Control Structures

Objectives

• To use the *If/Then* and *If/Then/Else* selection structures to choose among alternative actions.
• To use the *While, Do While/Loop* and *Do Until/Loop* repetition structures to execute statements in a program repeatedly.
• To use the assignment operators.
• To use the *For/Next, Do/Loop While* and *Do/Loop Until* repetition structures to execute statements in a program repeatedly.
• To understand multiple selection as implemented by the *Select Case* selection structure.
• To use the *Exit Do* and *Exit For* program control statements.
• To use logical operators.

*Who can control his fate?*
William Shakespeare

*Man is a tool-making animal.*
Benjamin Franklin

*Intelligence... is the faculty of making artificial objects, especially tools to make tools.*
Henri Bergson

*Let’s all move one place on.*
Lewis Carroll
3.1 Introduction

In this chapter, we present control structures that enable programmers to control the order of events in their programs. Visual Basic’s sequence, selection and repetition structures are used to select and repeat various statements and thereby execute complex algorithms. In the process, we introduce commonly used shorthand operators that allow the programmer quickly to calculate and assign new values to variables. When we study object-based programming in more depth in Chapter 5, we will see that control structures are helpful in building and manipulating objects.

3.2 Control Structures

Normally, statements in a program are executed one after another in the order in which they are written. This process is called sequential execution. However, various Visual Basic statements enable the programmer to specify that the next statement to be executed might not be the next one in sequence. A transfer of control occurs when a statement other than the next one in the program executes.

Visual Basic provides seven types of repetition structures: While, Do While/Loop, Do/Loop While, Do Until/Loop, Do/Loop Until, For/Next and For Each/Next. (For Each/Next is covered in Chapter 4, Procedures and Arrays.) The words If, Then,
Else, End, Select, Case, While, Do, Until, Loop, For, Next and Each are all Visual Basic keywords. (See Fig. 3.1 for a list of all the Visual Basic keywords.) We discuss many of Visual Basic’s keywords and their respective purposes throughout this book.

<table>
<thead>
<tr>
<th>Visual Basic Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddHandler AddHandler</td>
</tr>
<tr>
<td>AndAlso AndAlso</td>
</tr>
<tr>
<td>Auto Auto</td>
</tr>
<tr>
<td>ByVal ByVal</td>
</tr>
<tr>
<td>CBool CBool</td>
</tr>
<tr>
<td>CDec CDec</td>
</tr>
<tr>
<td>Class Class</td>
</tr>
<tr>
<td>CShort CShort</td>
</tr>
<tr>
<td>Date Date</td>
</tr>
<tr>
<td>Delegate Delegate</td>
</tr>
<tr>
<td>Double Double</td>
</tr>
<tr>
<td>End End</td>
</tr>
<tr>
<td>Event Event</td>
</tr>
<tr>
<td>For For</td>
</tr>
<tr>
<td>GetGetType GetType</td>
</tr>
<tr>
<td>Implements Implements</td>
</tr>
<tr>
<td>Integer Integer</td>
</tr>
<tr>
<td>Like Like</td>
</tr>
<tr>
<td>Mod Mod</td>
</tr>
<tr>
<td>MyBase MyBase</td>
</tr>
<tr>
<td>Next Next</td>
</tr>
<tr>
<td>NotNotOverridable NotOverridable</td>
</tr>
<tr>
<td>Optional Optional</td>
</tr>
<tr>
<td>Overridable Overridable</td>
</tr>
<tr>
<td>Private Private</td>
</tr>
<tr>
<td>RaiseEvent RaiseEvent</td>
</tr>
<tr>
<td>RemoveHandler RemoveHandler</td>
</tr>
<tr>
<td>Set Set</td>
</tr>
<tr>
<td>Single Single</td>
</tr>
<tr>
<td>String String</td>
</tr>
<tr>
<td>Then Then</td>
</tr>
</tbody>
</table>

Fig. 3.1 Keywords in Visual Basic. (Part 1 of 2.)
### 3.3 If/Then Selection Structure

In a program, a selection structure chooses among alternative courses of action. For example, suppose that the passing grade on an examination is 60 (out of 100). Then the Visual Basic .NET code

```vbnet
If studentGrade >= 60
    Console.WriteLine("Passed")
End If
```

determines whether the condition `studentGrade >= 60` is true or false. If the condition is true, then “Passed” is printed, and the next statement in order is “performed.” If the condition is false, the `Console.WriteLine` statement is ignored, and the next statement in order is performed. A decision can be made on any expression that evaluates to a value of Visual Basic’s `Boolean` type (i.e., any expression that evaluates to `True` or `False`).

### 3.4 If/Then/Else Selection Structure

The `If/Then` selection structure performs an indicated action only when the condition evaluates to true; otherwise, the action is skipped. The `If/Then/Else` selection structure allows the programmer to specify that a different action be performed when the condition is true than that performed when the condition is false. For example, the statement

```vbnet
If studentGrade >= 60
    Console.WriteLine("Passed")
Else
    Console.WriteLine("Failed")
End If
```

prints “Passed” if the student’s grade is greater than or equal to 60 and prints “Failed” if the student’s grade is less than 60. In either case, after printing occurs, the next statement in sequence is “performed.”
3.5 While Repetition Structure

A repetition structure allows the programmer to specify that an action be repeated a number of times, depending on the value of a condition. Visual Basic .NET provides seven repetition structures—one of which is the While repetition structure.

As an example of a While structure, consider a program segment designed to find the first power of two larger than 1000. Suppose Integer variable product contains the value 2. When the following While structure finishes executing, product contains the result:

```vbnet
Dim product As Integer = 2
While product <= 1000
    product = product * 2
End While
```

When the While structure begins executing, product is 2. Variable product is multiplied by 2 repeatedly, taking on the values 4, 8, 16, 32, 64, 128, 256, 512 and 1024, successively. When product becomes 1024, the condition product <= 1000 in the While structure becomes false. This condition causes the repetition to terminate, with 1024 as product’s final value. Execution continues with the next statement after the While structure. [Note: If a While structure’s condition is initially false, the body statement(s) will never be executed.]

3.6 Do While/Loop Repetition Structure

The Do While/Loop repetition structure behaves like the While repetition structure. As an example of a Do While/Loop structure, consider another version of the segment designed to find the first power of two larger than 1000:

```vbnet
Dim product As Integer = 2
Do While product <= 1000
    product = product * 2
Loop
```

When the Do While/Loop structure is entered, the value of product is 2. The variable product is multiplied by 2 repeatedly, taking on the values 4, 8, 16, 32, 64, 128, 256, 512 and 1024 successively. When product becomes 1024, the condition in the Do While/Loop structure, product <= 1000, becomes false. This condition causes the repetition to terminate, with 1024 as product’s final value. Program execution continues with the next statement after the Do While/Loop structure.

