Visual Basic 2008 Programming

Business Applications with a Design Perspective

Jeffrey J. Tsay

Table of Contents

Chapter 7

Repetition

One of the most powerful features of the computer is its capability to execute a group of instructions repetitively while the data handled by these instructions change. With this capability, many tasks that are repetitive in nature can be handled with much shorter code and greater flexibility. Imagine a payroll routine that computes for each employee the gross pay, various deductions, and the net pay. The steps to handle each employee's pay are basically the same. Only the data—the employee and the pay—in question change. Without the capability to execute the group of instructions repetitively, the same code will have to be repeated for each employee. The length of code will be sizable, to say the least. Worse yet, the number of repetitions in code, which vary with the number of employees to be paid, will have to be known beforehand. Each time the number of employees changes, the program will have to be modified—an extremely tedious task.

Many of the problems (much more than can be imagined) handled by the computer are repetitive in nature. To list a few, populating a list box or combo box, reversing a string, computing the square root of a number (searching for the root of any function in general), and listing the content of a file all require repeating a selected group of instructions.

In VB, there are several structures by which you can construct code for repetition. In each of these structures, the repetition starts with a keyword, such as Do or For, in an opening statement and ends with another keyword, such as Loop or Next, in the closing statement. Statements enclosed physically inside the two statements are repeated until a certain condition is satisfied. Because the execution of the group of statements starts from the opening statement top down, ends at the closing statement, and starts over again from the opening statement, the repetition looks like a loop and is commonly so referenced. This chapter provides detailed discussions of these structures.

After completing this chapter, you should be able to:

- Explain all variations of the Do...Loop structure,
- Develop code that calls for the use of the Do...Loop structure with all variations,
- Explain the For...Next structure with different values and signs of the increment parameter,
- Develop code that requires the use of the For...Next structure, and
- Develop code that calls for the use of nested loop structures.

7.1 The Do...Loop Structure

One of the structures for repetition is the *Do…Loop structure*. You can construct the Do...Loop structure with either of the following syntax:

```
    Do [{While|Until} condition]
        Statements to be repeated
    Loop
```

or

```
    Do
        Statements to be repeated
    Loop [{While|Until} condition]
```

where *condition* = any expression that can be evaluated to True or False; any expression that can be used in an If statement can be used here.
While = the keyword that will make the loop continue as long as the condition is true, and
Until = the keyword that will make the loop continue as long as the condition is false.
When the loop ends, the execution continues to the line below the Loop statement. The following examples illustrate how the various variations of the Do...Loop structure work.

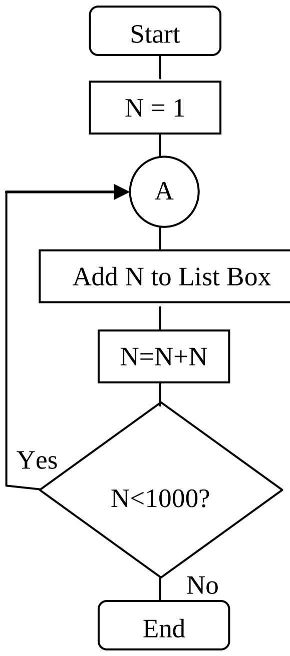
Example 1. Showing a Sequence of Numbers

Suppose you would like to know the sequence of numbers in the range of 1 and 1,000, which are successively doubled beginning with 1. To solve the problem, you can do the following:

1. Create a variable such as N, and assign it with a value 1.
2. Show N in a list box.
3. Double N.
4. Compare N with 1,000. If N is less than 1,000, repeat steps 2 and 3; otherwise, end the process.

A flowchart that depicts the process is presented in Figure 7-1.

Figure 7-1
A flowchart to display a sequence of numbers



The flowchart gives a visual hint that the connector (A) and the decision (N <1,000) form a loop. The following routine routine carries out this process:

```
Private Sub Form1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
  Handles MyBase.Click
    Dim N As Integer
    N = 1
    Do
        lstNumbers.Items.Add(N)
        N = N + N
    Loop While N < 1000
End Sub
```

Before the loop is executed, N is set to 1. Within the loop, the value N is added to the lstNumbers list box, using the control's Items.Add method. N is then added to itself, doubling its value in effect. When the loop reaches the end (the Loop statement), N is compared with 1,000. The While condition states that if N is less than 1,000, the repetition continues. The execution transfers back to the beginning of the loop (the Do statement). The two statements inside the loop will be executed again. This cycle continues until N is greater than or equal to 1,000; then the While condition is no longer true. The execution control goes to the statement immediately below the Loop statement. In this case, the End Sub statement is reached, and the Form Click event procedure ends.

The process can be depicted with the following execution table.

Iteration No.	N (in lstNumbers)	N (After doubling)
1	1	2
2	2	4
3	4	8
4	8	16
.	.	.
10	512	1024

Notice that the Loop While statement can be rewritten with the Until keyword as follows:

```
    Loop Until N >= 1000
```

Replacing this statement for the preceding Loop While line will have exactly the same result as the original routine. Try both ways. The last number you should see in the list box is 512.

Example 2. Reading a Name List and Populating a List Box

Suppose you have created a list of names with one name per line, and kept it in a file located at c:\friends\namelist.txt. The content of the file appears as follows:

```
Andrea Aaron
Ben Bennett
.
.
Zeff Zenor
```

You would like to populate the lstNames list box with these names as soon as your program starts. To do this, you must first open the file (associate the file with a StreamReader); then your program should do the following:

Read a name from the file.

Add the name to the list box using the Items.Add method.

Repeat steps 1 and 2 until the file runs out of names.

Assume the StreamReader associated with the file is NameFileReader, and the list box name is lstNames. The first approximation of the repetition should appear as follows:

```
    Do
        TheName = NameFileReader.ReadLine() 'Read a name and assign it to TheName
        lstNames.Items.Add(TheName) 'Add the name to the list box
    Loop Until NameFileReader.Peek() = -1 'terminate the loop if it's end of file
```

In this structure, when the loop is executed, the computer will read the name from the file and then add it to the list box before the execution reaches the Loop statement, where the EOF condition is tested. Recall that the StreamReader's Peek method returns the next character in the file without moving the current position, and it returns a value –1 when the EOF is reached. Because the Loop statement uses the keyword Until, the loop will continue when the Peek method does not return –1; that is, EOF is not reached.

