CHAPTER 1

Laying Your Visual Basic .NET Foundation

TIPS IN THIS CHAPTER

Over the past ten years, Visual Basic has emerged as the programming language of choice for the vast majority of programmers. Many programmers site Visual Basic's ease of use as the key to its success. Others claim that the ability to drag and drop controls onto a form to quickly build a program's user interface lead to Visual Basic's widespread use.

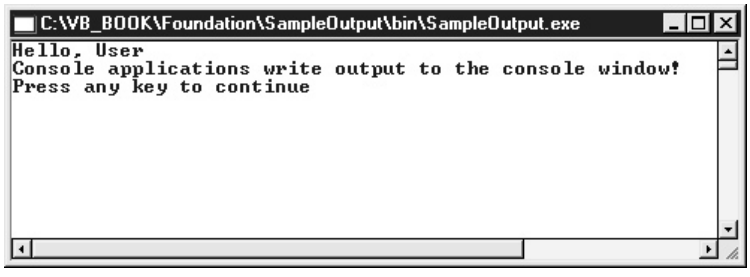
While masses of programmers have used Visual Basic to implement solutions for a wide range of programming tasks, a large group of developers, many of whom have been programming with languages such as C and C++ for years, have refused to acknowledge Visual Basic's suitability as a professional programming language. Many such programmers have stated that while Visual Basic provides a convenient way to build a prototype, programmers should later rewrite the code using a language such as C++ to achieve better performance.

With the release of the .NET environment, Microsoft has included two new key programming languages, Visual Basic .NET and C# (Microsoft also included Visual C++ .NET as a part of the .NET environment). As you will learn, the .NET environment provides programming-language independent classes and routines that are used by both C# and Visual Basic .NET. This means whether a programmer is using Visual Basic .NET or C#, the programmer has the same .NET capabilities available for use. From a performance perspective, Visual Basic .NET applications will run neck and neck with identical programs written using C#. Although the .NET environment provides C# programmers with the ability to drag and drop controls onto a form to quickly build a user interface, the sheer number of Visual Basic programmers who migrate to Visual Basic .NET will make Visual Basic .NET the .NET programming language of choice.

Throughout this book's 18 chapters, you will examine Visual Basic .NET and the .NET environment in detail. This chapter exists to provide programmers who are new to Visual Basic with a foundation from which they can build their knowledge and understanding of the capabilities Visual Basic .NET and the .NET environment provide. If you are an experienced Visual Basic programmer, you may want to simply scan the titles of the Tips this chapter provides in search of topics you may find new and then turn to the chapter's final Tip, which summarizes key differences between Visual Basic and Visual Basic .NET. In either case, it's time to get started.

Creating Your First Console Application

Using Visual Basic .NET, you can create a variety of application types, such as a console-based program that displays its output in an MS-DOS-like window, as shown here, a Windows-based program that often displays a form-based interface, an ASP.NET page, and more.



Because of the console-based application's ease of use (you can quickly create programs to display simple output without having to place controls onto a form), many of the Tips this book presents use console-based applications.

USE IT To create a console application using Visual Studio, perform these steps:

1. In Visual Studio, select File | New | Project. Visual Studio will display the New Project dialog box.

2. In the New Project dialog box, click the Console Application icon. In the Name field, type a project name that describes the program you are building, such as DemoProgram (do not type an extension). Then, in the Location field, type the name of the folder in which you want Visual Studio to place the project's folder and files. Click OK. Visual Studio will display a code window, as shown in Figure 1-1, in which you can type your program statements.

   In the code window, type the following statements in the Main subroutine:

```
Sub Main()
   Console.WriteLine("My first VB.NET Program!")
   Console.ReadLine()
End Sub
```

Next, to run the program, select Debug | Start. Visual Studio will display your program's output in a console window. To end the program, which will direct Visual Studio to close the window, press ENTER.

As you type program statements in a Visual Basic .NET program, you must type the statements exactly as they appear in this book, making sure you include the quotes, commas, periods, and so on. Otherwise, your programs may violate one or more of the Visual Basic .NET syntax rules (the rules that define the language structure and the format you must use as you create programs). When a syntax

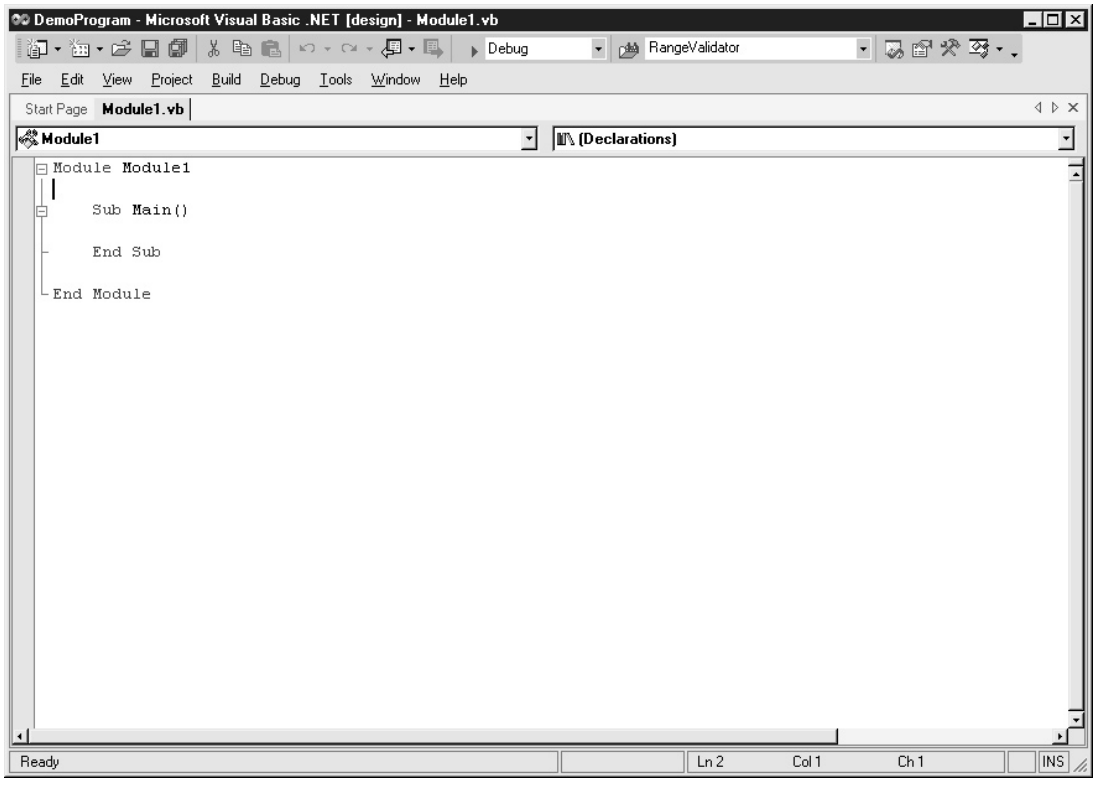


Figure 1-1 Displaying the code window in Visual Studio

error occurs, Visual Studio will display an error message that describes the error and the line number in your code where the error occurs. Before Visual Studio will build your program, you must locate and correct the syntax error.

Each time you make a change to your program code, you must direct Visual Studio to rebuild your program to put your change into effect. To rebuild your program, select Build | Build Solution. For example, in the previous program statements, change the code to display the message "Hello, user" by changing the Console.WriteLine method as follows:

```
Console.WriteLine("Hello, user")
```

Next, rebuild your program and then select Debug | Start to view your new output.

Building a Windows-Based Application

Using Visual Studio, you can create a variety of application types. Normally, to create a Windows-based application, programmers will drag and drop one or more controls onto a form to provide the user interface, as shown in Figure 1-2.

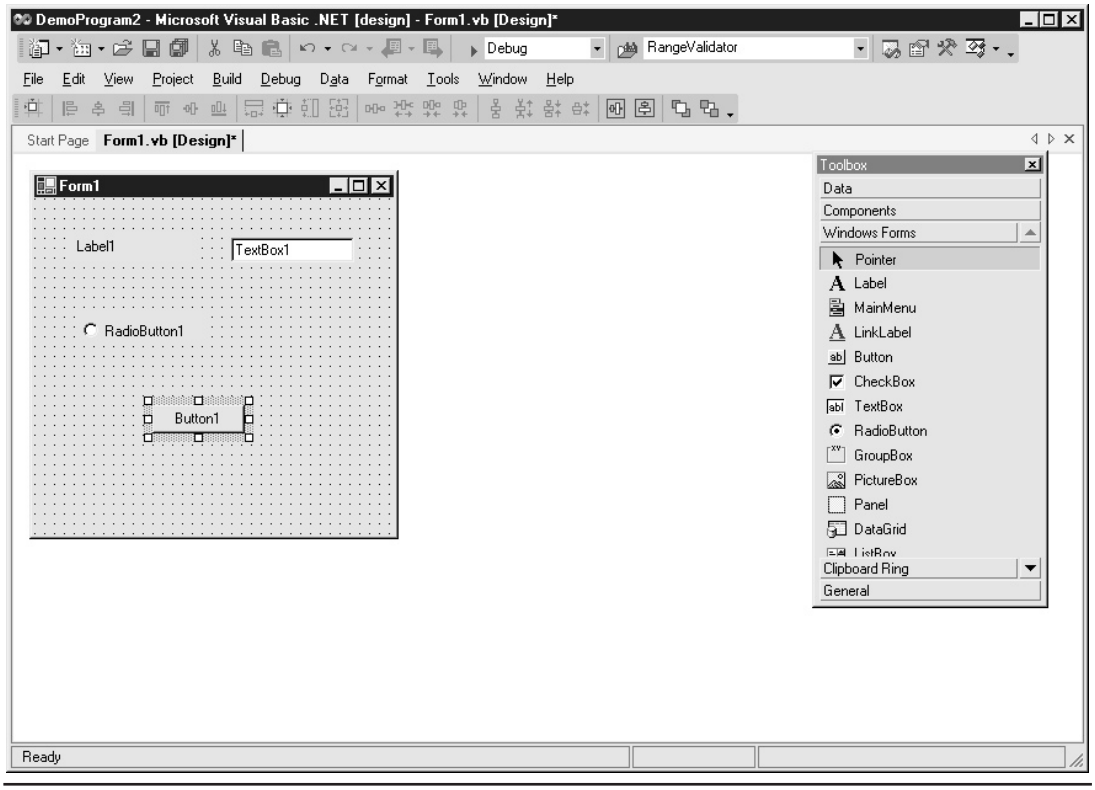


Figure 1-2 Placing controls on a form in Visual Studio to build a Windows-based application

USE IT Chapter 11 examines in detail the controls you can place onto a form. To create a simple Windows-based application using Visual Studio, perform these steps:

1. In Visual Studio, select File | New | Project. Visual Studio will display the New Project dialog box.
2. In the New Project dialog box, click the Windows Application icon. In the Name field, type a project name that describes the program you are building, such as DemoProgram (do not type an extension). Then, in the Location field, type the name of the folder in which you want Visual Studio to place the project's folder and files. Click OK. Visual Studio will display a design window, similar to that previously shown in Figure 1-3, in which you can drag and drop controls onto your form.
3. To display the toolbox that contains the controls you can drag and drop onto the form, select View | Toolbox. Visual Studio will open the Toolbox window.
4. In the Toolbox window, locate the Label control. Drag and drop the control onto the form.
5. In the form, right-click the Label control and choose Properties. Visual Studio will display the Label control's properties in the Properties window.
6. In the Properties window, locate the Text property and type **Hello, user**.

To build your program, select Build | Build Solution. Then, to run your program, select Debug | Start. Visual Studio, in this case, will display your program's form in its own window as shown in Figure 1-3.

If you make changes to the program's form, a control that resides on the form, or your program code, you must direct Visual Studio to rebuild your program to put your changes into effect.

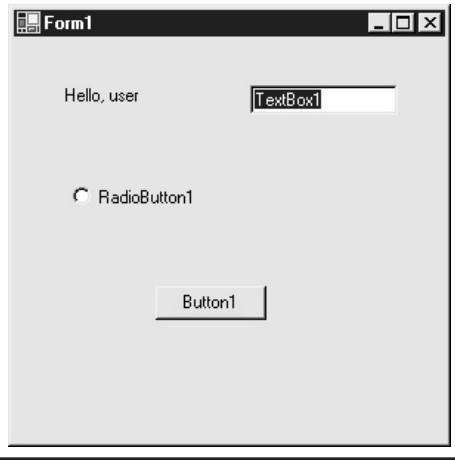


Figure 1-3 Creating and running a simple Windows-based application using Visual Studio

Choosing the Correct Visual Basic Types

To store information as they execute, programs place data into named storage locations that programmers refer to as variables—so named because the data a variable stores can change (vary) as the program executes. A program might use one variable to store a user's name, a second to store a user's e-mail address, and a third to store the user's age.