3.7 Do Until/Loop Repetition Structure

Unlike the While and Do While/Loop repetition structures, the Do Until/Loop repetition structure tests a condition for falsity for repetition to continue. Statements in the body of a Do Until/Loop are executed repeatedly as long as the loop-continuation test evaluates to false. As an example of a Do Until/Loop repetition structure, once again consider the segment designed to find the first power of two larger than 1000:
Dim product As Integer = 2

Do Until product >= 1000
    product = product * 2
Loop

3.8 Do/Loop While Repetition Structure

The Do/Loop While repetition structure is similar to the While and Do While/Loop structures. In the While and Do While/Loop structures, the loop-continuation condition is tested at the beginning of the loop, before the body of the loop always is performed. The Do/Loop While structure tests the loop-continuation condition after the body of the loop is performed. Therefore, in a Do/Loop While structure, the body of the loop is always executed at least once. When a Do/Loop While structure terminates, execution continues with the statement after the Loop While clause. As an example of a Do/Loop While repetition structure, once again consider the segment designed to find the first power of two larger than 1000:

Dim product As Integer = 1

Do
    product = product * 2
Loop While product <= 1000

3.9 Do/Loop Until Repetition Structure

The Do/Loop Until structure is similar to the Do Until/Loop structure, except that the loop-continuation condition is tested after the body of the loop is performed; therefore, the body of the loop executes at least once. When a Do/Loop Until terminates, execution continues with the statement after the Loop Until clause. As an example of a Do/Loop Until repetition structure, once again consider the segment designed to find the first power of two larger than 1000:

Dim product As Integer = 1

Do
    product = product * 2
Loop Until product >= 1000

3.10 Assignment Operators

Visual Basic .NET provides several assignment operators for abbreviating assignment statements. For example, the statement

value = value + 3

can be abbreviated with the addition assignment operator (+=) as

value += 3
The `+=` operator adds the value of the right operand to the value of the left operand and stores the result in the left operand’s variable. Any statement of the form

```
variable = variable operator expression
```

where `operator` is one of the binary operators `+`, `-`, `*`, `^`, `&`, `/` or `\`, can be written in the form

```
variable operator = expression
```

Figure 3.2 lists the arithmetic assignment operators and provides sample expressions using these operators and corresponding explanations.

Although the symbols `=`, `+=`, `-=` `*=` `/=` `\=` `^=` and `&=` are operators, we do not include them in operator-precedence tables. When an assignment statement is evaluated, the expression to the right of the operator always is evaluated first and subsequently assigned to the variable on the left. Unlike Visual Basic’s other operators, the assignment operators can occur only once in a statement.

### 3.11 For/Next Repetition Structure

The **For/Next** repetition structure handles the details of counter-controlled repetition. The example in Fig. 3.3 uses the **For/Next** structure to display the even digits from 2–10.

<table>
<thead>
<tr>
<th>Assignment operator</th>
<th>Sample expression</th>
<th>Explanation</th>
<th>Assigns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+=</code></td>
<td><code>c += 7</code></td>
<td><code>c = c + 7</code></td>
<td>11 to <code>c</code></td>
</tr>
<tr>
<td><code>-=</code></td>
<td><code>c -= 3</code></td>
<td><code>c = c - 3</code></td>
<td>1 to <code>c</code></td>
</tr>
<tr>
<td><code>*=</code></td>
<td><code>c *= 4</code></td>
<td><code>c = c * 4</code></td>
<td>16 to <code>c</code></td>
</tr>
<tr>
<td><code>/=</code></td>
<td><code>c /= 2</code></td>
<td><code>c = c / 2</code></td>
<td>2 to <code>c</code></td>
</tr>
<tr>
<td><code>\=</code></td>
<td><code>c \= 3</code></td>
<td><code>c = c \ 3</code></td>
<td>1 to <code>c</code></td>
</tr>
<tr>
<td><code>^=</code></td>
<td><code>c ^= 2</code></td>
<td><code>c = c ^ 2</code></td>
<td>16 to <code>c</code></td>
</tr>
<tr>
<td><code>&amp;=</code></td>
<td><code>d &amp;= &quot;llo&quot;</code></td>
<td><code>d = d &amp; &quot;llo&quot;</code></td>
<td>&quot;Hello&quot; to <code>d</code></td>
</tr>
</tbody>
</table>

**Fig. 3.2** Assignment operators.

```vbnet
' Fig. 3.3: ForCounter.vb
' Using the For/Next structure to demonstrate counter-controlled repetition.
Module modForCounter
    Sub Main()
        Dim counter As Integer
```
The Main procedure of the program operates as follows: When the For/Next structure (lines 12–14) begins its execution, the control variable counter is initialized to 2, thus addressing the first two elements of counter-controlled repetition—control variable name and initial value. Next, the implied loop-continuation condition counter <= 10 is tested. The To keyword is required in the For/Next structure. The optional Step keyword specifies the increment (i.e., the amount that is added to counter each time the body of the For/Next structure is executed). The increment of a For/Next structure could be negative, in which case it is a decrement, and the loop actually counts downwards. If Step and the value following it are omitted, the increment defaults to 1. Thus, programmers typically omit the Step portion for increments of 1.

Because, the initial value of counter is 2, the implied condition is satisfied (i.e., True), and the counter’s value (2) is output in line 13. The required Next keyword marks the end of the For/Next repetition structure. When the Next keyword is reached, variable counter is incremented by the specified value of 2, and the loop begins again with the loop-continuation test.

At this point, the control variable is equal to 4. This value does not exceed the final value, so the program performs the body statement again. This process continues until the counter value of 10 has been printed and the control variable counter is incremented to 12, causing the loop-continuation test to fail and repetition to terminate. The program continues by performing the first statement after the For/Next structure. (In this case, procedure Main terminates, because the program reaches the End Sub statement on line 16.)

### 3.12 Example: Using the For/Next Structure to Compute Compound Interest

The next example computes compound interest, using the For/Next structure. Consider the following problem:

A person invests $1000.00 in a savings account that yields 5% interest. Assuming that all interest is left on deposit, calculate and print the amount of money in the account at the end of each year, over a period of 10 years. To determine these amounts, use the following formula:

\[ a = p (1 + r)^n \]
where

\[ p \text{ is the original amount invested (i.e., the principal)} \]

\[ r \text{ is the annual interest rate (e.g., .05 stands for 5%)} \]

\[ n \text{ is the number of years} \]

\[ a \text{ is the amount on deposit at the end of the } n\text{th year.} \]

This problem involves a loop that performs the indicated calculation for each of the 10 years that the money remains on deposit. The solution is shown in Fig. 3.4.