Dealing with an Empty File

As long as there is at least one name in the file, the preceding code structure will perform properly without any error; however, if the file happens to be empty, this structure will have a problem. When it attempts to read the name the first time, the EOF is reached and an error occurs. The program will fail. To guard against this possibility, the EOF condition should be tested at the beginning of the loop; that is, the condition should be placed in the Do statement. If the file is empty, no statements inside of the loop will be executed. The loop should appear as follows:

```
    Do Until NameFileReader.Peek() = -1 ‘Check for EOF
        ‘ Read a name from NameFile
        TheName = NameFileReader.ReadLine()
        ‘ Add the name to the list box
        lstNames.Items.Add(TheName)
    Loop
```

The Complete Procedure

Because you want to populate the list box as soon as the program starts, you should place the code in the Form Load event procedure as shown here:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
  Handles MyBase.Load
    Dim TheName As String
    Dim NameFileReader As New System.IO.StreamReader("C:\Friends\NameList.txt")
    Do Until NameFileReader.Peek() = -1 'Check for EOF
        ' Read a name from NameFile
        TheName = NameFileReader.ReadLine()
        ' Add the name to the list box
        lstNames.Items.Add(TheName)
    Loop
    NameFileReader.Close()
End Sub
```

In the procedure, the Dim statement declares the variable NameFileReader as a StreamReader. The New keyword creates a StreamReader object that is associated with the NameList.txt file. After the file is associated with the StreamReader, the Do...Loop structure will perform the required repetition until it exhausts the records in the file.

Difference in Effect Between the Two Structures

The preceding discussion suggests that whether the statements inside the loop will be executed at least once depends on the placement of the condition in the Do or Loop statement. Placing the condition with the Do statement can prevent those statements from being executed; placing the condition with the Loop statement ensures that those statements will be executed at least once.

If the execution of those statements strictly depends on the condition, you should place the condition in the Do statement; however, if those statements must be executed at least once or if it is inconsequential whether the condition is checked before entering the loop, you may place the condition in the Loop statement. For example, the code given in Example 1 can be changed to the following without affecting the result or exposing any potential possibility for errors:

```
Private Sub Form1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
  Handles MyBase.Click
    Dim N As Integer
    N = 1
    Do While N < 1000
        lstNumbers.Items.Add(N)
        N = N + N
    Loop
End Sub
```

Why? You know for certain that the first time the loop is executed the value of N will be 1 and thus is less than 1,000. When the loop is executed enough times such that N is at least 1,000, the loop will end regardless of where the checking is performed with Do or with Loop.

The While… End While Loop

As a side note, the Do While…Loop structure has a cousin in syntax: *While…End While*. With this syntax structure, you can code the loop as follows:

```
    While N < 1000
        lstNumbers.Items.Add(N)
        N = N + N
    End While
```

This code will produce exactly the same result. (Note that there is no Until… End Until Structure.)

The Endless Loop

Notice that in the previous structures, some of the statements inside the loop must produce some result that causes the condition to be False in conjunction with the keyword While, or True in conjunction with the keyword Until; otherwise, the loop will execute endlessly, resulting in an undesirable trap. Thoroughly test your program to prevent this situation from happening.

Do Loop Without a Condition on Either Statement

If you inspect the syntax of the Do...Loop structure again, you may discover that the While/Until condition is optional; that is, the entire structure can do without any While/Until condition. For example, it is perfectly legitimate to have a Do...Loop structure as follows:

```
    Do
        'Statements to be executed
    Loop
```

Similar to the preceding structures, there must be some condition(s)—plus some statements—inside the loop that can terminate the loop; otherwise, the loop will repeat endlessly. Typically, this structure is used when the condition to be checked is neither at the beginning nor at the end of the group of statements to be executed. The following example illustrates this situation.

Example 3. Computing the Value of an Infinite Series

Consider the following series:

```
    S = 1/2 + (1/2)² + (1/2)³ + ....
```

Is there a way to approximate the value of the series numerically? The series starts with a value of 1/2. Each subsequent value to be added is half of the previous value. Although the addition is supposed to be carried out infinitely, your intuition is that at a certain point, the number in the series will be so small that any further addition will not alter the tangible value of the result. You can certainly stop the addition, or get out of the loop, at that time. For practical purposes, you can consider 0.1E-07 (10^{-8}) very small.

Let *S* be the variable to accumulate the value for the series, and *V* be the variable to represent the value at each point of the series. The algorithm to compute the series can be described as follows:

1. Set V to 1, and S to 0.
2. Divide V by 2; that is, V = V/2.
3. Compare V with the very small number. If V is less than that number, terminate the computation; otherwise, proceed to step 4.
4. Add V to S; that is, S = S + V.
5. Go to step 2 and repeat the process.

With this algorithm, the following code can be written to approximate the result:

```
    Dim S As Single
    Dim V As Single
    V = 1
    S = 0
    Do
        V = V / 2
        If V < .1E-07 Then Exit Do
        S = S + V
    Loop
    MsgBox("The value of the series is " & S)
```

The following execution table shows the value of S and V at various stages.

Iteration No.	Value of V	Value of S
0	1	0
1	1/2	½
2	$(1/2)^2$	$(1/2) + (1/2)^2$
3	$(1/2)^3$	$(1/2) + (1/2)^2 + (1/2)^3$
.	.	.
.	.	.
.	.	.
N	$(1/2)^n$	$(1/2) + (1/2)^2 + \ldots + (1/2)^n$

The Exit Do Statement

Notice how the loop is ended in the previous routine. You place an If statement among the group of statements inside the loop. Each time the statement is reached (after V has been divided by 2), it checks whether V is very small (when V < 0.1E-07). If the value is not very small, execution will drop down to the next statement and V will be added to S. When V is very small, the *Exit Do statement* will be executed, terminating the loop. When this statement is executed, the execution control is transferred to the statement immediately following the Loop statement; that is, the MsgBox statement, exactly like the situation in which the condition with the Until/While keyword is True.

Place the preceding code in an event procedure and then test the result. (*Note:* VB will automatically convert the number 0.1E-07 to 0.00000001.) It should print 1 as the result. Are you wondering whether the computation of series has any practical application in business? Yes, there are problems of this nature in business. Exercise 7-28 provides an example.

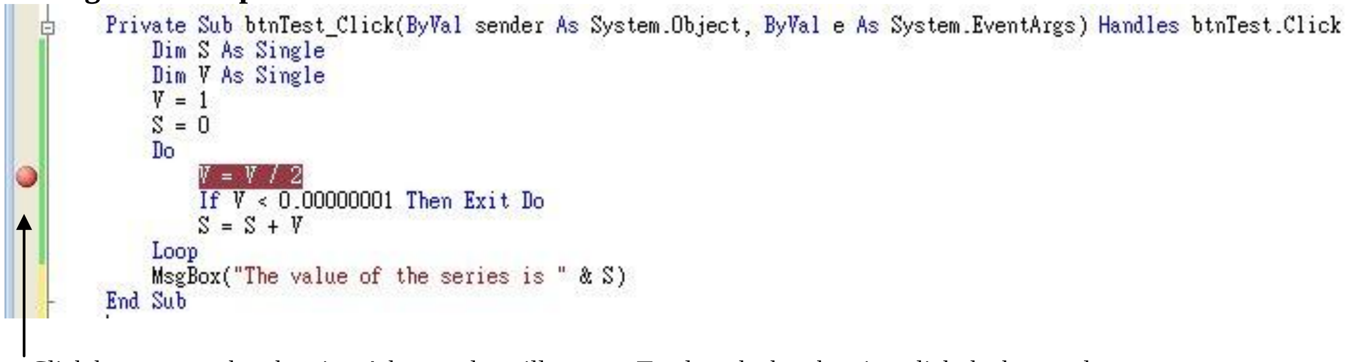
Tip

The statement to terminate the While… End While loop from inside is the *Exit While* statement.

Viewing Data in the Loop

In many cases in the process of testing your program, you need to inspect the values of the variables in the loop. Their values at different stages can provide clues as to how the program executes as well as to why a program fails to produces the expected results. In the preceding example, you may wonder how the values of V and S change at different stages of execution. One straightforward way is, of course, just to insert statements that display the values. This approach, however, has some drawbacks. You have to not only take time to insert these statements initially, but also take care to remove them afterwards.
A more convenient alternative is to set the *breakpoint* using the IDE's debugging facility. At design time, while in the code window, locate the statement of interest and then click on the left margin of the code window. You will see the statement highlighted in brown color as shown in 7-2. At run time, the execution breaks at this statement, waiting for your action. You can then rest the mouse pointer on any variable of interest. The value of the variable will be displayed. To have the execution move forward step-by-step, press the F8 key. Each time you press F8, the execution moves forward by one step. (Again, you can inspect the value of any variable at any time.) You can cancel the break point by clicking the circle of the highlighted line to clear the breakpoint. To resume normal execution, click the Start button in the toolbar again.

Figure 7-2
Setting the breakpoint



