` method from the base class (`Register`) and replace the implementation of `Execute` with a new implementation that is custom tailored for the `Linux` show.

Here's the code from `Register4.aspx` that overrides the `Register.Execute` method:

```
Class LinuxRegister
    Inherits Register
    Overrides Public Function Execute() As String
        Return "<BR /><font color=""blue"">Register.Execute</font>"
    End Function
End Class
```

This code won't work, however, unless the `Execute` method is marked as an overridable method back in the base class. To do that, add the `Overridable` keyword to the declaration of the method in the base class. Here's the `Register` class from `Register4.aspx`:

```
Class Register
    Public CustomerFirstName As String
    Public CustomerLastName As String
    Public CreditCardNumber As String
    Public CreditCardExpires As String

    Overridable Public Function Execute() As String
        Return "<BR /><font color=""green"">Register.Execute</font>"
    End Function

    Overridable Public Function Cancel() As String
        Return "<BR /><font color=""red"">Register.Cancel</font>"
    End Function
End Class
```

Notice that we have marked the `Execute` and `Cancel` functions as `Overridable`. Methods in VB.NET, by default, are *not* overridable (you can also explicitly include the `NotOverridable` keyword when you define the method but it's not necessary).

If you wish to access a base method from a derived method, you can use the `MyBase` psuedo-object. For example, if you wished to call the `Register` class's `Execute` function from the `LinuxRegister Execute` method, you would use code like this:

```
Class LinuxRegister
    Inherits Register
    Overrides Public Function Execute() As String
        MyBase.Execute()
    End Function
End Class
```

Constructors and Destructors

VB.NET lets you create subroutines that are executed when an object is instantiated or destroyed.

A *class constructor* is called when an object is instantiated. To create a constructor, you add a subroutine named `New` to the class definition (you can place it anywhere in the class definition). The class constructor subroutine will be called when an object is instantiated before any other code in the class is executed. Class constructors may have parameters; these types of constructors are called parameterized constructors. *Parameterized constructors* allow you to pass parameters to a class when the object is instantiated.

A *class destructor* is called when an object is destroyed. To create a destructor, you create a subroutine in the class definition named `Destruct` (you can place it anywhere in the class definition). Unlike the `Class_Terminate` method of VB 6.0 that was always executed immediately when the object was destroyed, you cannot be sure exactly when your class destructor code will execute by the .NET Framework garbage collector. The only thing you can be sure of is that it will occur sometime *after* you release all references to the object.

The `Register` class from `Register5.aspx` contains a parameterized class constructor that you can use to optionally set the customer name when instantiating the class:

```
Sub New(Optional LastName As String = "", Optional FirstName As String = "")
    CustLName = LastName
    CustFName = FirstName
    FullName = CustFName & " " & CustLName
End Sub
```

Here's code from the two derived classes that also contain class constructors:

```
Class ASPRegister
    Inherits Register

    Sub New(Optional LastName As String = "", Optional FirstName As String =
"")
        MyBase.New(LastName, FirstName)
    End Sub
End Class
```

```
Class XMLRegister
    Inherits Register

    Public UserGroupAffiliation As String

    Sub New(Optional LastName As String = "", Optional FirstName As String =
"")
        MyBase.New(LastName, FirstName)
        UserGroupAffiliation = "none"
    End Sub
End Class
```

Notice the use of the `MyBase.New`. Derived classes need to explicitly call the `MyBase.New` constructor or the base class constructor code will not run.

Here's the code from `Register5.aspx` that instantiates the `ASPRegister` and `XMLRegister` objects, passing along the optional customer name information to the objects' constructors:

```
ASPReg = New ASPRegister("Gates", "Bill")
XMLReg = New XMLRegister("Balmer", "Steve")
```

Summary

This chapter introduced the Visual Basic.NET language and discussed its core features, including operators, procedures, variables, parameters, data types, constants, and branching and looping statements. The chapter also introduced object-oriented programming (OOP) concepts and VB.NET OOP features, including classes, properties, methods, inheritance, and overriding.

VB.NET is an exciting new release of the venerable Visual Basic language, and a language that is now on equal footing when compared to Visual C++ and C#. Microsoft has added a number of new features to the language that will undoubtedly be appreciated by the vast majority of existing VB and ASP programmers who will quickly forget about VB 6.0 and VBScript.