Line 9 declares two \texttt{Decimal} variables. Type \texttt{Decimal} is used for monetary calculations. Line 10 declares \texttt{rate} as type \texttt{Double}, and lines 14–15 initialize \texttt{principal} to 1000.00 and \texttt{rate} to 0.05 (i.e., 5%).

The body of the \texttt{For/Next} structure is executed 10 times, varying control variable \texttt{year} from 1 to 10 in increments of 1. Line 21 performs the calculation from statement of the problem, that is,

\[ a = p (1 + r)^n \]

where \(a\) is \texttt{amount}, \(p\) is \texttt{principal}, \(r\) is \texttt{rate} and \(n\) is \texttt{year}.

---

1 ' Fig. 3.4: Interest.vb
2 ' Calculating compound interest.
3 Imports System.Windows.Forms
4 Module modInterest
5 Sub Main()
6     Dim amount, principal As Decimal ' dollar amounts
7     Dim rate As Double ' interest rate
8     Dim year As Integer ' year counter
9     Dim output As String ' amount after each year
10
11     principal = 1000.00
12     rate = 0.05
13
14     output = "Year" & vbTab & "Amount on deposit" & vbCrLf
15
16     ' calculate amount after each year
17     For year = 1 To 10
18         amount = principal * (1 + rate) ^ year
19         output &= year & vbTab & _
20         String.Format("{0:C}", amount) & vbCrLf
21     Next
22
23     ' display output
24     MessageBox.Show(output, "Compound Interest", _
25         MessageBoxButtons.OK, MessageBoxIcon.Information)
26 End Sub ' Main
27 End Module ' modInterest

Fig. 3.4 \texttt{For/Next} structure used to calculate compound interest. (Part 1 of 2.)
Lines 22–23 append additional text to the end of String output. The text includes the current value of year, a tab character (vbTab) to position the cursor to the second column, the result of the method call String.Format("{0:C}", amount) and, finally, a newline character (vbCrLf) to start the next output on the next line. The first argument passed to Format is the format string. Previously in the text, we have seen Strings containing {0}, {1} and so on, where the digit within the braces indicates the argument being displayed. In more complicated format strings, such as "{0:C}" , the first digit (0) serves the same purpose. The information specified after the colon (:) is called the formatting code. The C (for “currency”) formatting code indicates that its corresponding argument (amount) should be displayed in monetary format. Figure 3.5 explains several formatting codes; a complete list can be found in the MSDN documentation “Standard Numeric Format Strings.” All formatting codes are case insensitive. Note that format codes D and X can be used only with integer values. [Note: Method Format uses .NET’s string formatting codes to represent numeric and monetary values according to the user’s localization settings.1 For example, in the United States, an amount would be expressed in dollars (e.g., $634,307.08), while in Malaysia, the amount would be expressed in ringgits (e.g., R634,307.08).]

<table>
<thead>
<tr>
<th>Format Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Currency. Precedes the number with $, separates every three digits with commas and sets the number of decimal places to two.</td>
</tr>
</tbody>
</table>

Fig. 3.4 For/Next structure used to calculate compound interest. (Part 2 of 2.)

Fig. 3.5 Formatting codes for Strings. (Part 1 of 2.)

1. Localization is the customization of software (e.g., an operating system) to display information by means of the customs and languages of a geographical region. Localization settings can be customized through the Start Menu by selecting Control Panel > Regional and Language Options > Regional Options in Windows XP and by selecting Control Panel > Regional Options in Windows 2000.
Variables `amount` and `principal` are of type `Decimal`. We use this type because we are dealing with fractional parts of dollars and need a type that allows precise calculations with monetary amounts; `Single` and `Double` do not.

**Good Programming Practice 3.1**

*Do not use variables of type `Single` or `Double` to perform precise monetary calculations. The imprecision of floating-point numbers can cause errors that result in incorrect monetary values. Use the data type `Decimal` for monetary calculations.*

Variable `rate` is also of type `Double`, because it is used in the calculation `1.0 + rate`, which appears as the right operand of the exponentiation operator. In fact, this calculation produces the same result each time through the loop, so performing the calculation in the body of the `For/Next` loop is wasteful.

**Performance Tip 3.1**

*Avoid placing inside a loop the calculation of an expression whose value does not change each time through the loop. Such an expression should be evaluated only once, prior to the loop.*

The FCL and Visual Basic .NET allow multiple procedures with the same name to be defined. Programmers often use this technique, known as *procedure overloading*, to define several procedures that perform similar actions, but take a different set of arguments. One example of the use of procedure overloading is method `MessageBox.Show`. In a previous example, we have provided only one argument to this method—a `String` to be displayed in a message dialog. Figure 3.4 uses a version of method `MessageBox.Show` (lines 27–28) that takes four arguments. The dialog in the output of Fig. 3.4 illustrates the four arguments. The first argument is the message to display. The second argument is the string to display in the dialog’s title bar. The third argument is a value indicating which button(s) to display. The fourth argument indicates which icon to display to the left of the message. Figures 3.6 and 3.7
provide a listing of the `MessageBoxButtons` and `MessageBoxIcon` choices. Information about other versions of method `MessageBox.Show` can be found in the MSDN documentation provided with Visual Studio .NET. We discuss procedure overloading in more detail in Chapter 4, Procedures and Arrays.