Each variable you create in your programs must be of a specific type. A variable's type defines the set of values a variable can store, such as counting or floating-point numbers (a number with a decimal point), or alphanumeric characters (such as the letters of the alphabet that make up a name). A variable's type also defines the set of operations a program can perform on a variable. It makes sense, for example, that a program can multiply two floating-point numbers, but it would not make sense to multiply two strings (such as two names). Table 1-1 briefly describes the Visual Basic data types. For each type, the table lists the range of values the type can store, along with the number of bytes of memory Visual Basic .NET must set aside to store the variable's value.

USE IT As you select a data type for use in your programs, choose a type that best matches your data. Assume your program must store values in the range –10,000 to 20,000. The Visual Basic .NET data types Integer, Long, and Short can each store values in this range. However, by using the type Short, your programs will allocate less memory to store the value—which means your program can store and retrieve the value faster than it could with a larger data type. More importantly, however, by selecting the Short type, you provide another programmer who reads your code with insight into the variable's use. In the case of a variable defined as the type Short, another programmer who reads your code immediately knows the variable will store values in the range –32,768 to 32,767. Visual Basic .NET has "retired" (no longer supports) the Currency and Variant data types that existed in previous versions of Visual Basic.

Type	Values	Size
Boolean	Represents a True or False value.	2 bytes
Byte	Represents an 8-bit value in the range 0 to 255.	1 byte
Char	Represents a 16-bit Unicode character.	2 bytes
DateTime	Represents a date and time value.	8 bytes
Decimal	Represents a value with 28 significant digits in the range +/–79,228,162,514,264,337,593,543,950,335 with no decimal point to 7.9228162514264337593543950335 with 28 places to the right of the decimal point.	12 bytes
Double	Represents a floating-point value using 64 bits.	8 bytes
Integer	Represents a value in the range –2,147,483,648 to 2,147,483,647.	4 bytes
Long	Represents a value in the range –9,223,372,036,854,775,808 through 9,223,372,036,854,775,807.	8 bytes
Short	Represents a value in the range –32,678 to 32,677.	2 bytes
Single	Represents a floating-point value using 32 bits.	4 bytes

Table 1-1 The Visual Basic .NET Types

Declaring Variables in a Visual Basic .NET Program

Variables exist to let programs easily store and later retrieve information as the program executes. Programmers often describe a variable as "a named location in RAM" (a storage container) that holds a value. The size of the variable's storage container depends on the variable's type.

USE IT To declare a variable in a Visual Basic .NET program, you use the Dim statement to specify the variable's name and type. The Dim statement is so named because it specifies the dimensions of the variable's storage container. The following Dim statement declares a variable named EmployeeNumber of type Integer:

```
Dim EmployeeNumber As Integer
```

In this case, by declaring the variable as type Integer, you direct Visual Basic .NET to allocate 4 bytes to store the variable's value, for which the type Integer can be in the range –2,147,483,648 to 2,147,483,647. Often, programs must declare several variables at one time. The following statements declare variables to store an employee's name, ID, salary, and phone number:

```
Dim EmployeeName As String
Dim EmployeePhoneNumber As String
Dim EmployeeNumber As Integer
Dim EmployeeSalary as Double
```

In the previous variable declarations, the first two statements declare variables of type String, which can store alphanumeric characters. When you declare variables of the same type, Visual Basic .NET lets you place the declarations in the same statement, as shown here:

```
Dim EmployeeName, EmployeePhoneNumber As String
```

The following statement is equivalent to that just shown—each declares two String variables:

```
Dim EmployeeName As String, EmployeePhoneNumber As String
```

In general, you should consider declaring variables on individual lines, so you can place a comment to the right of the declaration that describes the variable's purpose:

```
Dim EmployeeName As String         ' Employee first and last name
Dim EmployeePhoneNumber As String  ' 10 digits in the form ###-###-####
```

When you name variables, choose names that meaningfully describe the information the variable stores. In that way, another programmer who reads your code can better understand the variable's purpose and your code's processing simply by reading the variable's name. If you consider the following variable declarations, the first variable's name provides you with the variable's implied use, whereas the second variable does not:

```
Dim CityName As String
Dim X As String
```

After you declare a variable in Visual Basic .NET, you can then use the assignment operator (the equal sign) to assign a value to the variable, as shown here:

```
CityName = "Houston"
EmployeeName = "Smith"
EmployeeSalary = 60000
```

Often, programmers must assign an initial value to a variable. To do so, some programmers choose to declare the variables on one line and then initialize the variable on the next, as shown here:

```
Dim EmployeeName As String         ' Employee first and last name
EmployeeName = "Bill Smith"

Dim EmployeePhoneNumber As String  ' 10 digits in the form ###-###-####
EmployeePhoneNumber = "281-555-1212"
```

Visual Basic .NET, however, lets you combine these two operations into one statement, as shown here:

```
Dim EmployeeName As String = "Bill Smith"

Dim EmployeePhoneNumber As String  = "281-555-1212"
```

Visual Basic .NET is a case-independent programming language, which means it treats upper- and lowercase letters the same in a variable name. The following statements each assign the value 50000 to the variable named EmployeeSalary:

```
EmployeeSalary = 50000
employeesalary = 50000
EMPLOYEESALARY = 50000
eMpLoYeEsAlArY = 50000
```

If you do not assign an initial value to a variable, Visual Basic .NET will initialize your variables for you during compilation. Visual Basic .NET will use the initial values listed in Table 1-2 based on the variable's type.

Type	Value
Boolean	False
Date	12:00:00AM
Numeric types	0
Object	Nothing

Table 1-2 Default Values Visual Basic .NET Assigns to Variables of Specific Data Types

The following program, DeclareVariables.vb, declares and initializes several variables. The program then displays each variable's value using the Console.WriteLine method:

```
Module Module1

    Sub Main()
        Dim EmployeeName As String
        Dim EmployeePhoneNumber As String
        Dim EmployeeSalary As Double
        Dim NumberOfEmployees As Integer

        EmployeeName = "Buddy Jamsa"
        EmployeePhoneNumber = "555-1212"
        EmployeeSalary = 45000.0
        NumberOfEmployees = 1

        Console.WriteLine("Number of employees: " & NumberOfEmployees)
        Console.WriteLine("Employee name: " & EmployeeName)
        Console.WriteLine("Employee phone number: " & EmployeePhoneNumber)
        Console.WriteLine("Employee salary: " & EmployeeSalary)

        Console.ReadLine() ' Pause to view output
    End Sub

End Module
```

After you compile and execute this program, your screen will display the following output:

```
Number of employees: 1
Employee name: Buddy Jamsa
Employee phone number: 555-1212
Employee salary: 45000
```

NOTE

In previous versions of Visual Basic, programmers used the LET statement to assign a value to a variable. Visual Basic .NET does not support the LET statement.

Displaying Screen Output Using Console.Write and Console.WriteLine

When you create a console-based application, your programs display their output to an MS-DOS window similar to that shown in Figure 1-4.

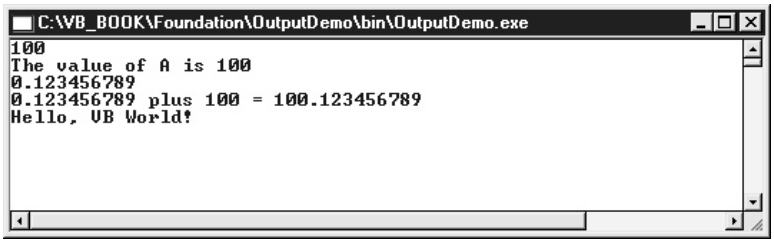


Figure 1-4 Displaying program output to a console window

To write output to the console window, your programs use the Console.Write and the Console.WriteLine methods. The difference between two methods is that Console.WriteLine will send a carriage-return and linefeed combination following your program output to advance the cursor to the start of the next line, whereas Console.Write will not.

To display a string message using Console.WriteLine, you simply pass the message as a parameter, placing the text in double quotes, as shown here:

```
Console.WriteLine("Hello, Visual Basic World!")
```

Similarly, to display a number or a variable's value, you pass the number or variable name to the method (without quotes) as shown here:

```
Console.WriteLine(1001)
Console.WriteLine(EmployeeName)
```

Often, programmers will precede a value with a text message, such as "The user's age is:". To display such output, the code will normally use the Visual Basic .NET concatenation operator (&) to append the value to the end of the string as shown here:

```
Console.WriteLine("The user's age is: " & UserAge)
```

USE IT The following program, OutputDemo.vb, illustrates the use of the Console.Write and Console.WriteLine methods:

```
Module Module1

    Sub Main()
        Dim A As Integer = 100
        Dim B As Double = 0.123456789
        Dim Message As String = "Hello, VB World!"

        Console.WriteLine(A)
        Console.WriteLine("The value of A is " & A)
        Console.WriteLine(B)
```

```
        Console.WriteLine(B & " plus " & A & " = " & B + A)
        Console.WriteLine(Message)

        Console.ReadLine()
    End Sub

End Module
```

After you compile and execute this program, your screen will display the following output:

```
100
The value of A is 100
0.123456789
0.123456789 plus 100 = 100.123456789
Hello, VB World!
```

Note that the program places the Console.ReadLine() statement as the last statement in the program. When you run a console application in Visual Studio, the console window will immediately close after the program completes its processing. By placing the Console.ReadLine() statement at the end of the code, the program will pause, waiting for the user to press the ENTER key, before the program ends and the window closes.

Formatting Program Output Using Console.WriteLine

In the previous Tips, you used the Console.WriteLine and Write functions to display messages to the console window. In each example, your application simply wrote the character string:

```
Console.WriteLine("Hello, world!")
```

Using the Console.WriteLine and Write functions, you can display values by using placeholders in the method's text output in the form {0}, {1}, and so on, and then passing parameters for each placeholder. The following statement uses placeholders in the Console.WriteLine function:

```
Console.WriteLine("The number is {0}", 3 + 7)
```

In this case, the WriteLine function will substitute the value 10 for the placeholder {0}. The following statement uses three placeholders:

```
Console.WriteLine("The result of {0} + {1} = {2}",  3, 7, 3+7)
```

In this case, the function will substitute the value 3 for the placeholder {0}, the value 7 for the placeholder {1}, and the value 10 for the placeholder {2}.

Using placeholders such as {0} and {1}, you can display values and variables using the Console.WriteLine and Write functions. When you specify placeholders, you must make sure

that you include values the function is to substitute for each placeholder. If you specify the placeholders {0} and {1}, for example, and only provide one value, the function will generate an exception. If your program does not handle the exception, your program will immediately end.

When your programs use the Console.WriteLine and Write functions to display output, your programs can place a format specifier after the placeholder number, such as {1, d} or {2, 7:f}. The format specifier can include an optional width value, followed by a colon and a character that specifies the value's type. Table 1-3 briefly describes the type specifiers.

In the previous Tip, you learned to use several different format specifiers in the Console.WriteLine and Console.Write functions. When your program displays floating-point values, there will be times, such as when the value represents currency, when you will want to specify the number of digits the functions display to the right of the decimal point. Assume your program must display the variable Amount, which contains the value 0.123456790. To control the number of digits the WriteLine and Write functions display, you specify the placeholder, width, format specifier, and number of digits, as shown here:

```
Console.WriteLine("See decimals {0, 12:f1}", _
   0.123456789)  ' 0.1
Console.WriteLine("See decimals {0, 12:f9}", _
   0.123456789)  ' 0.123456789
```

In addition to the width and format specifiers, the functions also let you use the pound sign (#) to format your data. The following statement directs Console.WriteLine to display a floating-point value with two digits to the right of the decimal point:

```
Console.WriteLine("The value is {0, 0:###.##}", Value)
```

Specifier	Value Type
C or c	Local currency format.
D or d	Decimal value.
E or e	Scientific notation.
F or f	Floating point.
G or g	Selects scientific or floating point depending on which is most compact.
N or n	Numeric formats which include commas for large values.
P or p	Percentage formats.
R or r	Called the "round-trip" specifier. Used for floating-point values to ensure that a value that is converted to a string and then back to floating point yields the original value.
X or x	Hexadecimal formats.