<table>
<thead>
<tr>
<th><code>MessageBoxIcon</code> Constants</th>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MessageBoxIcon.Exclamation</code></td>
<td><img src="image" alt="Exclamation Point Icon" /></td>
<td>Icon containing an exclamation point. Typically used to warn the user of potential problems.</td>
</tr>
<tr>
<td><code>MessageBoxIcon.Information</code></td>
<td><img src="image" alt="Letter 'i' Icon" /></td>
<td>Icon containing the letter “i.” Typically used to display information about the state of the application.</td>
</tr>
<tr>
<td><code>MessageBoxIcon.Question</code></td>
<td><img src="image" alt="Question Mark Icon" /></td>
<td>Icon containing a question mark. Typically used to ask the user a question.</td>
</tr>
<tr>
<td><code>MessageBoxIcon.Error</code></td>
<td><img src="image" alt="Error Symbol Icon" /></td>
<td>Icon containing an ∞ in a red circle. Typically used to alert the user of errors or critical situations.</td>
</tr>
</tbody>
</table>

**Fig. 3.6** Message-dialog icon constants.

<table>
<thead>
<tr>
<th><code>MessageBoxButtons</code> constants</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MessageBoxButtons.OK</code></td>
<td>OK button. Allows the user to acknowledge a message. Included by default.</td>
</tr>
<tr>
<td><code>MessageBoxButtons.OKCancel</code></td>
<td>OK and Cancel buttons. Allows the user to either continue or cancel an operation.</td>
</tr>
<tr>
<td><code>MessageBoxButtons.YesNo</code></td>
<td>Yes and No buttons. Allows the user to respond to a question.</td>
</tr>
<tr>
<td><code>MessageBoxButtons.YesNoCancel</code></td>
<td>Yes, No and Cancel buttons. Allows the user to respond to a question or cancel an operation.</td>
</tr>
<tr>
<td><code>MessageBoxButtons.RetryCancel</code></td>
<td>Retry and Cancel buttons. Typically used to allow the user either to retry or to cancel an operation that has failed.</td>
</tr>
<tr>
<td><code>MessageBoxButtons.AbortRetryIgnore</code></td>
<td>Abort, Retry and Ignore buttons. When one of a series of operations has failed, these buttons allow the user to abort the entire sequence, retry the failed operation, or ignore the failed operation and continue.</td>
</tr>
</tbody>
</table>

**Fig. 3.7** Message-dialog button constants.
3.13 Select Case Multiple-Selection Structure

Occasionally, an algorithm contains a series of decisions in which the algorithm tests a variable or expression separately for each value that the variable or expression might assume. The algorithm then takes different actions based on those values. Visual Basic provides the **Select Case** multiple-selection structure to handle such decision making. The program in Fig. 3.8 uses a Select Case structure to count the number of different letter grades on an exam. Assume that the exam is graded as follows: 90 and above is an “A,” 80–89 is a “B,” 70–79 is a “C,” 60–69 is a “D” and 0–59 is an “F.” The instructor generously gives a minimum grade of 10 for students who were present for the exam. Students not present for the exam receive a 0.

Line 7 in Fig. 3.8 declares variable *grade* as type *Integer*. This variable stores each grade that is input. Lines 8–12 declare variables that store the total number grades of each type. Lines 18–57 use a **While** loop to control repetition.

Line 20,

```
    Select Case grade
```

begins the Select Case structure. The expression following the keywords **Select Case** is called the **controlling expression**. The controlling expression (i.e., the value of *grade*) is compared sequentially with each **Case**. If a matching **Case** is found, the code in the **Case** executes, and program control proceeds to the first statement after the **Select Case** structure (line 55).

**Common Programming Error 3.1**

Duplicate Case statements are logic errors. At execution time, the first matching Case statement is executed.

The first **Case** statement (line 22) determines whether the value of *grade* is exactly equal to 100. The next **Case** statement (line 27) determines whether *grade* is between 90 and 99, inclusive. Keyword **To** specifies the range. Lines 31–44 use this keyword to present a series of similar **Cases**.

**Common Programming Error 3.2**

If the value on the left side of the **To** keyword in a **Case** statement is larger than the value on the right side, the **Case** statement is ignored during program execution, potentially causing a logic error.

---

```
1    ' Fig. 3.8: SelectTest.vb
2    ' Using the Select Case structure.
3
4 Module modEnterGrades
5
6    Sub Main()
7        Dim grade As Integer = 0   ' one grade
8        Dim aCount As Integer = 0  ' number of As
9        Dim bCount As Integer = 0  ' number of Bs
10       Dim cCount As Integer = 0  ' number of Cs
11       Dim dCount As Integer = 0  ' number of Ds
```

**Fig. 3.8** Select Case structure used to count grades. (Part 1 of 3.)
Dim fCount As Integer = 0 ' number of Fs

Console.Write("Enter a grade, -1 to quit: ")
grade = Console.ReadLine()

' input and process grades
While grade <> -1

    Select Case grade ' determine which grade was input

        Case 100 ' student scored 100
            Console.WriteLine("Perfect Score!" & vbCrLf & "Letter grade: A" & vbCrLf)
aCount += 1

        Case 90 To 99 ' student scored 90-99
            Console.WriteLine("Letter Grade: A" & vbCrLf)
aCount += 1

        Case 80 To 89 ' student scored 80-89
            Console.WriteLine("Letter Grade: B" & vbCrLf)
bCount += 1

        Case 70 To 79 ' student scored 70-79
            Console.WriteLine("Letter Grade: C" & vbCrLf)
cCount += 1

        Case 60 To 69 ' student scored 60-69
            Console.WriteLine("Letter Grade: D" & vbCrLf)
dCount += 1

        Case 0, 10 To 59 ' student scored 0 or 10-59 (10 points for attendance)
            Console.WriteLine("Letter Grade: F" & vbCrLf)
fCount += 1

        Case Else

            ' alert user that invalid grade was entered
            Console.WriteLine("Invalid Input. " & 
                "Please enter a valid grade." & vbCrLf)
    End Select

    Console.Write("Enter a grade, -1 to quit: ")
    grade = Console.ReadLine()
End While

' display count of each letter grade
Console.WriteLine(vbCrLf & "Totals for each letter grade are: " & vbCrLf & 
    "A: " & aCount & vbCrLf & "B: " & bCount & 
    "C: " & cCount & vbCrLf & "D: " & dCount & 
    "F: " & fCount)

Fig. 3.8  Select Case structure used to count grades. (Part 2 of 3.)
When multiple values are tested in a `Case` statement, they are separated by commas. On line 44, either `0` or any value in the range `10` to `59`, inclusive, matches to this `Case`. Line 48 contains the optional `Case Else`, which is executed when the input does not match the controlling expression in any of the previous `Cases`. `Case Else` commonly is used to check for invalid input. When employed, the `Case Else` must be the last `Case`. The required `End Select` keywords terminate the `Select Case` structure.

**Common Programming Error 3.3**

When using the optional `Case Else` statement in a `Select Case` structure, failure to place the `Case Else` as the last `Case` is a syntax error.

`Case` statements also can use relational operators to determine whether the controlling expression satisfies a condition. For example,
Case Is < 0 uses keyword \textit{Is} along with the relational operator < to test for values less than 0.

\textbf{Testing and Debugging Tip 3.1}

Provide a \textit{Case Else} in Select Case structures. Cases not handled in a Select Case structure are ignored unless a Case Else is provided. The inclusion of a Case Else statement facilitates the processing of exceptional conditions. In some situations, no Case Else processing is needed.