Table 1-3 Format Specifiers You Can Use with Console.WriteLine and Console.Write

When you use the pound-sign character to specify an output format, the WriteLine function will not display leading zeros. In other words, the function would output the value 0.123 as simply .123. When you want to display leading zeros, you can replace the pound sign with a zero, as shown here:

```
Console.WriteLine("The value is {0, 0:000.00}", Value)
```

The following program, ConsoleWriteLineDemo.vb, illustrates the use of the various Console.WriteLine formatting capabilities:

```
Module Module1

    Sub Main()
        Dim A As Double = 1.23456789
        Console.WriteLine("{0} {1} {2}", 1, 2, 3)
        Console.WriteLine("{0, 1:D} {1, 2:D} {2, 3:D}", 1, 2, 3)
        Console.WriteLine("{0, 7:F1} {1, 7:F3} {2, 7:F5}", A, A, A)
        Console.WriteLine("{0, 0:#.#}", A)
        Console.WriteLine("{0, 0:#.###}", A)
        Console.WriteLine("{0, 0:#.#####}", A)

        Console.ReadLine()
    End Sub

End Module
```

After you compile and execute this program, your screen will display the following output:

```
1 2 3
1  2   3
    1.2   1.235 1.23457
1.2
1.235
1.23457
```

Concatenating Characters to the End of a String

In Visual Basic .NET programs, the String type lets variables store alphanumeric characters (the upper- and lowercase letters of the alphabet, the digits 0 through 9, and punctuation symbols). To assign a value to a String variable, you must place the value in double quotes, as shown here:

```
Dim Name As String = "John Doe"
```

The following program, UseStrings.vb, assigns values to several different String variables, which the program later displays using Console.WriteLine:

```
Module Module1

    Sub Main()
        Dim Book As String = _
          "Visual Basic .Net Programming Tips & Techniques"
        Dim Author As String = "Jamsa"
        Dim Publisher As String = "McGraw-Hill/Osborne"

        Console.WriteLine(Book)
        Console.WriteLine(Author)
        Console.WriteLine(Publisher)

        Console.ReadLine()
    End Sub

End Module
```

After you compile and execute this program, your screen will display the following output:

```
Visual Basic .Net Programming Tips & Techniques
Jamsa
McGraw-Hill/Osborne
```

When your programs use String variables, a common operation your code will perform is to append characters to a String variable's existing contents. Programmers refer to operations that append characters to a String as concatenation operations.

USE IT To concatenate one string to another, you use the ampersand (&), which is the Visual Basic .NET concatenation operator. The following statement assigns an employee's first name and last name to a variable called EmployeeName, by concatenating the FirstName and LastName variables and assigning the result to the EmployeeName variable:

```
Dim FirstName As String = "Bill"
Dim LastName As String = "Gates"
Dim EmployeeName As String
EmployeeName = FirstName & " " & LastName
```

As you can see, the code separates the first and last names with a space by concatenating the space to the end of the string the FirstName variable contains. Throughout this book, you will encounter Console.WriteLine statements that use the concatenation operator to append a value to a string:

```
Console.WriteLine("The result is " & SomeVariable)
```

The following program, ConcatenateDemo.vb, illustrates the use of the concatenation operator:

```
Module Module1

    Sub Main()
        Dim WebSite As String
        Dim Publisher As String = "Osborne"

        Console.WriteLine("This book's publisher: " & Publisher)

        WebSite = "www." & Publisher & ".com"
        Console.WriteLine("View their books at " & WebSite)

        Console.ReadLine()
    End Sub

End Module
```

After you compile and execute this program, your screen will display the following output:

```
This book's publisher: Osborne
View their books at www.Osborne.com
```

Forcing Programs to Specify a Variable's Type

To reduce possible errors that result from misspelled variable names, you should force programs to declare each variable the code uses to store data. By declaring a variable, your program specifies the variable's type, which, in turn, limits the range of values the program can assign to the variable and the operations the program can perform on the variable.

The following program, BadVariables.vb, does not declare the variables it uses. The program assigns values to several employee-based variables and then displays the employee's name:

```
Option Explicit Off  ' Lets the program use variables without
                     ' declaring the variables
Module Module1

    Sub Main()
        EmployeeName = "Buddy Jamsa"
        EmployeePhoneNumber = "555-1212"
        EmployeeSalary = 45000.0
        NumberOfEmployees = 1

        Console.WriteLine("Number of employees: " & NumberOfEmployees)
        Console.WriteLine("Employee name: " & EmployeName)
        Console.WriteLine("Employee phone number: " & EmployeePhoneNumber)
```

```
        Console.WriteLine("Employee salary: " & EmployeeSalary)

        Console.ReadLine() ' Pause to view output
    End Sub

End Module
```

Unfortunately, when you execute this program, the program does not display the employee's name. Instead, the program displays blank output for the name, as shown here:

```
Number of employees: 1
Employee name:
Employee phone number: 555-1212
Employee salary: 45000
```

If you examine the program statements closely, you will find that the Console.WriteLine statement that displays the employee's name misspells the variable name (it omits the ending *e* in Employee). Normally, Visual Studio requires that you declare each variable your program uses. By forcing the program to declare each variable, you eliminate such errors. In this case, the statement Option Explicit Off directs the compiler to let the program use a variable without first declaring the variable (an option you should not enable).

USE IT To force a program to declare variables, you place the following statement at the start of your program:

```
Option Explicit On
```

If you insert the statement at the start of the BadVariables.vb program, the Visual Basic .NET compiler will generate a syntax error message similar to those shown in Figure 1-5, that tells you

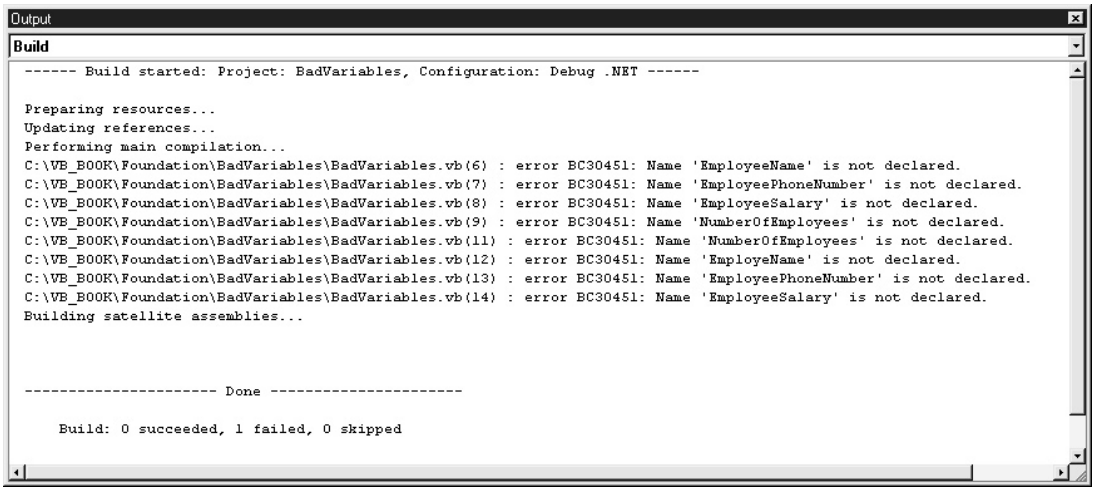


Figure 1-5 Syntax-error messages that correspond to undeclared variables

the variables are not declared. As you declare variables to remove the syntax errors, you will likely discover the misspelled variable name.

Beware of Variable Overflow and Precision

As you learned, a variable's type specifies the range of values a variable can store and a set of operations a program can perform on the variable. If you assign a value to a variable that exceeds the range of values the variable can store, an overflow error occurs. Assume that your program is using a variable of type Short to store a value in the range –32,678 to 32,767. Further, assume that the variable contains the value 32,767 and your program adds the value 1 to the variable as shown here:

```
Dim Value As Short = 32767
Value = Value + 1
```

When the program executes this statement, the value 32,768 will fall outside of the range of values the variable can store. When the overflow error occurs, your program will generate an exception and will end, displaying an error message similar to that shown in Figure 1-6. Chapter 9 discusses exceptions and how your programs should respond to such errors.

USE IT Just as a variable's type limits the range of values a variable can store, it also limits the accuracy (or precision) of the value a variable can represent. Variables of type Single are generally precise to 7 digits to the right of the decimal point, whereas variables of type Double are precise to 15 digits. The following program, ShowPrecision.vb, illustrates the accuracy of single- and double-precision variables:

```
Module Module1

    Sub Main()
        Dim A As Single = 0.123456789012345
        Dim B As Double = 0.123456789012345

        Console.WriteLine("Single: " & A)
        Console.WriteLine("Double: " & B)

        Console.ReadLine()
    End Sub

End Module
```

After you compile and execute this program, your screen will display the following output:

```
Single: 0.1234568
Double: 0.123456789012345
```

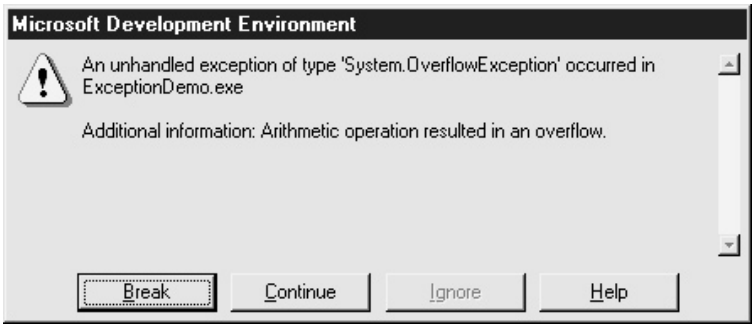


Figure 1-6 An exception (error) that occurs when the value a program assigns exceeds a type's acceptable range of values

As your programs manipulate floating-point numbers, a type's limited precision can lead to errors that are difficult to detect. The following program, PrecisionError.vb, uses a For loop to move through the values 0.01, 0.02, 0.03 up to 0.1. When the variable A contains the value 0.05, the program displays a message so stating:

```
Module Module1

    Sub Main()
        Dim A As Single

        For A = 0.01 To 0.1 Step 0.01
            If (A = 0.05) Then
                Console.WriteLine("Reached 0.05")
            End If
        Next

        Console.WriteLine("Done with loop")
        Console.ReadLine()
    End Sub

End Module
```

After you compile and execute the program, your screen will display the following output:

```
Done with loop
```

As you can see, the program does not display the message "Reached 0.05." That's because the computer's limited precision prevents the computer from exactly representing the value 0.05. For the loop to display the message, you must change the If statement to take the limited precision into account and test for the value being a few digits from 0.05, as shown here:

```
If (Math.Abs(A – 0.05) < 0.00001) Then
```

Performing Numeric Operations

To accomplish meaningful work, a program must perform operations on its variables. A program might multiply a variable that contains a user's total purchases by a sales tax amount, in order to determine the amount of tax the user must pay. Then the code may add the sales tax and shipping costs to the purchase amount in order to determine the total amount of money the user owes for his or her purchase of two items:

```
Purchases = 10.0 + 20.0
SalesTax = Purchases * 0.05
TotalDue = Purchases + SalesTax
```

To perform such operations, programs use arithmetic operations. Table 1-4 briefly describes the Visual Basic .NET arithmetic operators.

USE IT The following program, OperatorDemo.vb, illustrates the use of the various Visual Basic .NET arithmetic operators:

```
Module Module1

    Sub Main()
        Console.WriteLine("1 + 2 = " & 1 + 2)
        Console.WriteLine("3 - 4 = " & 3 - 4)
        Console.WriteLine("5 * 4 = " & 5 * 4)
        Console.WriteLine("25 / 4 = " & 25 / 4)
        Console.WriteLine("25 \ 4 = " & 25 \ 4)
        Console.WriteLine("25 Mod 4 = " & 25 Mod 4)
        Console.WriteLine("5 ^ 2 = " & 5 ^ 2)

        Console.ReadLine()
    End Sub

End Module
```

Operator	Purpose
+	Addition
-	Subtraction
*	Multiplication
/	Division (standard)
\	Division (integer)
Mod	Modulo (remainder)
^	Exponentiation

Table 1-4 The Visual Basic .NET Arithmetic Operators

After you compile and execute this program, your screen will display the following output:

```
1 + 2 = 3
3 - 4 = -1
5 * 4 = 20
25 / 4 = 6.25
25 \ 4 = 6
25 Mod 4 = 1
5 ^ 2 = 25
```

When your programs perform arithmetic operations, you must understand that Visual Basic .NET assigns a precedence to each operator, which controls the order in which the program will perform the operations. Assume that your program must calculate the result of the following expression:

```
Result = 5 + 3 * 2
```

If Visual Basic .NET were to perform the operations from left to right, it would calculate the result 16, which is not correct. Instead, Visual Basic .NET will first perform the multiplication operation, because the multiplication operator has a higher precedence than addition:

```
Result = 5 + 3 * 2
Result = 5 + 6
Result = 11
```

Table 1-5 illustrates the precedence of Visual Basic .NET arithmetic operators.