3.14 Using the Exit Keyword in a Repetition Structure

The \textit{Exit Do}, \textit{Exit While} and \textit{Exit For} statements alter the flow of control by causing immediate exit from a repetition structure. The \textit{Exit Do} statement can be executed in a Do While/Loop, Do/Loop While, Do Until/Loop or Do/Loop Until structure and causes the program to exit immediately from that repetition structure. Similarly, the \textit{Exit For} and \textit{Exit While} statements cause immediate exit from For/Next and While loops, respectively. Execution continues with the first statement that follows the repetition structure. Figure 3.9 demonstrates the \textit{Exit For}, \textit{Exit Do} and \textit{Exit While} statements in various repetition structures.

```
' Fig. 3.9: ExitTest.vb
' Using the Exit keyword in repetition structures.

Imports System.Windows.Forms

Module modExitTest

Sub Main()
    Dim output As String
    Dim counter As Integer

    For counter = 1 To 10
        ' skip remaining code in loop only if counter = 3
        If counter = 3 Then
            Exit For
        End If

        Next

    output = \
        "counter = " & counter & _
        " after exiting For/Next structure" & vbCrLf

    Do Until counter > 10
        ' skip remaining code in loop only if counter = 5
        If counter = 5 Then
            Exit Do
        End If

    End If

Fig. 3.9 Exit keyword in repetition structures. (Part 1 of 2.)
```
The header of the **For/Next** structure (line 12) indicates that the body of the loop should execute 10 times. During each execution, the **If/Then** structure (lines 15–17) determines if the control variable, `counter`, is equal to 3. If so, the **Exit For** statement (line 16) executes. Thus, as the body of the **For/Next** structure executes for the third time (i.e, `counter` is 3), the **Exit For** statement terminates execution of the loop. Program control then proceeds to the assignment statement (lines 21–22), which appends the current value of `counter` to **String** variable `output`.

The header of the **Do Until/Loop** structure (line 24) indicates that the loop should continue executing until `counter` is greater than 10. (Note that `counter` is 3 when the **Do Until/Loop** structure begins executing.) When `counter` has the values 3 and 4, the body of the **If/Then** structure (lines 27–29) does not execute, and `counter` is incremented (line 31). However, when `counter` is 5, the **Exit Do** statement (line 28) executes, terminating the loop. The assignment statement (lines 34–35) appends the value of `counter` to **String** variable `output`. Note that the program does not increment `counter` (line 31) after the **Exit Do** statement executes.
The **While** structure (lines 37–45) behaves similarly to the **Do While/Loop**. In this case, the value of `counter` is 5 when the loop begins executing. When `counter` is 7, the **Exit While** statement (line 41) executes, terminating execution of the **While** structure. Lines 47–48 append the final value of `counter` to **String** variable `output`, which is displayed in a message dialog (lines 50–51).

### Software Engineering Observation 3.1
Some programmers feel that **Exit Do**, **Exit While** and **Exit For** violate the principles of structured programming.

### Software Engineering Observation 3.2
Debates abound regarding the relative importance of quality software engineering and program performance. Often, one of these goals is accomplished at the expense of the other. For all but the most performance-intensive situations, apply the following guidelines: First, make your code simple and correct; then make it fast and small, but only if necessary.

### 3.15 Logical Operators

So far, we have studied only simple conditions, such as `count <= 10`, `total > 1000` and `number <> sentinelValue`. Each selection and repetition structure evaluated only one condition with one of the operators `>`, `<`, `>=`, `<=`, `=` and `<>`. To make a decision that relied on the evaluation of multiple conditions, we performed these tests in separate statements or in nested **If/Then** or **If/Then/Else** structures.

To handle multiple conditions more efficiently, Visual Basic provides logical operators that can be used to form complex conditions by combining simple ones. The logical operators are **AndAlso**, **And**, **OrElse**, **Or**, **Xor** and **Not**. We consider examples that use each of these operators.

Suppose we wish to ensure that two conditions are both true in a program before a certain path of execution is chosen. In such a case, we can use the logical **AndAlso** operator as follows:

```vbnet
If gender = "F" AndAlso age >= 65 Then
    seniorFemales += 1
End If
```

This **If/Then** statement contains two simple conditions. The condition `gender = "F"` determines whether a person is female, and the condition `age >= 65` determines whether a person is a senior citizen. The two simple conditions are evaluated first, because the precedences of `=` and `>=` are both higher than the precedence of **AndAlso**. The **If/Then** statement then considers the combined condition

```vbnet
gender = "F" AndAlso age >= 65
```

This condition evaluates to true *if and only if* both of the simple conditions are true. When this combined condition is true, the count of `seniorFemales` is incremented by 1. However, if either or both of the simple conditions are false, the program skips the incrementation step and proceeds to the statement following the **If/Then** structure. The readability of the preceding combined condition can be improved by adding redundant (i.e., unnecessary) parentheses:
(gender = "F") AndAlso (age >= 65)

Figure 3.10 illustrates the effect of using the `AndAlso` operator with two expressions. The table lists all four possible combinations of true and false values for `expression1` and `expression2`. Such tables often are called truth tables. Visual Basic evaluates to true or false expressions that include relational operators, equality operators and logical operators.

Now let us consider the `OrElse` operator. Suppose we wish to ensure that either or both of two conditions are true before we choose a certain path of execution. We use the `OrElse` operator in the following program segment:

```
If (semesterAverage >= 90 OrElse finalExam >= 90) Then
    Console.WriteLine("Student grade is A")
End If
```

This statement also contains two simple conditions. The condition `semesterAverage >= 90` is evaluated to determine whether the student deserves an “A” in the course because of an outstanding performance throughout the semester. The condition `finalExam >= 90` is evaluated to determine whether the student deserves an “A” in the course because of an outstanding performance on the final exam. The `If/Then` statement then considers the combined condition

```
(semesterAverage >= 90 OrElse finalExam >= 90)
```

and awards the student an “A” if either or both of the conditions are true. Note that the text “Student grade is A” is always printed, unless both of the conditions are false. Figure 3.11 provides a truth table for the `OrElse` operator.

<table>
<thead>
<tr>
<th><code>expression1</code></th>
<th><code>expression2</code></th>
<th><code>expression1 AndAlso expression2</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

Fig. 3.10 Truth table for the `AndAlso` operator.