If based only on operator precedence, the order in which Visual Basic .NET performs arithmetic operations often will not match the order you need. Assume that your program needs to determine the sales tax (using 5% for this example) for two items priced at $10 and $20. Consider the following expression:

```
SalesTax = 10 + 20 * 0.05
```

Arithmetic Operator	Purpose
^	Exponentiation.
-	Negation.
*, /	Multiplication and division.
\	Integer division.
Mod	Remainder.
+, -	Addition and subtraction. Note that string concatenation using + has equal precedence to addition and subtraction, whereas concatenation using & has lower precedence.
And, Or, Not, Xor	Bitwise operators.

Table 1-5 Operator Precedence with Visual Basic .NET

Because the multiplication operator has a higher precedence than the addition operator, Visual Basic .NET will perform the multiplication first, which results in the following incorrect result:

```
SalesTax = 10 + 20 * 0.05
         = 10 + 10
         = 20
```

USE IT To control the order in which Visual Basic .NET performs operations, you must group operations in parentheses. When Visual Basic .NET evaluates an expression, Visual Basic .NET will perform operations that appear in parentheses first. In the previous sales tax example, you can use parentheses as follows to ensure that the program adds the first two items and then performs the multiplication on the result:

```
SalesTax = (10 + 20) * 0.05
```

In this case, the program would calculate the sales tax as follows:

```
SalesTax = (10 + 20) * 0.05
         = (30) * 0.05
         = 1.50
```

The following program, PrecedenceDemo.vb, illustrates how using parentheses to force the order of an expression's evaluation can change the result:

```
Module Module1

    Sub Main()
        Dim Expression1 As Double
        Dim Expression2 As Double
        Dim Expression3 As Double
        Dim Expression4 As Double

        Expression1 = 5 ^ 2 + 1 * 3 – 4
        Expression2 = 5 ^ (2 + 1) * 3 – 4
        Expression3 = 5 ^ (2 + 1) * (3 - 4)
        Expression4 = 5 ^ ((2 + 1) * (3 - 4))

        Console.WriteLine(Expression1)
        Console.WriteLine(Expression2)
        Console.WriteLine(Expression3)
        Console.WriteLine(Expression4)

        Console.ReadLine()
    End Sub

End Module
```

After you compile and execute this program, your screen will display the following output:

```
24
371
-125
0.008
```

Casting a Value of One Variable Type to Another

A variable's type specifies a range of values a variable can store and a set of operations a program can perform on a variable. In your programs, there will be times when you must assign a value of one type of variable to a variable of a different type. Programmers refer to such operations as "casting" the variable's type.

When you assign a value of a "smaller type," such as a value of type Short, to a larger type, such as a variable of type Integer, Visual Basic .NET can perform the assignment because the smaller variable's value can fit into the larger variable's storage capacity. Programmers refer to an assignment operation that casts the value of a variable of a smaller type to a variable of a larger type as an implicit cast.

In contrast, if you reverse the assignment and assign a value of a larger type to a smaller type, Visual Basic .NET must discard bits that represent the larger variable's value. If you assign an Integer value (which Visual Basic .NET represents using 32 bits) to a variable of type Short (which uses 16 bits), Visual Basic .NET will discard the value's upper 16 bits, which obviously can lead to erroneous results.

USE IT For such cases when your code must assign the value of a larger type to a variable of a smaller type and you do not care that Visual Basic .NET may discard part of the larger value, your code must perform an operation that programmers call an explicit cast. To perform an explicit cast, your program must use one of the built-in functions listed in Table 1-6.

Function	Purpose
ToBoolean	Converts a value to a Boolean (True or False).
ToByte	Converts a value to an 8-bit Byte in the range 0 to 255.
ToChar	Converts a value to a 2-byte Unicode character.
ToDateTime	Converts a value to a DateTime object.
ToDecimal	Converts a value to a 12-byte Decimal.
ToDouble	Converts a value to an 8-byte Double.
ToInt16	Converts a value to a 2-byte Short.
ToInt32	Converts a value to a 4-byte Integer.

Table 1-6 Type Conversion Routines Provided in the System.Convert Namespace

Function	Purpose
ToInt64	Converts a value to an 8-byte Integer.
ToSByte	Converts a value to an 8-bit signed value in the range –128 to 127.
ToSingle	Converts a value to a 4-byte Single.
ToString	Converts a value to its String representation.
ToUInt16	Converts a value to a 2-byte unsigned Short in the range 0 to 65,535.
ToUInt32	Converts a value to a 4-byte unsigned Integer in the range 0 to 4,294,967,295.
ToUInt64	Converts a value to an 8-byte unsigned long Integer in the range 0 to 18,446,744,073,709,551,615.

Table 1-6 Type Conversion Routines Provided in the System.Convert Namespace *(continued)*

The following program, CastDemo.vb, performs two simple cast operations. The first assigns a value of type Integer to a value of type Short. The second casts a value of type Double to a value of type Single:

```
Module Module1

    Sub Main()
        Dim BigInteger As Integer = 10000
        Dim LittleInteger As Short

        Dim BigFloat As Double = 0.123456789
        Dim LittleFloat As Single

        LittleInteger = BigInteger
        LittleFloat = BigFloat

        Console.WriteLine(LittleInteger)
        Console.WriteLine(LittleFloat)

        Console.ReadLine()
    End Sub

End Module
```

After you compile and execute this program, your screen will display the following output:

```
10000
0.1234568
```

In the case of the floating-point value 0.123456789, you can see that the assignment caused the operation to lose significant digits. The Integer to Short assign, in this case, was successful because

the Integer variable contained a value in the range a variable of type Short can store. If, for example, you change the program to use the value 40000, the assignment will cause the program to generate an overflow exception. So, depending on the value that the Integer variable contains, this program may or may not work.

To reduce potential errors that can occur when your programs assign a value from a larger type to a smaller type, your code can demand that the Visual Basic .NET compiler not allow such assignments by placing the following statement at the start of your programs:

```
Option Strict On
```

After you enable strict type-conversion processing, the following assignment statement would cause the Visual Basic .NET compiler to generate a syntax error:

```
LittleInteger = BigInteger
```

Making Decisions Using Conditional Operators

A program is simply a set of instructions the CPU executes to perform a specific task. The programs this chapter has presented thus far have begun their execution with the first statement and then have continued to execute statements one following another, up to and including the last statement.

As your programs perform more complex operations, your code will often perform one set of operations for a given condition and a second set for a different condition. For example, a program that implements a Web-based shopping cart would use one sales tax for customers in Texas and another for customers in New York.

Programmers refer to the process of a program making decisions about which statements to execute as conditional processing. To perform conditional processing, programs make extensive use of the If and If-Else statements.

To use an If statement, programs must specify a condition that Visual Basic .NET evaluates to either True or False. A condition might, for example, use an If statement to determine whether a user's name is "Smith" or a student's test score was greater than or equal to 90:

```
If (Username = "Smith") Then
   ' Statements to execute
End If

If (TestScore > 90) Then
   ' Statements to execute
End If
```

When Visual Basic .NET encounters the If statement, Visual Basic .NET will evaluate the corresponding condition. If the condition evaluates to True, your program will execute the statements that appear between the If and End If statements. If, instead, the condition evaluates to False, your program will not execute the statements, continuing its execution at the first statement following the End If.

Often, programs must perform one set of statements when a condition is true and a second when the condition is false. In such cases, the programs can use the If-Else statement. The following statement displays a message that tells a student that he or she passed an exam (the student passes if his or her score is greater than or equal to 70). If the student did not pass the test, the code displays a message telling the student that he or she failed:

```
If (TestScore >= 70) Then
  Console.WriteLine("You passed!")
Else
  Console.WriteLine("You failed")
End If
```

To improve a program's readability, programmers normally indent the statements that appear in constructs, such as the If and Else statements. Using indentation, a programmer, at a glance, can determine which statements relate. To simplify the indentation process, Visual Studio will automatically indent the statements you type in an If statement.

Many times, a program must evaluate a variety of conditions. For example, rather than simply telling the student whether he or she passed or failed the exam, a better program would display the student's grade based on the following. One way to display a message that corresponds to the student's grade is to use several If statements, as shown here:

```
If (TestScore >= 90) Then
  Console.WriteLine("You got an A")
End If

If (TestScore >= 80) And (TestScore < 90)
  Console.WriteLine("You got a B")
End If

If (TestScore >= 70) And (TestScore < 80)
  Console.WriteLine("You got a C")
End If

If (TestScore < 70) Then
  Console.WriteLine("You failed")
End If
```

The problem with the previous series of If statements is that regardless of the user's test score, the program must evaluate each If statement, which adds unnecessary processing. A better solution is to use a series of If-Else statements as shown here:

```
If TestScore >= 90 Then
  Console.WriteLine("Test grade: A")
ElseIf TestScore >= 80 Then
  Console.WriteLine("Test grade: B")
```

```
ElseIf TestScore >= 70 Then
  Console.WriteLine("Test grade: C")
Else
  Console.WriteLine("Test grade: F")
End If
```

USE IT The following program, IfDemo.vb, illustrates the use of several different If and If-Else statements:

```
Module Module1

    Sub Main()
        Dim TestScore As Integer = 80

        If TestScore >= 90 Then
            Console.WriteLine("Test grade: A")
        ElseIf TestScore >= 80 Then
            Console.WriteLine("Test grade: B")
        ElseIf TestScore >= 70 Then
            Console.WriteLine("Test grade: C")
        Else
            Console.WriteLine("Test grade: F")
        End If

        Dim Language As String = "English"

        If (Language = "English") Then
            Console.WriteLine("Hello, world!")
        ElseIf (Language = "Spanish") Then
            Console.WriteLine("Hola, mundo")
        End If

        If (Now.Hour < 12) Then
            Console.WriteLine("Good morning")
        ElseIf (Now.Hour < 18) Then
            Console.WriteLine("Good day")
        Else
            Console.WriteLine("Good evening")
        End If
```

```
        Console.ReadLine()
    End Sub

End Module
```

After you compile and execute this program, your screen will display output similar to the following:

```
Test grade: B
Hello, world!
Good morning
```

Take time to experiment with this program by changing the values the program assigns to the TestScore and Language variables.

Taking a Closer Look at the Visual Basic .NET Relational and Logical Operators

In a condition, such as the test for an If or While statement, programs use relational operators to compare one value to another. Using relational operators, an If statement can test if one value is greater than, equal to, or less than another value. The result of the condition's test is always a Boolean (True or False) result. Table 1-7 briefly describes the Visual Basic .NET relational operators.

Depending on the condition a program examines, there are many times when a program must test two or more relationships. An If statement might test if a user's age is greater than or equal to 21 and

Operator	Relational Comparison
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
=	Equal
<>	Not equal
Like	Tests whether a string matches a specified pattern

Table 1-7 The Visual Basic .NET Relational Operators

if the user lives in the United States. To test two more or conditions, programs use logical operators. Visual Basic .NET supports the And, Or, Xor, and Not relational operators.

USE IT The following program, ConditionDemo.vb, illustrates the use of the And, Or, and Not logical operators:

```
Module Module1

    Sub Main()
        Dim OwnsAPet As Boolean = False
        Dim OwnsADog As Boolean = True
        Dim OwnsACat As Boolean = True

        If (Not OwnsAPet) Then
            Console.WriteLine("You need a pet")
        End If

        If (OwnsADog Or OwnsACat) Then
            Console.WriteLine("Dogs and Cats are great")
        End If

        If (OwnsADog And OwnsACat) Then
            Console.WriteLine("Do the dog and cat get along?")
        End If

        Console.ReadLine()
    End Sub

End Module
```

After you compile and execute this program, your screen will display the following output:

```
You need a pet
Dogs and Cats are great
Do the dog and cat get along?
```

Take time to experiment with this program by changing the True and False values the program assigns to each variable and then run the program to see how the change affects each condition.