<table>
<thead>
<tr>
<th><code>expression1</code></th>
<th><code>expression2</code></th>
<th><code>expression1 OrElse expression2</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

Fig. 3.11 Truth table for the `OrElse` operator.
The **AndAlso** operator has a higher precedence than the **OrElse** operator. An expression containing **AndAlso** or **OrElse** operators is evaluated only until truth or falsity is known. For example, evaluation of the expression

\[(gender = "F" \text{ AndAlso } age >= 65)\]

stops immediately if \(gender\) is not equal to "F" (i.e., the entire expression is false); the evaluation of the second condition is irrelevant because the first condition is false. Evaluation of the second condition occurs if and only if \(gender\) is equal to "F" (i.e., the entire expression could still be true if the condition \(age >= 65\) is true). This performance feature for the evaluation of **AndAlso** and **OrElse** expressions is called **short-circuit evaluation**.

**Performance Tip 3.2**

In expressions that use operator **AndAlso**, if the separate conditions are independent of one another, place the condition most likely to be false as the leftmost condition. In expressions that use operator **OrElse**, make the condition most likely to be true the leftmost condition. Each of these techniques can reduce a program’s execution time.

The **logical AND** operator without short-circuit evaluation (**And**) and the **logical inclusive OR** operator without short-circuit evaluation (**Or**) are similar to the **AndAlso** and **OrElse** operators, respectively, with one exception: The **And** and **Or** logical operators always evaluate both of their operands. No short-circuit evaluation occurs when **And** and **Or** are employed. For example, the expression

\[(gender = "F" \text{ And } age >= 65)\]

evaluates \(age >= 65\), even if \(gender\) is not equal to "F".

Normally, there is no compelling reason to use the **And** and **Or** operators instead of **AndAlso** and **OrElse**. However, some programmers make use of them when the right operand of a condition produces a side effect (such as a modification of a variable’s value) or when the right operand includes a required method call, as in the following program segment:

```csharp
Console.WriteLine("How old are you?")
If (gender = "F" And Console.ReadLine() >= 65) Then
    Console.WriteLine("You are a female senior citizen.")
End If
```

Here, the **And** operator guarantees that the condition \(\text{Console.ReadLine()} >= 65\) is evaluated, so **ReadLine** is called regardless of whether the overall expression is true. It would be better to write this code as two separate statements; the first would store the result of **Console.ReadLine()** in a variable, and the second would use that variable with the **AndAlso** operator in the condition.

**Testing and Debugging Tip 3.2**

Avoid expressions with side effects in conditions, as side effects often cause subtle errors.

A condition containing the **logical exclusive OR** (**Xor**) operator is true if and only if one of its operands results in a true value and the other results in a false value. If both operands are true or both are false, the entire condition is false. Figure 3.12 presents a truth table for the logical exclusive OR operator (**Xor**). This operator always evaluates both of its operands (i.e., there is no short-circuit evaluation).
Visual Basic’s **Not** (logical negation) operator enables a programmer to “reverse” the meaning of a condition. Unlike the logical operators **AndAlso**, **And**, **OrElse**, **Or** and **Xor**, which each combine two conditions (i.e., they are all binary operators), the logical negation operator is a unary operator, requiring only one operand. The logical negation operator is placed before a condition to choose a path of execution if the original condition (without the logical negation operator) is false. The following program segment demonstrates the logical negation operator:

```vbnet
If Not (grade = sentinelValue) Then
    Console.WriteLine("The next grade is " & grade)
End If
```

The parentheses around the condition `grade = sentinelValue` are necessary, because the logical negation operator (**Not**) has a higher precedence than the equality operator. Figure 3.13 provides a truth table for the logical negation operator.

In most cases, the programmer can avoid the use of logical negation by expressing the condition differently with relational or equality operators. For example, the preceding statement can be written as follows:

```vbnet
If grade <> sentinelValue Then
    Console.WriteLine("The next grade is " & grade)
End If
```

This flexibility aids programmers in expressing conditions more naturally. The console application in Fig. 3.14 demonstrates the use of the logical operators by displaying their truth tables.
' Fig. 3.14: LogicalOperator.vb
' Using logical operators.

Module modLogicalOperator

Sub Main()

' create truth table for AndAlso
Console.Write("AndAlso" & vbCrLf & _
"False AndAlso False: " & (False AndAlso False) & _
vbCrLf & "False AndAlso True: " & _
(True AndAlso False) & vbCrLf & _
"True AndAlso False: " & (True AndAlso False) & _
 vbCrLf & "True AndAlso True: " & _
(True AndAlso True) & vbCrLf & vbCrLf)

' create truth table for OrElse
Console.Write("OrElse" & vbCrLf & _
"False OrElse False: " & _
(False OrElse False) & vbCrLf & _
"False OrElse True: " & _
(False OrElse True) & vbCrLf & _
"True OrElse False: " & _
(True OrElse False) & vbCrLf & _
"True OrElse True: " & _
(True OrElse True) & vbCrLf & vbCrLf)

' create truth table for And
Console.Write("And" & vbCrLf & _
"False And False: " & _
(False And False) & vbCrLf & _
"False And True: " & _
(False And True) & vbCrLf & _
"True And False: " & _
(True And False) & vbCrLf & _
"True And True: " & _
(True And True) & vbCrLf & vbCrLf)

' create truth table for Or
Console.Write("Or" & vbCrLf & _
"False Or False: " & _
(False Or False) & vbCrLf & _
"False Or True: " & _
(False Or True) & vbCrLf & _
"True Or False: " & _
(True Or False) & vbCrLf & _
"True Or True: " & _
(True Or True) & vbCrLf & vbCrLf)

' create truth table for Xor
Console.Write("Xor" & vbCrLf & _
"False Xor False: " & _
(False Xor False) & vbCrLf & _
"False Xor True: " & _
(False Xor True) & vbCrLf & _
"True Xor False: " & _
(True Xor False) & vbCrLf & _
"True Xor True: " & _
(True Xor True) & vbCrLf & vbCrLf)

' create truth table for Not
Console.Write("Not" & vbCrLf & _
"Not False: " & _
(Not False) & vbCrLf & _
"Not True: " & _
(Not True) & vbCrLf) & vbCrLf)

End Sub ' Main

End Module ' modLogicalOperator

Fig. 3.14 Logical-operator truth tables. (Part 1 of 2.)
Lines 9–15 demonstrate operator **AndAlso**; lines 18–22 demonstrate operator **OrElse**. The remainder of procedure **Main** demonstrates the **And**, **Or**, **Xor** and **Not** operators. We use keywords **True** and **False** in the program to specify values of the **Boolean** data type. Notice that when a **Boolean** value is concatenated to a **String**, Visual Basic concatenates the string "**False**" or "**True**" on the basis of the **Boolean**’s value.