Handling Multiple Conditions Using Select

USE IT When programmers discuss conditional processing, they often focus their solutions on If and If-Else statements. Visual Basic .NET, however, provides the Select control structure (construct) that your programs can use to simplify the code you must write to handle complex conditions. At first glance, the Select statement looks quite similar to a series of If-Else statements. For example, the following Select statement displays a message based on the current day of the week:

```
Dim DayOfWeek As Integer
DayOfWeek = Now.DayOfWeek

Select Case DayOfWeek
  Case 0
    Console.WriteLine("Sunday")
  Case 1
    Console.WriteLine("Monday")
  Case 2
    Console.WriteLine("Tuesday")
  Case 3
    Console.WriteLine("Wednesday")
  Case 4
    Console.WriteLine("Thursday")
  Case 5
    Console.WriteLine("Friday")
  Case 6
    Console.WriteLine("Saturday")
End Select
```

Using a Select statement, you can also specify a condition and a series of possible matching results. The following Select statement determines and displays a user's grades:

```
Dim TestScore As Integer
TestScore = 84

Select Case TestScore
   Case Is >= 90
     Console.WriteLine("Grade: A")
   Case Is >= 80
```

```
      Console.WriteLine("Grade: B")
   Case Is >= 70
      Console.WriteLine("Grade: C")
   Case Else
      Console.WriteLine("Grade: F")
End Select
```

As you can see, when you specify a condition in a Select statement, you must use the Is keyword. Also note that the Select statement supports an Else case that it executes when none of the specified cases match the selected value. In addition to using a condition in a Select statement as just shown, you can also specify a range of matching values, as shown here:

```
Dim TestScore As Integer
TestScore = 84

Select Case TestScore
   Case 90 To 100
      Console.WriteLine("Grade: A")
   Case 80 To 89
      Console.WriteLine("Grade: B")
   Case 70 To 79
      Console.WriteLine("Grade: C")
   Case Else
      Console.WriteLine("Grade: F")
End Select
```

In the first example, you learned how to use a Select statement to match one value. Depending on the processing your program performs, there may be times when you want to perform the same processing for a range of values. The following Select statement displays messages based on the current day. For several days of the week, the code displays the same message:

```
Dim DayOfWeek As Integer
DayOfWeek = Now.DayOfWeek

Select Case DayOfWeek
  Case 0, 6
    Console.WriteLine("Enjoy the weekend")
  Case 1, 2, 3
    Console.WriteLine("Too many days until the weekend")
  Case 4, 5
    Console.WriteLine("Almost the weekend!")
End Select
```

Repeating a Series of Instructions

Just as there may be times when your programs must perform conditional processing in order to make decisions, there will also be times when your programs must perform one or more statements as long as a specific condition is true, or you may want the statements to execute a specific number of times. Programmers refer to such repetitive processing as iterative processing. Visual Basic .NET provides four iterative constructions your programs can use to repeat one or more statements: the For, While, For Each, and Do While loops.

The For loop exists to let your programs repeat one or more statements a specific number of times. The following statement uses a For loop to display the numbers 1 to 5 on the screen using Console.WriteLine:

```
Dim I As Integer

For I = 1 To 5
   Console.WriteLine(I)
Next
```

When this loop executes, your screen would display the following output:

```
1
2
3
4
5
```

The For loop consists of three parts. The first part of the statement initializes the loop's control variable. The second part compares the control variable to an ending condition. The third part is optional and specifies the amount by which the loop increments or decrements the control variable with each iteration. The following loop displays the numbers 0, 10, 20, … to 100 by incrementing the control variable by 10 with each iteration:

```
For I = 0 To 100 Step 10
   Console.WriteLine(I)
Next
```

When your programs use a For loop, Visual Basic .NET does not restrict you to using only counting numbers. You can also use variables of type Single and Double as a loop's control variable. The following statements use a For loop to display values 0.0 to 1.0 by incrementing the loop's control variable by 0.1 with each iteration:

```
Dim X As Double
For X = 0.0 To 1.0 Step 0.1
  Console.WriteLine(X)
Next
```

By using a negative step value, a For loop can move downward through a range of values. The following statements loop through the values 100 down to 0, decrementing the step value by 10 with each iteration:

```
For I = 100 To 0 Step -10
   Console.WriteLine(I)
Next
```

In contrast to the For loop, which repeats one or more statements a specific number of times, the While loop repeats statements as long as a specific condition is true. In your programs, you might use a While loop to read and display lines of a file. In this case, the statements in the loop would continue to read and display the file's content while (as long as) the file contains content the program has not yet read and displayed (meaning, you have not yet reached the end of the file). Or you might use a While loop to display and respond to a user menu's option selections until the user chooses the Quit option. The format of the While loop is as follows:

```
While (Condition)
  ' Statements to repeat
End While
```

Note that unlike previous versions of Visual Basic that used the Wend statement to mark the end of a While loop, Visual Basic .NET uses End While.

The For Each statement lets you repeat one or more statements for each element of an array. The following statements use a For Each statement to display the names of files that reside in the current directory:

```
Dim Files As String() = Directory.GetFiles(".")
Dim Filename As String

For Each Filename In Files
  Console.WriteLine(Filename)
Next
```

Finally, the Do loop is similar to the While loop in that it lets your programs repeat one or more statements while a specific condition is met. However, unlike the While loop, which places the test at the start of the loop, the Do loop places the test at the end of the loop. This means the statements a Do loop contains will always execute at least one time:

```
Do
  ' Statements to repeat
Loop While (Condition)
```

USE IT The following program, LoopDemos.vb, uses the For, While, and Do While loops to iterate through a range of values. Then the code uses the For Each loop to display the names of files in the current directory:

```
Imports System.IO

Module Module1

    Sub Main()
        Dim I As Integer

        For I = 0 To 10
            Console.Write(I & " ")
        Next

        Console.WriteLine()

        Dim X As Double = 0.0
        While (X < 100)
            Console.Write(X & " ")
            X = X + 25
        End While

        Console.WriteLine()

        Do
            Console.Write(X & " ")
            X = X – 10
        Loop While (X < 0)

        Console.WriteLine()

        Dim Files As String() = Directory.GetFiles(".")
        Dim Filename As String

        For Each Filename In Files
            Console.WriteLine(Filename)
        Next
        Console.ReadLine()
    End Sub
End Module
```

After you compile and execute this program, your screen will display the following output:

```
0 1 2 3 4 5 6 7 8 9 10
0 25 50 75
100
.\LoopDemo.exe
.\LoopDemo.pdb
```

Avoiding Infinite Loops

Iterative constructs such as the For and While statements exist to let your programs repeat a series of statements a specific number of times or as long as a specific condition is true. When your program uses iterative constructs, there may be times when a loop does not end (normally because of a programming error). Programmers refer to unending loops as infinite loops, because unless you can end the program by closing the program window, the loop will continue to execute forever. The goal of the following While loop, for example, is to display the values 0 through 99. However, if you examine the loop, you will find that it does not increment the variable I. As a result, after the loop starts, the loop will reach its ending condition (I equal to 100), so the code will repeat forever:

```
Dim I As Integer = 0

While I < 100
  Console.WriteLine(I)
End While
```

USE IT To reduce the possibility of an infinite loop in your programs, you should examine each loop to ensure that the loop correctly performs the following four steps:

1. Initializes a control variable
2. Tests the control variable's value
3. Executes the loop's statements
4. Modifies the control variable

You can remember these four steps using the ITEM (Initialize, Test, Execute, Modify) acronym. Consider the previous While loop. The code initializes the variable I when it declares the variable. Then the first statement of the While loop tests the control variable. In the While loop, the code executes the Console.WriteLine statement. However, the code does not modify the control variable in the loop, which leads to the infinite loop.

Executing a Loop Prematurely

In a Visual Basic .NET program, loops let your code repeat a set of instructions a specific number of times or while a specific condition is met. Ideally, a loop should have one condition that determines if the code will perform (and later repeat) the loop's statements. In a For loop, the loop's processing ends when the loop's control variable's value is greater than the loop's ending value.

USE IT That said, there may be times when your code must terminate a For loop's (or a For Each loop's) processing prematurely. In such cases, your code can use the Exit For statement, which directs Visual Basic .NET to end the loop's processing and to continue the program's execution at the first statement that follows the For statement (the first statement that follows Next). In a similar way, to exit a While loop prematurely, your code can execute the Exit While statement.

NOTE

Later in this chapter, you will examine subroutines that let you group a set of related statements that perform a specific task. Normally, a subroutine will execute its statements in succession, from the first statement to the last. However, there may be times when you must end a subroutine's processing prematurely. In such cases, your code can issue the Exit Sub statement. However, as is the case of the For and While loops, to improve the readability of your code, you should avoid using Exit statements whenever possible.

Visual Basic .NET Supports Lazy Evaluation to Improve Performance

When your programs perform conditional and iterative processing, you can improve your program's performance by changing the way that Visual Basic .NET handles conditions that use the logical And and Or operators. In statements such as the If, Select, and While statements, your programs can use the logical And operator to specify two conditions that must evaluate to True before the program will perform the corresponding statements. The following statement uses the logical And operator to test if the employee is a programmer and if the programmer knows how to program using Visual Basic .NET:

```
If (UserIsProgrammer) And (UserKnowsVB) Then
  ' Statements
End If
```

Normally, when your code uses logical operators to evaluate a condition, Visual Basic will examine each part of the condition and then determine whether the condition is true. However, in the case of the And operator, if the first half of a condition evaluates to False, you know the entire condition will be False. In a similar way, in the case of the Or operator, if the first part of a condition evaluates to True, you know the entire condition will be True. To improve performance, you can take advantage of the AndAlso and OrElse operators. The AndAlso operator directs Visual Basic .NET to only evaluate the second part of a condition if the first part evaluates as a True. The OrElse operator directs Visual Basic .NET to evaluate the second part of a condition that uses OrElse should the first part of the condition evaluate to False.

USE IT The following program, LazyEvaluation.vb, illustrates the use of AndAlso and OrElse operators:

```
Module Module1
    Sub Main()
        Dim OwnsDog As Boolean = True
        Dim OwnsCat As Boolean = False

        If OwnsDog AndAlso OwnsCat Then
            Console.WriteLine("Owns both a dog and a cat")
        End If
```

```
        If OwnsDog OrElse OwnsCat Then
            Console.WriteLine("Owns a dog or cat--maybe both")
        End If

        Console.ReadLine()
    End Sub
End Module
```

You may be wondering why Visual Basic .NET does not simply use lazy evaluation all the time. The reason is that there may be times when not executing the second half of a condition can introduce errors. Consider the following If statement that calls a function in each of the conditions:

```
If (IsAfterBusinessHours() And BuildingIsSecure()) Then
```

Depending on the processing the BuildingIsSecure function performs, you may not want the program to skip the function's execution simply because the IsAfterBusinessHours function returns a False result (which would end the statement's execution if you use lazy evaluation).

Wrapping Long Statements

As you examine the programs this book's Tips present, there will be many times when the code wraps a long statement onto two or more lines because of the limitations of the printed page. As you program, there may be times when you will want to wrap a long statement onto the next line so you do not have to continually scroll horizontally to see the program statements.

USE IT To wrap a statement to the next line, you must place a space followed by an underscore (_) character at the end of the line, as shown here:

```
SomeVeryLargeVariableName = SomeLongSubroutineName(ParameterOne, _
  ParameterTwo, ParameterThree, ParameterFour)
```

Often, programmers will wrap long character strings over two or more lines. To wrap a character string, you must break the string into multiple strings, so that one string ends at the point you want to wrap the line and a new string begins at the location you want on the following line. Between each of the strings, you place the concatenation operator (&) as shown here:

```
Console.Writeline("The title of the book is: Visual Basic .Net" & _
   " Programming Tips and Techniques")
```

When you wrap a string in this way, you must remember to place a space character at either the end of one string or at the start of the new string if the strings were originally separated by a space.

Taking Advantage of the Visual Basic Assignment Operators

In programs, it is common to perform an arithmetic operation that uses a variable's current value and then assigns the result of the operation back to the variable. The following statement adds the value 1 to the variable Counter:

```
Counter = Counter + 1
```

USE IT To simplify operations that use a variable's value in an expression and then assign the result back to the variable, Visual Basic .NET provides the set of assignment operators listed in Table 1-8.

The following statement uses the addition assignment operator to increment the value of the Counter variable by 1:

```
Counter += 1
```

The following program, AssignmentDemo.vb, illustrates the use of the Visual Basic .NET assignment operators:

```
Module Module1

    Sub Main()
        Dim A As Integer

        A = 0
        A += 10
        Console.WriteLine("A += 10 yields " & A)

        A -= 5
        Console.WriteLine("A -=5 yields " & A)

        A *= 3
        Console.WriteLine("A *= 3 yields " & A)

        A /= 5
        Console.WriteLine("A /= 5 yields " & A)

        A ^= 2
        Console.WriteLine("A ^= 2 yields " & A)

        Console.ReadLine()
    End Sub

End Module
```

Operator	Purpose
+=	Adds the specified expression to a variable's current value.
-=	Subtracts the specified expression from a variable's current value.
*=	Multiplies a variable's current value by the specified expression and assigns the result back to the variable.
/= and \=	Divides a variable's value contents by the specified expression and assigns the result back to the variable.
^=	Raises a variable's current value to the power of the specified expression and assigns the result back to the variable.
&=	Concatenates the String to a variable's value and assigns the result back to the variable.

Table 1-8 The Visual Basic .NET Assignment Operators

After you compile and execute this program, your screen will display the following output:

```
A += 10 yields 10
A -=5 yields 5
A *= 3 yields 15
A /= 5 yields 3
A ^= 2 yields 9
```

Commenting Your Program Code

As you program, you should place comments throughout your code that explain the processing your program performs or a specific variable's use. Later, you or another programmer who is reviewing the code can read the comments to quickly understand the processing. To place a comment in a Visual Basic .NET program, you place a single quote on a line followed by the comment text. The Visual Basic .NET compiler will ignore any text that appears to the right of the single quote, treating the text as a comment.

USE IT In a program, you can place comments on their own lines or to the right of a statement:

```
' The following subroutine displays the current date and time
Sub ShowDateTime()
  Console.WriteLine(Now())  'Now returns the current date and time
End Sub
```

As you test your programs, there may be times when you will want to disable one or more statements. Rather than removing the statements from your code, you can simply place the single-quote character in front of the statement. When Visual Basic .NET compiles your program, it will ignore the statement,

which it will treat as a comment. If you later want the program to execute the statement, you can simply remove the single quote:

```
Console.WriteLine("This line will appear")
' Console.WriteLine("This line will not")
```

Reading Keyboard Input Using Console.Read and Console.ReadLine

When you create a console application, your programs can read keyboard input from the user using the Console.Read and Console.ReadLine methods. The difference between the methods is that Console.ReadLine returns all the characters up to the ENTER key, whereas Console.Read reads characters up to the first whitespace character, such as a space or tab. To assign the value a user types to a variable, you use the assignment operator as follows:

```
VariableName = Console.ReadLine()
```

Many of the Tips this book presents place the statement Console.ReadLine() at the end of the program code. When you run a console application in Visual Studio, the console window will immediately close after the program completes its processing. By placing the Console.ReadLine() statement at the end of the code, the program will pause, waiting for the user to press the ENTER key, before the program ends and the window closes.

USE IT The following program, KeyboardInputDemo.vb, illustrates the use of the Console.Read and Console.ReadLine methods:

```
Module Module1

    Sub Main()
        Dim Age As Integer
        Dim FirstName As String
        Dim Salary As Double

        Console.Write("Age: ")
        Age = Console.ReadLine()

        Console.Write("Name: ")
        FirstName = Console.ReadLine()

        Console.Write("Salary: ")
        Salary = Console.ReadLine()
        Console.WriteLine(Age & " " & FirstName & " " & Salary)
```

```
        Console.Write("Enter Age, Name Salary: ")
        Age = Console.Read()
        FirstName = Console.Read()
        Salary = Console.ReadLine()
        Console.WriteLine(Age & " " & FirstName & " " & Salary)
    End Sub

End Module
```

Take time to experiment with the program. You will find that if you type a value that does not correspond to the data type the program expects (such as if the program prompts for an age and you type your name instead), the program will generate an exception and will end. Chapter 9 examines exception processing in detail. To avoid such errors, many programs will read all keyboard input into character strings and then convert the string to an Integer or Double value as required.

Displaying a Message in a Message Box

Normally, console-based applications will display output to the console window using the Console.Write and Console.WriteLine methods. Likewise, a Windows-based program will display output using one or more controls that appear on a form. Both console- and Windows-based applications, however, can use a message box, similar to that shown here, to display a message to the user and to get a user's button response (such as OK or Cancel).

For years, Visual Basic programmers have used the MsgBox function to display a message box to the user:

```
MsgBox("Hello, User")
```

Although Visual Basic .NET supports the MsgBox function, most newer Windows-based programs will use the MessageBox class Show method to display a message box (as it turns out, behind the scenes, the MsgBox function itself calls MessageBox.Show):

```
MessageBox.Show("Hello, User")
```

To determine which message-box button a user selects, you assign the result of the MessageBox.Show (or MsgBox) method to a variable, as shown here:

```
VariableName = MessageBox.Show("Message", "Title", _
   MessageBoxButtons.OKCancel)
```

USE IT The following program, MsgBoxDemo.vb, illustrates the use of the MsgBox function in a console application:

```
Module Module1
  Sub Main()
     MsgBox("Message")
     MsgBox("Message", MsgBoxStyle.AbortRetryIgnore, "Title")
  End Sub
End Module
```

The following program, MessageBoxDemo.vb, illustrates the use of the various message box types:

```
Public Class Form1
    Inherits System.Windows.Forms.Form

#Region " Windows Form Designer generated code "
   ' Code not shown
#End Region

  Private Sub Form1_Load(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles MyBase.Load
      MessageBox.Show("Message")
      MessageBox.Show("Message", "Title")
      MessageBox.Show("Message", "Title", MessageBoxButtons.OKCancel)
      MessageBox.Show("Message", "Title", _
        MessageBoxButtons.AbortRetryIgnore, MessageBoxIcon.Warning)
      Me.Close()
  End Sub
End Class
```

Prompting the User for Input Using an Input Box

Normally, console-based applications will get keyboard input from the user using the Console.Readline method. Likewise, Windows-based applications usually get input using one or more form-based controls. That said, both application types can use the InputBox function to prompt the user for input, as shown in Figure 1-7. Using the InputBox function, your code can assign the value the user enters to a variable, as shown here:

```
VariableName = InputBox("Enter name")
```

Figure 1-7 Using an input box to prompt the user for input

USE IT Using the InputBox function, you can pass parameters that specify the prompt the box displays, the box title, as well as the x and y offsets (from the upper-left corner of the screen) at which the box appears. The following program, InputBoxDemo.vb, illustrates the use of the InputBox function:

```
Module Module1

    Sub Main()
        Dim Name As String
        Dim Age As Integer
        Dim Salary As Double

        Name = InputBox("Enter name")
        Age = InputBox("Enter age", 21)
        Salary = InputBox("Enter salary")

        Console.WriteLine(Name)
        Console.WriteLine(Age)
        Console.WriteLine(Salary)
        Console.ReadLine()
    End Sub

End Module
```

Again, when you use the InputBox function, the user must enter the value in the format that matches the type to which you are assigning a result. If, for example, you assign the result to an Integer value and the user enters a nonnumeric value, the program will generate an exception. Chapter 9 examines exceptions in detail. To avoid such errors, programs normally assign the InputBox result to a String variable and then convert that value to the format that matches the target variable's type.

Breaking a Programming Task into Manageable Pieces

As programs become larger and more complex, programmers often break the program into smaller, more manageable pieces of code, each of which performs a specific task. To organize the program statements by task, programmers use functions and subroutines.

The difference between a function and a subroutine is that after a function performs its processing, the function returns a value (a result) and a subroutine does not. For example, the MessageBox.Show function displays a dialog box and then returns a value that corresponds to the user's button selection. To use the function's return value, your code normally assigns the result to a variable, as shown here:

```
Variable = SomeFunctionName(OptionalParameters)
```

To create a subroutine, your code uses the keyword Sub, followed by a unique name (that should correspond to the processing the subroutine performs), followed by parentheses that contain optional variables that will store values the program passes to the subroutine (which programmers call parameters). If the subroutine does not use parameters, you will simply place empty parentheses after the subroutine name. Next, you place the statements that correspond to the processing the subroutine performs followed by the End Sub statement, as shown here:

```
Sub SubroutineName()
  ' Statements here
End Sub
```

The following statements create a subroutine named GreetUser that displays messages to the user using the Console.WriteLine method:

```
Sub GreetUser()
  Console.WriteLine("Hello, user")
  Console.WriteLine("The current date and time is: " & Now())
  Console.WriteLine("Have a nice day.")
End Sub
```

To use a subroutine in your program, you place the subroutine name, followed by the parentheses in your program code, as shown here:

```
GreetUser()
```

Programmers refer to the process of using a subroutine as "calling the subroutine." When your program encounters a subroutine call, your program will jump to the statements the subroutine contains. After the program completes the subroutine's processing, your program will resume its processing with the statement in your code that follows the subroutine call.

NOTE

Visual Basic .NET no longer supports the GoSub statement which programmers used for many years to call a subroutine.

As your programs become larger, it is likely that they will contain many subroutines. A word processing program, for example, might use one subroutine to spell-check a document, another to save the document's contents to a file on disk, and yet another to print the document. In a console-based application, the program's execution will always begin with the first statement that appears in the subroutine called Main. (You can think of the Main subroutine as containing your main or primary program statements). From within the Main subroutine, your code can call other subroutines.

The following program, SubDemo.vb, creates and calls several subroutines:

```
Module Module1

    Sub ShowBookInformation()
        Console.WriteLine("Title: Visual Basic .Net " & _
           "Programming Tips & Techniques")
        Console.WriteLine("Author: Jamsa")
        Console.WriteLine("Publisher: McGraw-Hill/Osborne")
        Console.WriteLine("Price: 49.99")
    End Sub

    Sub GreetInEnglish()
        Console.WriteLine("Hello, world")
    End Sub

    Sub GreetInSpanish()
        Console.WriteLine("Hola, mundo")
    End Sub

    Sub ShowTime()
        Console.WriteLine("Current time is: " & Now)
    End Sub

    Sub Main()
        ShowTime()
        GreetInEnglish()
        GreetInSpanish()
        ShowBookInformation()
        Console.ReadLine()
    End Sub

End Module
```

As you can see, the program defines four subroutines, placing the statements for each between the Sub and End Sub statements. When the program runs, Visual Basic .NET will begin the program's execution in the subroutine named Main—a console application will always begin its execution in Main. When the program encounters the ShowTime subroutine call, the program will jump to the statements the ShowTime subroutine contains, which in this case is the Console.WriteLine statement that displays the current date and time. Then, after the subroutine completes its processing, the program will resume its execution back in Main at the first statement that follows the ShowTime subroutine call—the call to the GreetInEnglish subroutine. Again, the program will branch its execution to the statements the subroutine contains. The program will continue this process of calling a subroutine and then returning to Main until the program completes its statements.

After you compile and execute this program, your screen will display the following output:

```
Current time is: 4/9/2002 10:46:47 AM
Hello, world
Hola, mundo
Title: Visual Basic .Net Programming Tips & Techniques
Author: Jamsa
Publisher: McGraw-Hill/Osborne
Price: 49.99
```

As briefly discussed, a function differs from a subroutine in that, after the function completes its processing, the function will return a value that your programs can assign to a variable or use in an expression. In Chapter 5, you will examine the arithmetic functions provided by the Math class. The following statements illustrate the use of several of the Math class functions:

```
Dim SomeValue, SomeAngle As Double

SomeValue = Math.Sqrt(100)
SomeAngle = Math.Acos(2.225)
```

When the program encounters the Math.Sqrt function call, the program will branch its execution to the corresponding program statements. After the function completes its processing, the program will assign the value the function returns to the SomeValue variable. Then the program will continue its execution with the next statement, which in this case, calls the Math class Acos function.

To create a function, you use the Function keyword followed by a unique function name and parentheses that optionally declare variables to hold values the program passes to the function. You must also then specify the Returns keyword followed by the type (such as Integer or String) of the value the function returns. You place the function statements between the function header (which programmers also refer to as the function signature) and the End Function statement, as shown here:

```
Function UniqueFunctionName(OptionalParameters) As FunctionReturnType
   ' Function statements go here
End Function
```

The following statements create a function named GetBookPrice, which returns a value of type Double:

```
Function GetBookPrice() As Double
  ' Function statements go here
End Function
```

To return a value, a function can use the Return statement, or the function can assign the value to its own name. In a function named GetBookPrice, the following statements both return the value 49.99:

```
Return 49.99

GetBookPrice = 49.99
```

To call a function in your program code, you must specify the function name followed by the parentheses and optional parameters. Normally, your code will assign the function's result to a variable as shown here:

```
Dim Price As Double

Price = GetBookPrice()
```

However, you can use the value a function returns in any expression, such as A = B + SomeFunction(), or in a call to Console.WriteLine, as shown here:

```
Console.WriteLine("The price is: " & GetBookPrice())
```

The following program, FunctionDemo.vb, creates two functions, one that returns value of type Double and one that returns a String. The program first calls each function, assigning the value the function returns to a variable. Then the program calls each function from within a Console.WriteLine statement, displaying the value the function returns. Finally, the code uses the GetBookPrice function in an If statement:

```
Module Module1
    Function GetBookPrice() As Double
        GetBookPrice = 49.99
    End Function

    Function GetBookTitle() As String
        GetBookTitle = "Visual Basic .Net Programming Tips & Techniques"
    End Function

    Sub Main()
        Dim Price As Double
        Dim Title As String
```

```
        Price = GetBookPrice()
        Title = GetBookTitle()

        Console.WriteLine(Price)
        Console.WriteLine(Title)

        Console.WriteLine(GetBookPrice())
        Console.WriteLine(GetBookTitle())

        If (GetBookPrice() = 49.99) Then
            Console.WriteLine("The book is 49.99")
        End If

        Console.ReadLine()
    End Sub

End Module
```

After you compile and execute this program, your screen will display the following output:

```
49.99
Visual Basic .Net Programming Tips & Techniques
49.99
Visual Basic .Net Programming Tips & Techniques
The book is 49.99
```

NOTE

As you read Visual Basic .NET books, you will often find that books use the terms function, subroutine, and method interchangeably. Keep in mind that a function differs from a subroutine in that the function returns a result. The term method is a general term that describes functions and subroutines that appear in a class. Chapter 3 discusses class variables in detail.