The chart in Fig. 3.15 displays the precedence of the Visual Basic operators introduced thus far. The operators are shown from top to bottom in decreasing order of precedence.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>parentheses</td>
</tr>
<tr>
<td>^</td>
<td>exponentiation</td>
</tr>
</tbody>
</table>

Fig. 3.14 Logical-operator truth tables. (Part 2 of 2.)
3.16 Introduction to Windows Application Programming

Today, users demand software with rich graphical user interfaces (GUIs) that allow them to click buttons, select items from menus and much more. In this chapter and the previous one, we have created console applications. However, the vast majority of Visual Basic programs used in industry are Windows applications with GUIs. For this reason, we have chosen to introduce Windows applications early in the book, although doing so exposes some concepts that cannot be explained fully until later chapters.

In Chapter 2, we introduced the concept of visual programming, which allows programmers to create GUIs without writing any program code. In this section, we combine visual programming with the conventional programming techniques introduced in this chapter and the previous chapter. Through this combination, we can enhance considerably the Windows application introduced in Chapter 2.

Before proceeding, load the project ASimpleProgram from Chapter 2 into the IDE, and change the (Name) properties of the form, label and picture box to FrmASimpleProgram, lblWelcome and picBug, respectively. The modification of these names enables us easily to identify the form and its controls in the program code. [Note: In this section, we change the file name from Form1.vb to ASimpleProgram.vb, to enhance clarity.]

Fig. 3.15 Precedence of the operators discussed so far. (Part 2 of 2.)

3.16 Introduction to Windows Application Programming

Today, users demand software with rich graphical user interfaces (GUIs) that allow them to click buttons, select items from menus and much more. In this chapter and the previous one, we have created console applications. However, the vast majority of Visual Basic programs used in industry are Windows applications with GUIs. For this reason, we have chosen to introduce Windows applications early in the book, although doing so exposes some concepts that cannot be explained fully until later chapters.

In Chapter 2, we introduced the concept of visual programming, which allows programmers to create GUIs without writing any program code. In this section, we combine visual programming with the conventional programming techniques introduced in this chapter and the previous chapter. Through this combination, we can enhance considerably the Windows application introduced in Chapter 2.

Before proceeding, load the project ASimpleProgram from Chapter 2 into the IDE, and change the (Name) properties of the form, label and picture box to FrmASimpleProgram, lblWelcome and picBug, respectively. The modification of these names enables us easily to identify the form and its controls in the program code. [Note: In this section, we change the file name from Form1.vb to ASimpleProgram.vb, to enhance clarity.]

With visual programming, the IDE generates the program code that creates the GUI. This code contains instructions for creating the form and every control on it. Unlike with a console application, a Windows application’s program code is not displayed initially in the
editor window. Once the program’s project (e.g., ASimpleProgram) is opened in the IDE, the program code can be viewed by selecting View > Code. Figure 3.16 shows the code editor displaying the program code.

Notice that no module is present. Instead, Windows applications use classes. We already have seen examples of classes, such as Console and MessageBox, which are defined within the .NET Framework Class Library (FCL). Like modules, classes are logical groupings of procedures and data that simplify program organization. Modules are discussed in detail in Chapter 4, Procedures. In-depth coverage of classes is provided in Chapter 5, Object-Based Programming.

Every Windows application consists of at least one class that Inherits from class Form (which represents a form) in the FCL’s System.Windows.Forms namespace. The keyword Class begins a class definition and is followed immediately by the class name (FrmASimpleProgram). Recall that the form’s name is set by means of the (Name) property. Keyword Inherits indicates that the class FrmASimpleProgram inherits existing pieces from another class.

The class from which FrmASimpleProgram inherits—here, System.Windows.Forms.Form—appears to the right of the Inherits keyword. In this inheritance relationship, Form is called the superclass, or base class, and FrmASimpleProgram is called the subclass, or derived class. The use of inheritance results in a FrmASimpleProgram class definition that has the attributes (data) and behaviors (methods) of class Form. We discuss the significance of the keyword Public in Chapter 5.

A key benefit of inheriting from class Form is that someone else previously has defined “what it means to be a form.” The Windows operating system expects every window (e.g., a form) to have certain capabilities (attributes and behaviors). However, because class Form already provides those capabilities, programmers do not need to “reinvent the wheel” by defining all those capabilities themselves. In fact, class Form has over 400 methods! In our programs up to this point, we have used only one method (i.e., Main), so you can imagine how much work went into creating class Form. The use of Inherits to extend from class Form enables programmers to create forms quickly and easily.

Fig. 3.16  IDE showing code for the program in Fig. 2.14.
In the editor window (Fig. 3.16), notice the text Windows Form Designer generated code, which is colored gray and has a plus box next to it. The plus box indicates that this section of code is collapsed. Although collapsed code is not visible, it is still part of the program. Code collapsing allows programmers to hide code in the editor, so that they can focus on key code segments. Notice that the entire class definition also can be collapsed, by clicking the minus box to the left of Public. In Fig. 3.16, the description to the right of the plus box indicates that the collapsed code was created by the Windows Form Designer (i.e., the part of the IDE that creates the code for the GUI). This collapsed code contains the code created by the IDE for the form and its controls, as well as code that enables the program to run. Click the plus box to view the code.

Upon initial inspection, the expanded code (Fig. 3.17) appears complex. This code is created by the IDE and normally is not edited by the programmer. However, we feel that it is important for readers to see the code that is generated by the IDE, even though much of the code is not explained until later in the book. This type of code is present in every Windows application. Allowing the IDE to create this code saves the programmer considerable development time. If the IDE did not provide the code, the programmer would have to write it, which would require a considerable amount of time. The vast majority of the code shown has not been introduced yet, so you are not expected to understand how it works. However, certain programming constructs, such as comments and control structures, should be familiar. Our explanation of this code will enable us to discuss visual programming in greater detail. As you continue to study Visual Basic, especially in Chapters 5–10, the purpose of this code will become clearer.
When we created this application in Chapter 2, we used the Properties window to set properties for the form, label and picture box. Once a property was set, the form or control was updated immediately. Forms and controls contain a set of default properties, which are displayed initially in the Properties window when a form or control is created. These default properties provide the initial characteristics of a form or control when it is created. When a control, such as a label, is placed on the form, the IDE adds code to the class (e.g., FrmASimpleProgram) that creates the control and that sets some of the control’s property values, such as the name of the control and its location on the form. Figure 3.18 shows a portion of the code generated by the IDE for setting the label’s (i.e., lblWelcome’s) properties, including the label’s Font, Location, Name, Text and TextAlign properties. Recall from Chapter 2 that we explicitly set values for the label’s Text and TextAlign properties. Other properties, such as Location, are set only when the label is placed on the form.

The values assigned to the properties are based on the values in the Properties window. We now demonstrate how the IDE updates the Windows Form Designer–generated code created when a property value in the Properties window changes. During this process, we must switch between code view and design view. To switch views, select the corresponding tabs: ASimpleProgram.vb for code view and ASimpleProgram.vb [Design] for design view. Alternatively, you can select View > Code or View > Designer. Perform the following steps:

1. **Modify the file name.** First, change the name of the file from Form1.vb to ASimpleProgram.vb by clicking the file name in the Solution Explorer and changing the FileName property.
2. Modify the label control’s **Text** property, using the **Properties** window. Recall that properties can be changed in design view by clicking a form or control to select it and modifying the appropriate property in the **Properties** window. Change the **Text** property of the label to “*Deitel and Associates*” (Fig. 3.19).

3. Examine the changes in code view. Switch to code view, and examine the code. Notice that the label’s **Text** property is now assigned the text that we entered in the **Properties** window (Fig. 3.20). When a property is changed in design mode, the Windows Form Designer updates the appropriate line of code in the class to reflect the new value.

---

**Fig. 3.19** Properties window as used to set a property value.

**Fig. 3.20** Windows Form Designer-generated code reflecting new property values.
4. Modify a property value in code view. In the code-view editor, locate the three lines of comments indicating the initialization for `lblWelcome`, and change the `String` assigned to `Me.lblWelcome.Text` from “Deitel and Associates” to “Visual Basic .NET” (Fig. 3.21). Then switch to design mode. The label now displays the updated text, and the Properties window for `lblWelcome` displays the new `Text` value (Fig. 3.22). [Note: Property values should not be set using the techniques presented in this step. Here, we modify the property value in the IDE-generated code only as a demonstration of the relationship between program code and the Windows Form Designer.]

5. Change the label’s `Text` property at runtime. In the previous steps, we set properties at design time. Often, however, it is necessary to modify a property while a program is running. For example, to display the result of a calculation, a label’s text can be assigned a `String` containing the result. In console applications, such code is located in `Main`. In Windows applications, we must create a method that executes when the form is loaded into memory during program execution. Like `Main`, this method is invoked when the program is run. Double-clicking the form in design view adds a method named `FrmASimpleProgram_Load` to the class (Fig. 3.23). Notice that `FrmASimpleProgram_Load` is not part of the Windows Form Designer-generated code. Add the statement `lblWelcome.Text = "Visual Basic"` into the body of the method definition (Fig. 3.24). In Visual Basic, properties are accessed by placing the property name (i.e., `Text` in this case) after the class name (i.e., `lblWelcome` in this case), separated by the dot operator. This syntax is similar to that used when accessing class methods. Notice that the IntelliSense feature displays the `Text` property in the member list after the class name and dot operator have been typed (Fig. 3.23). In Chapter 5, Object-Based Programming, we discuss how programmers can create their own properties.
6. **Examine the results of the** `FrmASimpleProgram_Load` **method.** Notice that the text in the label looks the same in **Design** mode as it did in Fig. 3.22. Note also that the **Property** window still displays the value “**Visual Basic .NET**” as the label’s **Text** property. The IDE-generated code has not changed either. Select **Build > Build Solution** and **Debug > Start** to run the program. Once the form is displayed, the text in the label reflects the property assignment in `FrmASimpleProgram_Load` (Fig. 3.25).
7. Terminate program execution. Click the close button to terminate program execution. Once again, notice that both the label and the label’s Text property contain the text Visual Basic .NET. The IDE-generated code also contains the text Visual Basic .NET, which is assigned to the label’s Text property.

This chapter discussed how to compose programs from control structures that contain actions and decisions. In Chapter 4, Procedures and Arrays, we introduce another program-structuring unit, called the procedure. We will discuss how to compose programs by combining procedures that are composed of control structures. We also discuss how procedures promote software reusability. In Chapter 5, Object-Based Programming, we discuss in
more detail another Visual Basic program-structuring unit, called the class. We then create objects from classes and proceed with our treatment of object-oriented programming—a key focus of this book.

3.17 Summary

The If/Then single-selection structure selects or ignores a single action (or a single group of actions), based on the truth or falsity of a condition. The If/Then/Else double-selection structure selects between two different actions (or groups of actions), based on the truth or falsity of a condition.

The While and Do While/Loop repetition structures allow the programmer to specify that an action is to be repeated while a specific condition remains true. Eventually, the condition in a While, Do While/Loop or Do/Loop While structure becomes false. At this point, the repetition terminates, and the first statement after the repetition structure executes.

The Do Until/Loop and Do/Loop Until repetition structures allow the programmer to specify that an action is to be repeated while a specific condition remains false. Eventually, the condition in a Do Until/Loop or Do/Loop Until structure becomes true. At this point, the repetition terminates, and the first statement after the repetition structure executes. The For/Next repetition structure handles the details of counter-controlled repetition. The required To keyword specifies the initial value and the final value of the control variable. The optional Step keyword specifies the increment. The Exit Do, Exit While and Exit For statements alter the flow of control by causing immediate exit from a repetition structure.

Visual Basic provides the Select Case multiple-selection structure so that a variable or expression may be tested separately for each value that the variable or expression might assume. The Select Case structure consists of a series of Case labels and an optional Case Else.

The logical operators are AndAlso (logical AND with short-circuit evaluation), And (logical AND without short-circuit evaluation), OrElse (logical inclusive OR with short-circuit evaluation), Or (logical inclusive OR without short-circuit evaluation), Xor (logical exclusive OR) and Not (logical NOT, also called logical negation).

With visual programming, the IDE actually generates program code that creates the GUI. This code contains instructions for creating the form and every control on it. Windows application code is contained in a class. Like modules, classes are logical groupings of procedures and data that simplify program organization.

Forms and controls contain a set of default properties, which are displayed initially in the Properties window when a form or control is selected. These default properties provide the initial characteristics that a form or control has when it is created. When a change is made in design mode, such as when a property value is changed, the Windows Form Designer creates code that implements the change. Often, it is necessary to modify a property while a program is running. In Windows applications, the code that implements the change is placed in a procedure that executes when the form is loaded, which can be created by double-clicking the form in design view.