Passing Parameters to a Function or Subroutine

In a program, functions and subroutines exist to organize statements that perform a specific task. Often, to perform its processing, a function or subroutine will require that program pass it one or more values. The values a program passes to a function or subroutine are called parameters. Earlier in this chapter, for example, you passed values to the Console.WriteLine method for display:

```
Console.WriteLine("Values are {0}, {1}, {3}", 100, 200, 300)
Console.Writeline("The price is: " & GetBookPrice())
```

Similarly, your programs have passed values to the MessageBox.Show method:

```
MessageBox.Show("Hello, World")
```

To pass values to a subroutine or function, you must place the values in the parentheses that follow the routine's name, separating the values with a comma. Before a function or subroutine can use the values, the routine must declare variables in which Visual Basic .NET will place the values when the program calls the routine. You declare the variables between the parentheses that follow the function or subroutine name when you create the routine. The following statements create a subroutine called Greeting that lets programs pass a message (a parameter) to the routine that the subroutine will display:

```
Sub Greeting(ByVal Message As String)
  Console.WriteLine("Hello, user")
  Console.WriteLine("Today's message is: " & Message)
End Sub
```

To support parameters, you must declare a variable in the subroutine or function header that will hold the parameter's value as the function executes. In this case, the subroutine declares a Variable named Message of type String. For now, you can ignore the ByVal keyword, which you will examine in the following Tip. In the program code, you would call the Greeting subroutine as follows:

```
Greeting("Have a great day")
```

When a function or subroutine supports parameters, your program must pass the correct number and type of parameters to the routine. If a function expects a parameter of type Double, and your program passes a String value to the function, an error will occur. Likewise, if a routine expects three parameter values, your program must pass three parameters to the routine in the order the parameter expects the values.

The following program, ThreeSubs.vb, declares different subroutines, each of which uses a different number and type of parameter:

```
Module Module1
    Sub OneValue(ByVal Name As String)
        Console.WriteLine("Hello, " & Name)
    End Sub

    Sub TwoValues(ByVal Age As Integer, ByVal Name As String)
        Console.WriteLine("Age: " & Age)
        Console.WriteLine("Name: " & Name)
    End Sub

    Sub ThreeValues(ByVal Name As String, ByVal Age As Integer, _
          ByVal Salary As Double)
        Console.WriteLine("Name: " & Name)
        Console.WriteLine("Age: " & Age)
```

```
        Console.WriteLine("Salary: " & Salary)
    End Sub

    Sub Main()
        OneValue("Mr. Gates")
        Console.WriteLine()
        TwoValues(50, "Mr. Gates")
        Console.WriteLine()
        ThreeValues("Mr. Gates", 50, 250000.0)
        Console.ReadLine()
    End Sub

End Module
```

After you compile and execute this program, your screen will display the following output:

```
Hello, Mr. Gates

Age: 50
Name: Mr. Gates

Name: Mr. Gates
Age: 50
Salary: 250000
```

Declaring Local Variables in a Function or Subroutine

Depending on the processing a function or subroutine performs, there will be many times when the routine will require one or more variables to store values as the routine's statements execute. In a function or subroutine, you can declare variables, which programmers refer to as local variables, following the subroutine or function heading, as shown here:

```
Sub SomeFunction()
  Dim I As Integer
  Dim Sum As Double

  ' Statements go here
End Sub
```

Programmers refer to a subroutine or function's variables as "local" because the fact that the variables exist and the values the variables contain are known only to the function or subroutine. The code outside of the function or subroutine does not know about the routine's local variables, nor can the code use them. Because the variables are local, the names you use for the variables will not conflict with variables you have defined in other functions or subroutines.

The following program, LocalVarDemo.vb, defines local variables in a subroutine and function. Although the variables use the same names in each routine, the local variables are distinct and the values your code assigns to the variables in one routine do not affect the values of the variables in another:

```
Module Module1

    Sub YahooInfo()
        Dim Name As String = "Yahoo"
        Dim Price As Double = 17.45
        Dim I As Integer = 1001

        Console.WriteLine("In YahooInfo")
        Console.WriteLine("Name: " & Name)
        Console.WriteLine("Price: " & Price)
        Console.WriteLine("I: " & I)
    End Sub

    Sub BookInfo()
        Dim Name As String = "C# Programming Tips & Techniques"
        Dim Price As Double = 49.99
        Dim I As Integer = 0

        Console.WriteLine("In BookInfo")
        Console.WriteLine("Name: " & Name)
        Console.WriteLine("Price: " & Price)
        Console.WriteLine("I: " & I)
    End Sub

    Sub Main()
        YahooInfo()
        Console.WriteLine()
        BookInfo()
        Console.ReadLine()
    End Sub

End Module
```

After you compile and execute this program, your screen will display the following output:

```
In YahooInfo
Name: Yahoo
Price: 17.45
I: 1001

In BookInfo
```

```
Name: C# Programming Tips & Techniques
Price: 49.99
I: 0
```

Changing a Parameter's Value in a Subroutine

In the Tip titled "Passing Parameters to a Function or Subroutine," you learned how to pass a parameter to a function or subroutine. In each of the examples the Tip presented, the routines used, but did not change, the values of the parameters the routines received. Depending on the processing a subroutine or function performs, there may be times when you will want the subroutine to change a parameter's value. If you examine the previous functions and subroutines, you will find that each routine preceded its parameter declarations with the ByVal keyword:

```
Sub SomeSubroutineName(ByVal A As Integer)
```

The ByVal keyword specifies that Visual Basic .NET will pass the parameter to the function by value which means, when your program calls the routine, Visual Basic .NET will make a copy of the value the parameter contains. Then Visual Basic .NET will pass the copy of the value to the routine as opposed to the original variable. When you pass a parameter by value in this way, a subroutine or function cannot make a change to the parameter value that remains in effect after the routine ends. Consider the following program, ByValueDemo.vb, that passes the value of the Number parameter to a subroutine. In the subroutine, the code changes and displays the parameters value. After the subroutine ends and the program's execution resumes in the Main subroutine, the value of the variable number is unchanged from its original value of 100. That's because changes to a parameter that a program passes to a subroutine or function by value only remain in effect for the duration of the routine's processing:

```
Module Module1

    Sub NoChangeToParameter(ByVal A As Integer)
        A = 1001
        Console.WriteLine("Value of A in subroutine " & A)
    End Sub

    Sub Main()
        Dim Number As Integer = 100

        Console.WriteLine("Number before function call: " & Number)
        NoChangeToParameter(Number)
        Console.WriteLine("Number before function call: " & Number)
        Console.ReadLine()
    End Sub

End Module
```

After you compile and execute this program, your screen will display the following output:

```
Number before function call: 100
Value of A in subroutine 1001
Number before function call: 100
```

USE IT In order for a subroutine or function to change the value of a variable you pass to the routine as a parameter, the routine must know the variable's memory location (so the routine can use the memory location to replace the value the variable stores). To provide the routine with the address of the parameter, you must pass the parameter to the routine by reference. To do so, you precede the parameter variable name with the ByRef keyword. The following program, ByRefDemo.vb, passes a parameter by reference to the subroutine named ParameterChange. In the subroutine, the code changes and then displays the parameter's value. However, because the program passes the value by reference, the change to the value remains in effect after the subroutine ends:

```
Module Module1

    Sub ParameterChange(ByRef A As Integer)
        A = 1001
        Console.WriteLine("Value of A in subroutine " & A)
    End Sub

    Sub Main()
        Dim Number As Integer = 100

        Console.WriteLine("Number before function call: " & Number)
        ParameterChange(Number)
        Console.WriteLine("Number before function call: " & Number)
        Console.ReadLine()
    End Sub

End Module
```

After you compile and execute this program, your screen will display the following output:

```
Number before function call: 100
Value of A in subroutine 1001
Number before function call: 1001
```

Using Scope to Understand the Locations in a Program Where a Variable Has Meaning

In a Visual Basic .NET program, you can declare variables in functions and subroutines, as parameters to a function or subroutine, or in a construct such as an If or While statement. Depending on where

you declare a variable, the location in your program where the variable has meaning (in other words, the statements in your program where you can use the variable) will differ. Programmers refer to the program areas where a variable is known to the program as the variable's scope.

Assume that you have two subroutines that each use a variable named Counter, as shown here:

```
Sub BigLoop()
  Dim Counter As Integer

  For Counter = 1000 To 10000
    Console.WriteLine(Counter)
  Next
End Sub

Sub LittleLoop()
  Dim Counter As Integer

  For Counter = 0 To 5
    Console.WriteLine(Counter)
  Next
End Sub
```

When you declare a local variable in a subroutine (or function), the variable's scope (the locations where that variable is known) is limited to the subroutine. Outside of each of the subroutines, the program does not know that the subroutine uses a variable named Counter. In this case, the two variables named Counter each have different scope. If one subroutine changes the value of its Counter variable, the change has no effect on the second subroutine's variable. Because of each variable's limited scope, using the same variable name in different subroutines does not create a conflict.

When you create a console-based application, you can declare a variable outside of your subroutines and functions. The following statements declare a variable named Counter outside of the two subroutines:

```
Dim Counter As Integer

Sub BigLoop()
  For Counter = 1000 To 10000
    Console.WriteLine(Counter)
  Next
End Sub

Sub LittleLoop()
  For Counter = 0 To 5
    Console.WriteLine(Counter)
  Next
End Sub
```

When you declare a variable outside of the routines in this way, the variable has global scope—that is, the variable is known throughout your program code. Any function or subroutine in your program can change the value of a global variable (which can lead to errors that are very difficult to detect when you are not expecting a subroutine to change the variable's value). Because global variables can lead to such errors, you should not use (or you should severely limit the use of) global variables.

When a local variable has the same name as a global variable, the code will always use the local variable (ignoring the global variable). In this case, any changes the routines make to their local variables named Counter will not affect the global variable named Counter, and vice versa. However, because such conflicts can confuse a programmer who is reading your code, you should avoid the use of global variables. If a subroutine or function must change a variable's value, your code should pass the variable to the routine by reference (using the ByRef keyword in the routine's parameter declaration). That way, another programmer who reads your code can better track the fact that the routine changes the parameter's value.

USE IT The following program, ScopeDemo.vb, illustrates how scope affects variables in a program. The program uses a global variable named Counter, a local variable in a subroutine named Counter, and a variable named Counter whose scope corresponds to an If statement:

```
Module Module1

    Dim Counter As Integer

    Sub BigLoop()
        For Counter = 1000 To 1005      ' Use global Counter
            Console.Write(Counter & " ")
        Next
        Console.WriteLine()
    End Sub

    Sub LittleLoop()
        Dim Counter As Integer

        For Counter = 0 To 5      ' Use local Counter
            Console.Write(Counter & " ")
        Next
        Console.WriteLine()
    End Sub

    Sub Main()
        Counter = 100

        Console.WriteLine("Starting Counter: " & Counter)
        BigLoop()
```

```
        Console.WriteLine("Counter after BigLoop: " & Counter)
        LittleLoop()
        Console.WriteLine("Counter after LittleLoop: " & Counter)

        If (Counter > 1000) Then
            Dim Counter As Integer = 0

            Console.WriteLine("Counter in If statement: " & Counter)
        End If

        Console.WriteLine("Ending Counter: " & Counter)

        Console.ReadLine()

    End Sub

End Module
```

After you compile and execute this program, your screen will display the following output:

```
Starting Counter: 100
1000 1001 1002 1003 1004 1005
Counter after BigLoop: 1006
0 1 2 3 4 5
Counter after LittleLoop: 1006
Counter in If statement: 0
Ending Counter: 1006
```

Storing Multiple Values of the Same Type in a Single Variable

In a program, variables let the program store and later retrieve values as the program executes. Normally, a variable stores only one value at a time. Many programs, however, must work with many related values of the same type. For example, a program many need to calculate the average of 50 test scores, or changes in the prices of 100 stocks, or the amount of disk space consumed by files in the current directory. To store multiple values of the same type (such as 50 Integer values) in one variable, your programs can use an array data structure.

To create an array, you declare a variable of a specific type and then specify the number of elements the variable will store. The following statement declares an array named TestScores that can store 50 Integer values (the first array entry is at offset 0 in the array and the 50th is at offset 49):

```
Dim TestScores(49) As Integer
```

To access values in an array, you use an index value to specify the array element (the specific value's location) you desire. The first value in an array resides at location 0. The following statement assigns the test score 91 to the first element in the array:

```
TestScores(0) = 91
```

The following statements assign values to the first five elements of the TestScores array:

```
TestScores(0) = 91
TestScores(1) = 44
TestScores(2) = 66
TestScores(3) = 95
TestScores(4) = 77
```

Normally, to access the values in an array, a program uses a For loop that increments a variable the code uses to specify array locations. The following For loop would display the values previously assigned to the first five array elements:

```
For I = 0 To 4
  Console.WriteLine(TestScores(I))
Next
```

As discussed, the first element in an array resides at location 0. Likewise, the last element resides at the array size. In the case of the 50-element TestScores array, the last element would reside at TestScores(49). If your program tries to assign a value to an element outside of the array bounds (such as TestScores(100) = 45), the program will generate an exception. Chapter 9 examines exceptions and exception handling in detail.

To assign values to an array, a program might read the values from a file, prompt the user for the values, or calculate the values based on program-specific processing. In addition, a program can initialize an array by placing the values in left and right braces, as shown here (when you initialize an array in this way, you do not specify the array size in the parentheses that follow the array name):

```
Dim Values() As Integer = {100, 200, 300, 400, 500}
Dim Prices() As Double = {25.5, 4.95, 33.4}
```

As your programs execute, there may be times when you find that an array does not provide enough storage locations to hold the values you require. In such cases, your program can use the ReDim statement to increase or decrease the array's size.

The following program, ArrayDemo.vb, creates an array that stores ten values. The program then uses a For loop to display the array's contents. In Chapter 3, you will examine classes that let you group information, functions, and methods in a data structure. When you create an array in Visual Basic .NET programs, the array is actually a class object that stores information about the array (such as the number of elements the array contains) and provides methods programs can use to manipulate the array. This program illustrates several of the properties and methods the class provides:

```
Module Module1

    Sub Main()
        Dim Values() As Integer = {100, 200, 300, 400, 500}
        Dim MyValues(5) As Integer
        Dim Prices() As Double = {25.5, 4.95, 33.4}

        Dim I As Integer

        For I = 0 To 4
            Console.Write(Values(I) & " ")
        Next
        Console.WriteLine()

        ' Copy one array to another
        Values.CopyTo(MyValues, 0)
        For I = 0 To 4
            Console.Write(MyValues(I) & " ")
        Next
        Console.WriteLine()

        Values.Reverse(Values)
        For I = 0 To 4
            Console.Write(Values(I) & " ")
        Next
        Console.WriteLine()

        Console.WriteLine("Array length: " & Values.Length)
        Console.WriteLine("Array lowerbound: " & _
          Values.GetLowerBound(0))
        Console.WriteLine("Array upperbound: " & _
          Values.GetUpperBound(0))

        For I = 0 To Prices.GetUpperBound(0)
            Console.Write(Prices(I) & " ")
        Next
        Console.WriteLine()

        Console.ReadLine()
    End Sub

End Module
```

After you compile and execute this program, your screen will display the following output:

```
100 200 300 400 500
100 200 300 400 500
500 400 300 200 100
Array length: 5
Array lowerbound: 0
Array upperbound: 4
25.5 4.95 33.4
```

Grouping Values in a Structure

In the Tip titled, "Storing Multiple Values of the Same Type in a Single Variable," you learned how to group multiple values of the same type into an array. In an array, you can only store values of the same type, such as all Integer values, all Double values, and so on. Depending on the processing your programs perform, there may be times when your programs can use a program-defined data type called a structure to group related pieces of information. Unlike an array, which can only store values of the same type, a structure can store several values of different types.

Assume that your program must store information about a book, such as the title, author, publisher, and price. To do so, your programs can declare the following variables:

```
Dim Title As String
Dim Author As String
Dim Publisher As String
Dim Price As Double
```

Next, assume that your program creates several functions and subroutines that use the book information. In your code, you can pass the variables to a subroutine or function as parameters:

```
ShowBook(Title, Author, Publisher, Price)
```

Although passing the variables as parameters in this way lets the subroutines and functions work with the book information, assume that the user changes the program's requirements and you must now also track the number of pages and chapters in each book. In such cases, you could declare two new variables to store the page and chapter information and then change each and every function and subroutine to accept the information as parameters:

```
Dim PageCount As Integer
Dim ChapterCount As Integer

ShowBook(Title, Author, Publisher, Price, PageCount, ChapterCount)
```

As an alternative, your program can define a Book data structure in which you specify the information the structure must hold:

```
Structure Book
  Dim Title As String
  Dim Author As String
  Dim Publisher As String
  Dim Price As Double
End Structure
```

A structure is simply a type that groups related information. After you define the Book structure, your program can declare a variable of the Book type:

```
Dim BookInfo As Book
```

Next, your program can use the dot operator (which separates the variable name from a member variable) to assign values to each member variable:

```
BookInfo.Title = "Visual Basic .NET Programming Tips & Techniques"
Book.Author = "Jamsa"
Book.Publisher = "McGraw-Hill/Osborne"
Book.Price = 49.99
```

Then, rather than passing the individual variables to the function or subroutine, your code can instead pass the structure variable:

```
ShowBook(BookInfo)
```

Should the user change the program's requirement in the future, so that the program must also track the book's copyright date and weight, you can simply add the variables to the Book definition without having to change the subroutine calls:

```
Structure Book
  Dim Title As String
  Dim Author As String
  Dim Publisher As String
  Dim Price As Double
  Dim Copyright As String
  Dim Weight As Double
End Structure
```

USE IT The following program, StructureDemo.vb, creates the Book structure and then uses the structure to create a variable named BookInfo. Using the dot operator, the program then assigns values to each of the structure's members. Finally, the code passes the structure variable to a subroutine that uses the dot operator to display the member's values:

```
Module Module1

   Structure Book
```

```
    Dim Title As String
    Dim Author As String
    Dim Publisher As String
    Dim Price As Double
  End Structure

  Sub ShowBook(ByVal SomeBook As Book)
    Console.WriteLine(SomeBook.Title)
    Console.WriteLine(SomeBook.Author)
    Console.WriteLine(SomeBook.Publisher)
    Console.WriteLine(SomeBook.Price)
  End Sub

  Sub Main()
    Dim BookInfo As Book
    BookInfo.Title = "Visual Basic .Net Programming Tips & Techniques"
    BookInfo.Author = "Jamsa"
    BookInfo.Publisher = "McGraw-Hill/Osborne"
    BookInfo.Price = 49.99

    ShowBook(BookInfo)
    Console.ReadLine()
  End Sub
End Module
```

After you compile and execute this program, your screen will display the following output:

```
Visual Basic .Net Programming Tips & Techniques
Jamsa
McGraw-Hill/Osborne
49.99
```

Improving Your Code's Readability Using Constants

In most programs, you must often use numeric values in a variety of ways. You might use a numeric value to control a For loop's processing, as shown here:

```
For I = 0 To 50
  Console.WriteLine(I)
Next
```

Or you might use a numeric value in an If statement to perform a comparison:

```
If (I = 50) Then
  Console.WriteLine("Processing the last value")
End If
```

Further, you might use a numeric constant to define the size of an array variable (which stores multiple values of the same type in the same variable):

```
Dim Students(50) As Integer
```

The following program, UseNumbers.vb, uses numeric constants throughout the source code. If you examine the program statements, you will find that the code makes extensive use of the value 50:

```
Module Module1

    Sub Main()
        Dim Students(50) As Integer
        Dim I As Integer

        For I = 0 To 50
            Students(I) = I
        Next

        For I = 0 To 50
            Console.WriteLine(Students(I))
        Next

        Console.ReadLine() ' Pause to view output
    End Sub

End Module
```

Rather than use numeric values in this way, your programs should take advantage of constants that assign a meaningful name to the value, such as a constant named NumberOfStudents.

USE IT To create a constant, place a Const statement in your code similar to the following:

```
Const NumberOfStudents As Integer = 50
```

Then, in your program, use the constant name everywhere you would normally use the numeric constant. For example, you might declare arrays as follows:

```
Dim Students(NumberOfStudents) As Integer
```

Likewise, in the For loop, you would use the constant to express the loop's ending value as shown here:

```
For I = 0 To NumberOfStudents
  Students(I) = I
Next
```

If you compare the For loops, you will find that the use of the constant gives another programmer who reads your code more insight into the program's processing. With a glance at the loop, the programmer knows that the array will iterate (loop) through each of the students.

Using constants also simplifies your program should the number of students change from 50 to 100. In the first program, you would need to change each occurrence of the value 50 to the value 100. Each time you make a change to a program, you increase the likelihood of introducing an error (such as a typo that places the value 10 instead of 100 in the code). If your code uses a constant, you need only change the following statement:

```
Const NumberOfStudents As Integer = 100
```

Behind the scenes, the Visual Basic .NET compiler as it compiles your source code will perform the constant substitution for you.

Summarizing the Differences Between Visual Basic and Visual Basic .NET

If you are an experienced Visual Basic .NET programmer, you may have skipped many of the foundation Tips this chapter provides. This Tip will summarize many of the key differences between Visual Basic and Visual Basic .NET.

- Visual Basic .NET does not support the Variant data type. In Chapter 4, you will learn that all .NET classes inherit the System.Object type.
- Visual Basic .NET does not support the Currency data type. Instead, programs should use the Decimal type.
- Visual Basic .NET does not support the use of the LET statement to assign a value to a variable. Instead, your programs should simply use the assignment operator.
- Visual Basic .NET does not support the DefType statement that previous versions of Visual Basic used to define the program's default data type. As a matter of good programming practices, your programs should declare each and every variable.
- Visual Basic .NET does not support user-defined types. Instead, your programs should use a Structure (or class) to group related information.
- Visual Basic .NET no longer supports the IsMissing function. Instead, your programs should use IsNothing to determine whether an object contains a value.

- Visual Basic .NET does not support the use of the GoSub statement to call a subroutine. Instead, your program should simply call the subroutine by referencing the subroutine name followed by parentheses, which may contain optional parameters.
- Visual Basic .NET does not support Static subroutines or functions. If a variable in a subroutine or function must maintain its value from one invocation to the next, the routine should declare the variable as Static.
- Visual Basic .NET does not let programmers declare fixed length String variables.
- Visual Basic .NET changes the type Integer to 32 bits, which lets Integer variables store values in the range –2,147,483,648 to 2,147,483,647.
- Visual Basic .NET uses the type Short to represent 16-bit values, which can store numbers in the range –32,768 to 32,767.
- Visual Basic .NET changes the lower bound of an array to element 0 (as opposed to element 1). A Visual Basic .NET program cannot use the Option Base statement to specify the default array base. Visual Basic .NET does not let you specify an array's lower and upper bounds when you declare an array. All Visual Basic .NET arrays use the lower bound 0.
- Visual Basic .NET uses Math class methods to perform arithmetic and trigonometric operations such as calculating a value's square root or an angle's sine or cosine.
- When a Visual Basic .NET program calls a function or subroutine, the program must specify parentheses after the routine's name, even if the routine does not use parameters.
- By default, Visual Basic .NET passes variables to functions and subroutines by value (using the ByVal keyword), which means the routine cannot change the original variable's value. If a subroutine or function must change a variable, you must pass the variable to the routine by reference (using the ByRef keyword).
- Although Visual Basic .NET supports the MsgBox function, most newer programs will use the MessageBox class Show method.
- Visual Basic .NET replaces the Wend statement that indicates the end of a While loop with the End While statement.
- Visual Basic .NET replaces the Debug.Print statement with Debug.Writeline.
- Visual Basic .NET lets a program declare variables in a construct, such as a While loop or If statement. The variable's scope, in turn, corresponds to the construct.
- Visual Basic .NET uses the value Nothing to indicate that an object does not contain a value. Visual Basic .NET does not support Null or Empty.