```
Private Sub btnTest_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnTest.Click
    Dim S As Single
    Dim V As Single
    V = 1
    S = 0
    Do
        V = V / 2
        If V < 0.00000001 Then Exit Do
        S = S + V
    Loop
    MsgBox("The value of the series is " & S)
End Sub
```

Click here to set a break point. A brown dot will appear. To clear the break point, click the brown dot.

You can actually set as many breakpoints as you want. Whenever the execution hits a breakpoint, it breaks. If you have set many break points, you can use the Delete All Breakpoints option in the Debug menu to clear all breakpoints at one time. Of course, the use of breakpoints is not limited to checking values in loops. You can use it in any part of your program where you have a need to examine the variables.

Tip

Set the breakpoint on the statement from which you want to start inspecting the values of variables. Clear it after it is no longer needed. This approach is much simpler than inserting statements to display the values of the variables.

Example 4. Finding the Solution to an Equation Numerically

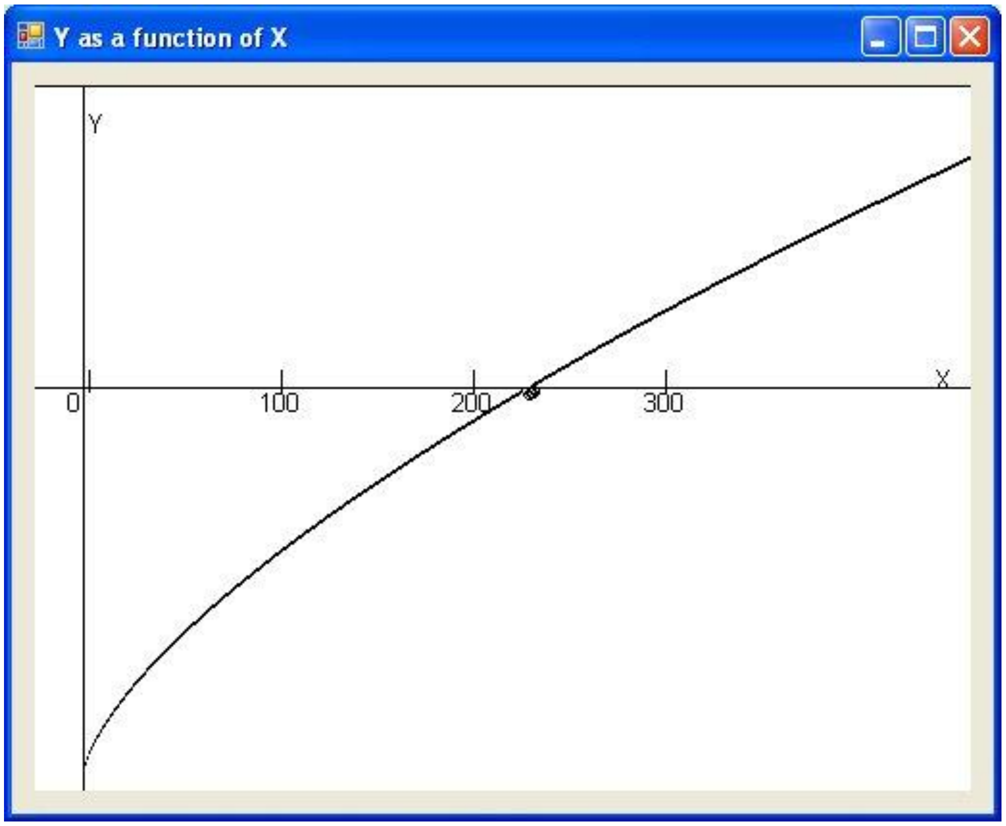
You have learned how to perform computations repetitively using the Do...Loop structure. Repetition is a powerful feature that you can use to solve various mathematical problems. To illustrate, consider the following equation:

$a\ x^c = 4{,}000$
where $a = 100$
$c = 1 + \log(.8)/\log(2)$
You are to write a VB program to find the solution (the value of x that makes the equation true). One possible way, of course, is to perform some analysis or algebraic manipulation so that you can express x in terms of all the other values. You can then just code the formula into your program. In this case, it should work.

Sometimes the equation is so complex, however, that it defies any analysis. It will then be impossible to solve the problem *analytically*. In such a case, you may still be able to use the computer to find the solution *numerically*. Pretend that this problem is complex enough that you would rather solve it numerically.

Where do you start? First, examine the equation again. It can be rewritten as $y = a\ x^c - 4000$. You can then plot the function as shown in Figure 7-3.

Figure 7-3
The function



Notice that the x value at which the y value is 0 is the solution. As you can see, the function increases monotonically with respect to x. In such a case, there is a fairly simple but efficient way to find the solution.

An Algorithm for the Numerical Solution

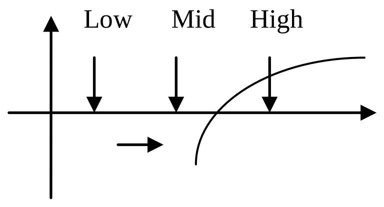
Inspect the chart again. Notice that for any x value beyond the solution, y is greater than 0 and below the solution y is less than 0. Now, suppose you know that x must fall in a certain range; you can do the following:

- Compute the midpoint of the range of x.
- Find the new y for the midpoint of x.
- Check the resulting y at this midpoint:
    - If y is less 0, this x value is too small. You can change the lower bound of the range to this midpoint.
    - If y is greater than 0, this x value is too large. You can change the upper bound of the range to this midpoint.

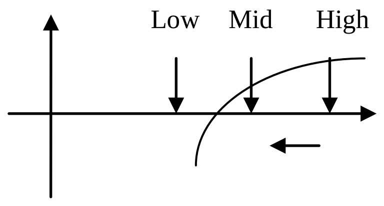
In either case, you have narrowed the possible range of the solution by half of the current range. The revised range can again be used to compute a new midpoint, which will give a clue to revising the range of the solution.

When this trial-and-error method is repeated a sufficient number of times, you will find the range is so small that any value within it can be considered a good approximation to the solution. This method of finding a solution of an equation numerically is recognized as the *half interval method*. Figure 7-4 illustrates graphically how this trial-and-error method is carried out.

Figure 7-4
The half interval method at work



If the midpoint of the range has a negative value of y, then it is on the left side of the solution. Adjust the range by replacing Low with the Mid value. (Move Low point to Mid point.)



If the midpoint of the range has a positive value of y, then it is on the right side of the solution. Adjust the range by replacing High with the Mid value. (Move High point to Mid point.)

Searching for the Initial Range

How do you set the initial range to search for the solution? In this example case, it is obvious that the lower bound cannot be anything less than 0. The upper bound can be identified by a reverse half interval method. Suppose you try a value for x at 1. You should find the function gives a y value that is less than 0. So, you can double this value of x and try again. If it is still smaller than 0, you can double x again. You can repeat this process until you obtain an x value that results in a y value greater than 0. This x value can then be used as the upper bound.

In sum, the procedure to find the numerical solution can be stated by the pseudo-code as follows:

```
1. Find an appropriate upper bound:
    a. Let x = 1.
```

    b. Double x.
    c. Compute y for the given x.
    d. If y is less than 0, repeat 1.b and 1.c; otherwise, proceed to step 2.
2. Find the numerical approximation (solution) using the half interval method:
    a. Let Low = 0; and High = x, which is obtained in step 1. (Low and High represent the lower and upper bounds of the range to search for the solution.)
    b. Compute x = (Low + High)/2.
    c. Compute y for the given x.
    d. If y is less than 0, let Low = x (that is, adjust the lower bound); otherwise, let High = x (that is, adjust the upper bound).
    e. If (High – Low) is still not small enough, repeat steps 2.b through 2.d; otherwise, terminate the search. x is a good approximation to the solution.

Suppose the user will click a button named btnSolve to initiate the computation. The complete procedure using this algorithm can be coded in VB as follows:

```
Private Sub btnSolve_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnSolve.Click
    Dim R As Double
    Dim C As Double
    Dim A As Double
    Dim T as Dboule
    Dim Y As Double
    Dim Low As Double
    Dim High As Double
    Dim X As Double
    R = 0.8
    A = 100
    T = 4000
    C = 1 + Math.Log(R) / Math.Log(2)
    ' Search for a point that Y(X) > T
    X = 1
    Do
        X = X + X
        Y = A * X ^ C - T
    Loop Until Y >= 0
    ' Search for X using the half interval method
    Low = 0
    High = X
    Do
        X = (High + Low) / 2
        Y = A * X ^ C - T
        If Y < 0 Then
            ' Too low; adjust lower bound
            Low = X
        Else
            ' Too high; adjust upper bound
            High = X
        End If
    Loop Until (High - Low) / X < 0.000001
    MsgBox("X = " & X)
End Sub
```

Notice that you have assigned 0.8 to a variable R, 100 to A, and 4000 to T. You can redesign the project so that the values of these variables can be obtained from data entered through text boxes. The modification is left to you as an exercise.

The Termination Criterion

Notice that you used the formula (High – Low)/X to determine whether the range of the true solution is small enough. The criterion is based on a precision *relative* to the value of the solution. In contrast, you can also use *absolute* precision as the criterion by comparing the difference between High and Low with a very small value without dividing by X. In general, the relative precision is better, especially when X is relatively large. When X is near zero, however, you may encounter an overflow problem with the computation of the relative precision.

The Real-World Meaning of the Example

Are you aware that you have just solved a learning curve problem? The problem asks, "If the first product unit takes 100 hours (A) to complete, how many units can we produce if we have 4,000 hours (T), assuming our factory has a 0.8 learning rate (R)?" Often used in the aircraft manufacturing industry, the learning curve model asserts that because of the task complexity, the initial time (A) to complete a unit of product takes longer. As the experience (production) doubles, the average time required to perform the same task decreases by a constant percentage, R. The formula you see at the beginning of this example is the one to compute the total hours required to produce X units. But it is used here to find the units of production; given 4,000 available hours (T=4,000), a learning rate of 80% (A=.8) and the first unit takes 100 hours to produce (A=100). The solution to the problem is approximately 230.5 units. This is intriguing because if every unit takes 100 hours to produce, you can produce only 40 units with 4,000 hours available!

As a side note, perhaps you have also learned a moral. When you first attempted to solve this problem, you were unaware of the problem's practical meaning. As long as you know how the function behaves, your knowledge (or the lack) of the practical meaning of the problem should not affect your ability to solve the programming problem.

Tip

You may encounter a situation in which the code contains a loop that will continue until the user clicks a button (say, btnStop). Apparently, the loop cannot include the click event procedure that is triggered when the button is clicked. How can you then terminate the loop when the event is fired? Here is the solution. Declare a class level Boolean variable (say, GoodBye). Set the variable to True in the button's click event procedure; that is, to code:

```
Dim GoodBye As Boolean
Private Sub btnStop_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnStop.Click
    GoodBye = True
End Sub
```

Assume the Do…Loop has no other termination conditions in the opening or closing statement. The loop can be structured as follows:

```
    Do Until GoodBye
       ' Place all other statements for the loop here.
        Application.DoEvents() 'Release execution control to the system
    Loop
```

Notice that there is an Application.DoEvents statement inside the loop. The statement releases execution control to the system for it to perform system level activities, including checking for the Button Click event so that the Boolean variable GoodBye can be set properly. Without the DoEvents statement, the loop will never terminate.

7.2 The For...Next Structure

The preceding section focused on the Do…Loop structure, and briefly mentioned the While…End While structure. Both of these structures perform the repetition based on the value of a condition (True or False). This section discusses the For…Next structure that performs repetition based on a counter. The For...Next structure has the following syntax:

```
    For Counter = Starting Value To Ending Value [Step Increment]
        Statements to be executed
    Next [Counter]
```

where *Counter* = a variable to serve as the counter in the repetition; it starts with the *starting value* and is added by the *increment* in each iteration.
Starting Value = an expression that can be evaluated to a numeric value, which will be assigned to *Counter* when the loop starts.
Ending Value = an expression that can be evaluated to a numeric value, which will be compared with *Counter*.
Increment = an optional parameter that will be added to *Counter* for each iteration. *Increment* can be either a positive or a negative value; if this parameter is omitted, 1 is the default value.
As the syntax shows, the For...Next structure starts a loop with the For statement and ends with the Next statement. The counter in the Next statement is optional.

When the loop starts, the counter is set to the starting value for the first iteration. The increment is then added to the counter for each of the subsequent iterations. At the beginning of each iteration, the counter value for the upcoming iteration is first compared with the ending value. Either the statements inside the loop will be executed, or the loop will terminate, depending on the sign of the increment and the result of comparison.

A Closer Look at the Increment

Notice that the increment can be either positive or negative. If the increment is positive, the loop will end as soon as the counter is greater than the ending value. If the starting value is greater than the ending value, the loop will terminate immediately and the statements inside the loop will never be executed. On the other hand, if the increment is negative, the loop will end as soon as the counter is less than the ending value. In this case, if the starting value is less than the ending value, statements inside the loop will never be executed. The two flowchart fragments in Figure 7-5 show how the For...Next loop works with positive and negative increment.

Notice that when the increment is negative, adding it to the counter will decrease the value of the counter. The following examples illustrate how the For...Next loop can be used.

Example 5. Listing a Sequence of Numbers

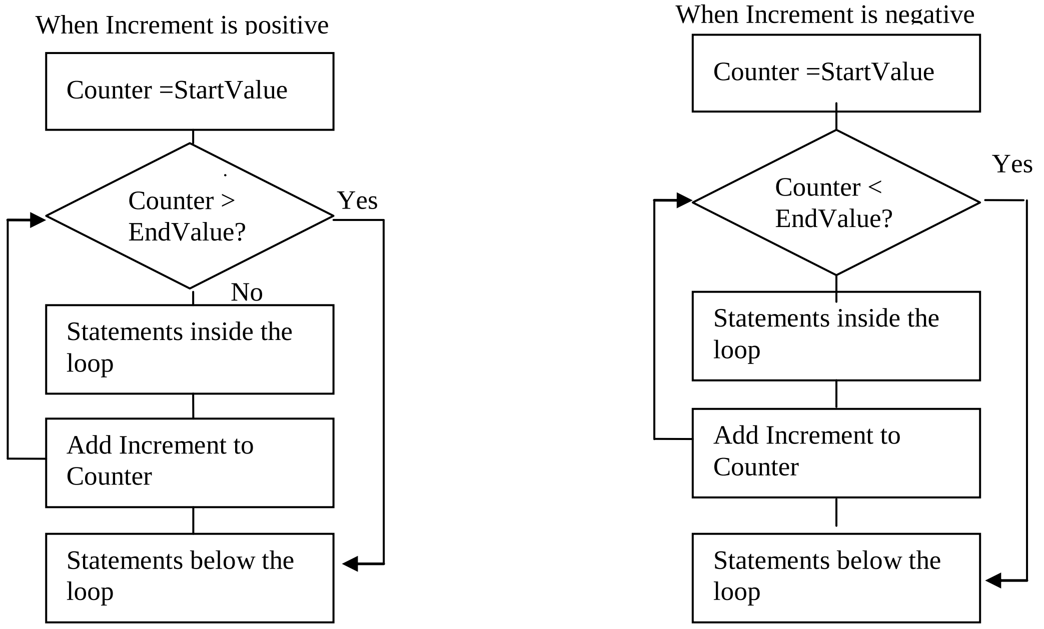
Suppose you would like to show by code the numbers 1 through 10 in the list box. What would be the easiest to do? You can use the For...Next loop, setting the starting value to one (1) and ending value to 10 with an increment of 1. You can then use the Items.Add method to add the value of the counter inside the loop. The code should appear as follows:

```
    Dim Counter As Integer
    lstNumbers.Items.Clear 'Clear the current content to prevent multiple lists
    For Counter = 1 To 10
        lstNumbers.Items.Add(Counter)
    Next
    ' execution will continue here after the above loop is done
```

Recall that in the For statement, if the Step optional parameter is omitted, the increment defaults to 1. In the preceding code, when the loop starts, the starting value 1 is assigned to the variable Counter. The list box's Items.Add method inside the loop is then executed, adding the current value of Counter into the list box.

In the next iteration, the default increment 1 is added to Counter, resulting in 2, which is compared with the ending value 10. Because 2 is not greater than 10, the statement inside the loop is executed. The iteration continues until Counter has a value of 11, which is greater than the ending value 10. The loop is then terminated. The execution control is transferred to the statement immediately below the Next statement. (Try the routine in a form Click event. Click the form a few times, and inspect the results.)

Figure 7-5
For loops with positive and negative increment


When Increment is positive
Counter =StartValue
Counter > EndValue?
Yes
No
Statements inside the loop
Add Increment to Counter
Statements below the loop
When Increment is negative
Counter =StartValue
Counter < EndValue?
Yes
Statements inside the loop
Add Increment to Counter
Statements below the loop

Example 6. Listing Who's Invited (the Invitation Project)

In Example 2, you saw how the lstNames list box was populated with the list of your friends' names in a file. Suppose the list box's SelectionMode property has been set to MultiSimple. You can select as many names as you want. When you click the btnInvite button with a text, Invite, you would like the computer to display in a message box the list of friends whom you plan to invite to your birthday party. How can you do that? (Eventually, you want to modify the program so that the computer will print a personal invitation to each of the friends invited.)

Recall that the list box has a SelectedItems property, and the property is indexed. SelectedItems(0) gives the first selected item, and SelectedItems(1) gives the second. Notice that the index of the property in the pair of parentheses does not have to be a constant. It can be a variable. If you use a variable for the index and have a way to change this variable such that it will go 0, 1, 2, and so on until the end of the list, you would be able to list all the items in the SelectedItems property.

If you set up a For loop with a beginning number 0 and an increment 1, the loop's counter will go 0, 1, 2 and so on, exactly the same values you would like for the index of the SelectedItems property to be. The Counter of the loop can then be used as the index. What is the value of the index for the last item in the list box? Recall that the SelectedItems.Count property gives the number of items selected. Because the index of SelectedItems starts with 0, the last item should have an index of SelectedItems.Count –1; for example, if three items are selected, giving a Count of 3, they will be indexed 0, 1, and 2. The code to show the list of friends to be invited should appear as follows:

```
Private Sub btnInvite_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnInvite.Click
    Dim MyMsg As String
    Dim I As Integer
    MyMsg = "I'll invite the following friends to my birthday party:"
    For I = 0 To lstNames.SelectedItems.Count - 1
        MyMsg = MyMsg & vbCrLf & lstNames.SelectedItems(I)
    Next
    MsgBox(MyMsg)
End Sub
```

In the procedure, the string "I'll invite…party:" is placed before the loop starts because you want the line to appear in the message only once, and before any of your friends' names. The For statement sets the starting value for the counter, I to 0 and the ending value to the number of selected items (SelectedItems.Count) minus 1. Inside the loop, the existing value of MyMsg is concatenated with the carriage return—vbCrLf, the end of line marker—and with the name corresponding to the index that has the value of the current Counter. The carriage return makes the subsequent string containing the name appear on the next line. The following execution table depicts the process of executing the For loop.

Iteration	I	SelectedItem(I)	Resulting MyMsg
Before loop			I'll invite the following friends to my birthday party:
1	0	Andrea Aaron	I'll invite the following friends to my birthday party: Andrea Aaron
2	1	George Gunther	I'll invite the following friends to my birthday party: Andrea Aaron George Gunther
3	2	Lisa Latimer	I'll invite the following friends to my birthday party: Andrea Aaron George Gunther Lisa Latimer

Example 7. Every Other Day

You have made a decision to run every other day in the month of October beginning on October 1. You would like to show in a list box all the days you should run as a reminder. How can you do this? You can still use the counter in a loop to represent the days that you are supposed to run. The only difference in this problem from Example 5 is that the increment is no longer 1 but 2; therefore, the routine can appear as follows:

```
    Dim Counter As Integer
    lstNumbers.Items.Clear
    lstNumbers.Items.Add("I am determined to run on the following days of
      October:")
    For Counter = 1 to 31 Step 2
        lstNumbers.Items.Add(Counter)
    Next
```

In this example, the Step parameter is specified explicitly as 2; therefore, Counter will begin with the starting value of 1. After the first iteration, it is increased by 2, resulting in 3 being added to the list box. The following numbers should appear in the list box below the text string:

```
1 3  5  7  9  11  13  15  17  19  21  23  25  27  29  31
```

Tip

In some languages, there is a restriction that the counter as well as the increment for the For loop must be Integer (or Long) variables. There is no such restriction in VB; however, it is advisable that both the counter and the increment are of the same data type. This will eliminate the need for data conversion and avoid unexpected problems, some of which are illustrated in the Explore and Discover exercises at the end of this chapter.

Example 8. Displaying a String on Two Lines (the ShowTwoLines Project)

Suppose you have a text string that is longer than 80 characters, but shorter than 140. You would like to display it in a message box in two lines, and you don't want the first line to be longer than 80 characters. How can this be done?
Assume your text string has embedded spaces between positions 1 and 80. One way to handle this is to search the string backward for a blank space starting from the 80th position. This space position can then be used as the split point; everything on the left can be printed on the first line and the remainder on the second.

To search for the space backward, you can compare each character in the string with the space character beginning with the 80th position, and go backward until the comparison is true. Recall that you can use the Mid function to extract a character from a string. The code to search for a blank space for this purpose should appear as follows:

```
    Dim P As Integer
    For P = 80 to 1 Step -1
        If Mid(Text, P, 1) = " " Then
            Exit For
         End If
    Next
```

In the preceding code, P is used as the counter with a starting value of 80. Inside the loop, the Mid function extracts a character at position P. This character is then compared with the space character. If the comparison is not True, nothing is done within the If block. The Next P statement transfers execution control to the For statement. In the next iteration, a value −1 is added to P. As long as P is greater than or equal to 1, the loop will continue. When the character in position P of the text string is a space, the comparison will be True. The statement inside the If block will then be executed. The *Exit For statement* will immediately terminate the loop, and transfer the execution control to the statement immediately following the Next statement, much like the Exit Do statement for the Do...Loop structure.
For example, assume the two words near position 80 are "getting excited" and are positioned as follows:

```
Position   69  70  71  72  73  74  75  76  77  78  79  80  81  82  83
Character  g   e   t   t   I   n   g       e   x   c   I   t   e   d
```

The following execution table depicts how the For loop is executed.

Iteration	P	Mid(Text, P, 1)	= " "	Action
1	80	I	False	(Next P)
2	79	C	False	(Next P)
3	78	X	False	(Next P)

4	77	E	False	(Next P)
5	76	(space)	True	Exit For

When the loop terminates, P should have a value of 76. After the position is found for the space, you can display the string in two separate lines as follows:

```
MsgBox(Microsoft.VisualBasic.Left(Text, P – 1) & vbCrLf & Mid(Text, P + 1))
```

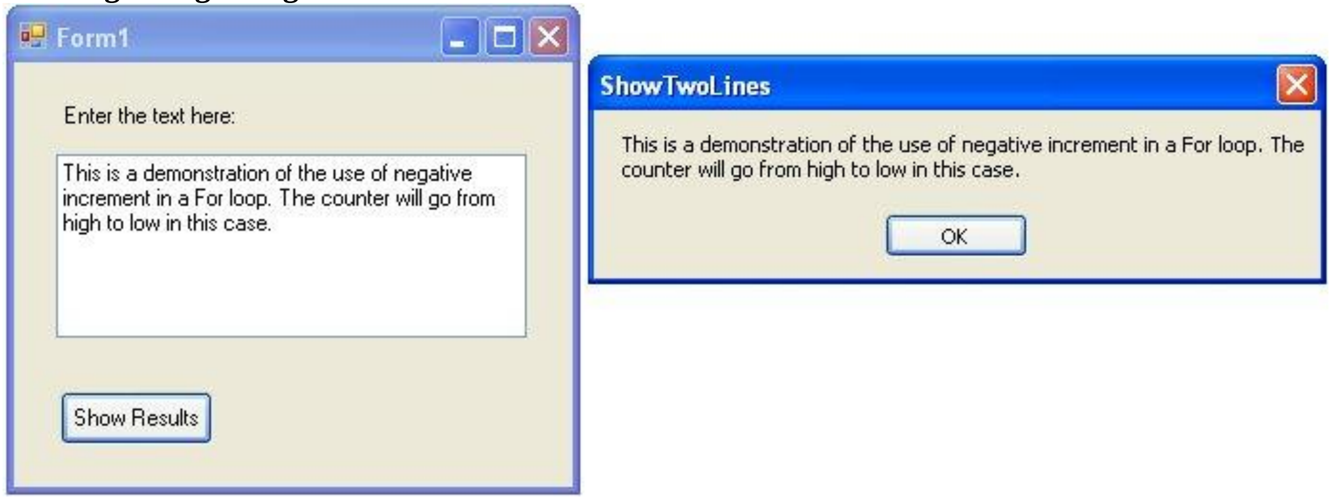
Recall that vbCrLf is the end of line marker, and will force the ensuing string to be displayed in the next line. The result will be similar to the following:

```
. . . getting
excited . . .
```

To test the preceding code, convert the problem into a project called ShowTwoLines. Here are the steps to create the visual interface:

- Draw a text box on a new form, and name it txtInput. This text box will be used to input the text string, which will be approximately 140 characters long.
- Draw a label above the text box. Put "Enter the text here:" as its text.
- Add a button to the form, and name it btnShow. The resulting visual interface should look similar to that shown in Figure 7-6.

Figure 7-6
Showing a long string in two lines



This is a demonstration of the use of negative increment in a For loop. The counter will go from high to low in this case.

This is a demonstration of the use of negative increment in a For loop. The counter will go from high to low in this case.

After the user enters the text string in txtInput, the user will click the button to initiate printing the text string in two lines.
With this modification, your code should be placed in the btnShow Click event. The complete code for the procedure should appear as follows:

```
Private Sub btnShow_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnShow.Click
    Dim P As Integer
    Dim Text As String
    Text = txtInput.Text
    For P = 80 To 1 Step -1
        If Mid(Text, P, 1) = " " Then
            Exit For
        End If
    Next P
    MsgBox(Microsoft.VisualBasic.Left(Text, P - 1) & vbCrLf & Mid(Text, P + 1))
```

```
End Sub
```

Notice that you have added the declaration for the variable Text. Because the source of the text string is the text box, txtInput, Text is assigned with the Text property of that control. If everything goes well, the program should display the text you have entered in the text box in two lines when you click the button. The code does not test whether the string is actually longer than 80 characters. If the string is not, the code will fail because the Mid function begins with testing the character at the 80th position. The additional code to safeguard against this error is left to you as an exercise.

Note that the preceding For loop was one way to find the position of a character in a string backward before VB6 was available. As you may recall, the InStrRev string function, discussed in Chapter 4, "Data, Operations, and Built-In Functions," can perform the same computation much more effectively. A statement such as the following:

```
    P = InStrRev(Text, " ", 80)
```

will give the same result as the For loop. Practically, you should use the For loop for this purpose only if you are using an older version of VB. Even so, this example should give you a fairly good idea about how a For loop works with a negative increment.

Nesting the Loops

Similar to the If blocks, the loop structures—the Do, While, and For structures—can be nested; that is, a Do...Loop structure can contain several Do...Loop structures inside itself. Similarly, a For...Next structure can contain several For...Next structures inside itself. In addition, a Do...Loop structure can also contain For...Next structures, and vice versa. The nesting can go practically any levels deep. The nested structure can appear as follows:

```
                          For I = 1 To 20
                                Statements to be executed (Level 1)
                                For J= I to 40
                                      Statements to be executed (Level 2)
                                      For K = J to 1 Step - 1
Level 1      Level 2      Level 3           Statements to be executed (Level 3)
                                      Next K
                                      Statements to be executed (Level 2)
                                Next J
                                Statements to be executed (Level 1)
                          Next I
```

Notice that the loops cannot be interposed; that is, one loop can contain another, and the inner loop must be complete before the end of the outer loop. In addition, you should not use the same counter variable for more than one loop. Example 9 illustrates the use of a nested loop structure.

Tip

When you are coding nested loops, you may want to have all the loop structures in place before starting to write the detailed code. After you start to focus on the detailed logic of your program, it is easy to become lost as to which part of the program you are in.

In some other cases, it may be easier to start with the innermost loop first and work outward one step at a time. After you finish the inner loop, you can treat it like a procedure that solves a problem. You can then use it to solve a bigger problem that is being handled in the outer loop.

Example 9. Matching Up Teams in a League

Suppose there are eight teams in a sport league. In a season, each team is to play with all other teams exactly once. You would like to list all the possible games showing which team is to play against the other team. How can this be done?

Consider the situation for Team 1. The team will play all the other teams, that is, Teams 2 through 8. You can list all these other teams the team will play against. How about Team 2? It will also play against all other seven teams. You should notice that the game Team 2 against Team 1 is the same game as Team 1 against Team 2. This game has already been listed for Team 1. There is no need to list this one again. So, for Team 2, you only need to list Teams 3 through 8. To avoid listing the same games, the general rule is to list as the opponents only those teams whose numbers are higher than the current team you are considering. This analysis yields the following table.

Pivotal Team	Opponents to List
1	2, 3, . . . 7, and 8
2	3, 4, . . . 7, and 8
3	4, 5, . . . 7, and 8
.	
.	
6	7 and 8
7	8

Notice that Team 8 is not listed as a pivotal team because it has been listed as the opponent to *all* other teams already.

Let *Team* be the focal team, and *Other* be the other team that Team will play against; then the following code should display in a list box, all the games for the league during a season.

```
    For Team = 1 To 7
        For Other = Team + 1 to 8
            lstGames.Items.Add("Team " & CStr(Team)
              & " against Team " & CStr(Other))
        Next Other
    Next Team
```

The inner For loop actually displays the other teams that Team will be playing against. The counter, Other, starts with a value Team + 1. As explained, all games against any teams with a team number smaller than this team have already been listed. The following table shows the results at different states of the two loops.

Team	Other	Game
1		
	2	Team 1 against Team 2
	3	Team 1 against Team 3
	.	
	.	
	7	Team 1 against Team 7
	8	Team 1 against Team 8
2		
	3	Team 2 against Team 3
	4	Team 2 against Team 4
	.	
	8	Team 2 against Team 8

.	.	
	.	
7		
	8	Team 7 against Team 8

Note that the inner loop with the counter, Other, repeats seven times. Each time, Other starts with a different value. When Team is 1, Other starts with 2; when Team is 2, Other starts with 3. The inner loop always ends after Other reaches 8. A total of 28 games should be displayed when the routine ends.

Example 10. What Is Your Chance to Win?

Nested loops are often used to solve complex computational problems. It is therefore important for you to become familiar and comfortable with their working. Here is another illustrative problem that calls for the use of nested loops: The probability for a gambler to win a hand of blackjack against the house is 49%. What is the chance to win at least 60 hands out of 100?

To solve this problem *analytically*, you will need to find the formulas to compute the probability to win 60 through 100 out of 100 hands and then sum the results. You can imagine how long the formula may look.

An alternative is to solve the problem *numerically by simulation*. You can use the random numbers generated by the computer to simulate the event and outcome. The resulting experience can then be used as an approximation to the probability of interest.

For example, the probability to win a hand is 49%, so you can draw a random number and then examine its value. If it is less than or equal to 0.49, you can consider that you have won a hand; otherwise, you have lost. Given sufficient repetition, the probability of winning will be approximately 49%, using this simulation scheme. To simulate playing a hand, you will draw a random number, and examine whether it is less than or equal to 0.49.

In the original problem, the question is the chance of winning 60 hands out of 100. You can simulate playing 100 hands. As each hand is played, you can determine whether you win or lose, and accumulate the number of hands won. At the end of this experiment, you can check to see how many hands you have won. If you have won at least 60 hands, you can claim it a success by the definition of the problem.

Let *R* be the random number, and *Wins* be the number of hands won; then the previous discussion can be translated into the following pseudo-code:

```
1. Let Wins = 0.
2. Repeat steps 2.a and 2.b 100 times.
    a. Draw a random number, R.
    b. If R is less than or equal to 0.49, add 1 to Wins.
3. If Wins is greater than or equal to 60, you have succeeded in winning 60 hands out of 100.
```

The code to implement the algorithm can appear as follows:

```
Wins = 0
For I = 1 To 100
    R = Rnd()
    If R <= 0.49 Then
        Wins = Wins + 1
    End If
Next I
If Wins >= 60 Then
    ' Claim that you have succeeded
End If
```

Notice that a For...Next loop has been used to play 100 hands. Inside the loop, a hand is played by drawing a random number, and the outcome (win or loss) is determined by comparing the random number with 0.49.

The preceding code gives the result of one experiment; that is, you experiment by playing 100 hands and see whether you succeed. To approximate the probability of success, you will need to experiment many times, some successful, others unsuccessful. The probability of success can then be approximated by the relative frequency by dividing the number of successes by the total number of experiments. Assume you experiment 1,000 times. The algorithm can be described as follows:

1. Repeat steps 1.a and 1.b 1,000 times.
    a. Experiment with 100 hands, and count the number of wins (W).
    b. If Wins is greater than or equal to 60, add 1 to Successes.
2. The probability of success equals Successes/1,000.

The code to implement the algorithm should appear as follows:

```
    For J = 1 To 1000
        ` Include the experiment to play 100 hands here
        If Wins >= 60 Then
        ` Claim that you have succeeded
            Successes = Successes + 1
        End If
    Next J
    ProbOfSuccess = Successes / 1000
```

The Complete Simulation Procedure

Assuming the computation will be initiated by a click on a button named btnSimulate, the complete procedure can appear as follows:

```
Private Sub btnSimulate_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnSimulate.Click
    Dim R As Single
    Dim Wins As Single
    Dim Successes As Single
    Dim I As Integer
    Dim J As Integer
    Dim ProbOfSuccess As Single
    Randomize() 'Randomize the random number sequence
    For J = 1 To 1000
        Wins = 0
        For I = 1 To 100
            R = Rnd()
            If R <= 0.49 Then
                Wins = Wins + 1
            End If
        Next I
        If Wins >= 60 Then
            ' Claim that you have succeeded
            Successes = Successes + 1
        End If
    Next J
    ProbOfSuccess = Successes / 1000
    MsgBox("Probability of winning 60 hands out of 100 is " & ProbOfSuccess)
End Sub
```

Notice that the Randomize statement has been inserted at the beginning of the program so that a different series of random numbers will be generated each time the simulation is performed. Test the program. You will see that the result oscillates around 2%. Does it surprise you?

Additional Remarks

The accuracy of the preceding simulation program depends primarily on the quality of the random number generator. If the random numbers are truly random, the results can be very accurate when the number of experiments is large.

Of course, you can modify the program so that it can be used to simulate a broad number of intriguing probability problems. For instance, in the previous example, the probability of winning a hand, the number of hands played, and the number of experiments are fixed at 0.49, 100, and 1,000, respectively. You can replace these simulation parameters with variables, and allow the user to specify their values. The program can then be used to answer a question such as “If the probability for a particular baseball player (say, Barry Bonds) to hit a home run each time he is at bat is 10%, what is the probability that he hits two or more home runs in a game when he is at bat five times?” The desired revision is left to you as an exercise.

The For Each...Next Structure

A variation of the For... Next structure is the For Each...Next structure, which has the following syntax:

```
    For Each Object In Collection
        Statements to be Executed
    Next [Object]
```

where *Object* is an object variable that represents an object such as a text box, and *Collection* represents a variable or predefined term that represents a set of object items; for example, the term *Controls* represents all the VB controls in a container. The expression, Me.Controls, represents the controls in the form.

It is probably easier to show you how the For Each structure works by example than by explanation of concept. The following code will count the number of text boxes in a form:

```
    Dim Ctrl As Control
    Dim Count As Integer
    For Each Ctrl In Me.Controls
        If TypeName(Ctrl) = “TextBox” Then
            Count += 1
        End If
    Next
```

As you can see in the code, the variable Ctrl is declared to be of a Control type, and Count is declared to be an Integer. Count is used to count the number of text boxes. In the For Each statement, the variable Ctrl is set to a control in the form. Inside the loop, the *TypeName function* returns a string representing the type name of the control. For example, if the control is a text box, the function returns a string “TextBox.” The string is compared with “TextBox” and if the two are equal, Count is increased by 1. In the next iteration, Ctrl is set to the next control in the Controls collection. The loop continues until the Controls collection runs out of the control object; that is, until all controls in the form have been enumerated.

Note that in the code, Me qualifies Controls. In VB 2008, each control container has its own Controls collection. All controls in the form but not within a container control, such as group box, are included in the form’s Controls collection. Those controls within a container control are included in the container’s Controls collection, but not in the form’s Controls collection. To enumerate those controls within a container control, qualify Controls with the container name. For example, to refer to those controls in a group box named GroupBox1, you should code:

```
    GroupBox1.Controls
```

Tip

Another way to identify the type of a control/object is to use the TypeOf keyword; however, it can be used only in the If statement as follows:

```
    If TypeOf(Ctrl) Is TextBox Then
        'Other statements
    End If
```

Notice that the Is operator must be used to test the type.

Clearing Text Boxes and Masked Text Boxes in a Form

With this understanding, you should be able to explain why the following code clears all the text boxes and masked text boxes in the form:

```
Dim Ctrl As Control
For Each Ctrl In Me.Controls 'Enumerate controls in the form
    Select Case TypeName(Ctrl)
        Case "TextBox", "MaskedTextBox"
            ' If the control is a text box
            Ctrl.Text = "" 'Clear the text box
    End Select
Next Ctrl
```

The routine uses the For Each...Next loop to enumerate through all the controls in the form. If a control is either a text box or a masked text box, a zero-length string is assigned to its Text property to clear the text box.

The Advantages of Generality

You may be wondering whether you would need a routine like this one at all. After all, when you develop a project and design the visual interface, you certainly know all the text boxes and the mask edit boxes in the form. Will it not be equally convenient to just code like the following?

```
    txtName.Text = ""
```

The answer is that the loop approach is better in most cases, especially for an application with many controls on a form. You do not need to know the names of the controls to make the code work properly. In addition, one statement, or a group of statements such as:

```
    Ctrl.Text = ""
```

will take care of clearing all controls of the same type.
Furthermore, when the visual interface is changed, such as addition or deletion of some controls, you do not have to worry about revising the code. It is generalized and will work properly for all text boxes and masked text boxes. This generic nature of the code enhances code maintainability.

Tip

So, how do you find out the type name of different controls? Use the TypeName function. The following routine will show the names and type names of all the controls in the form.

```
    Dim Ctrl As Control
    For Each Ctrl In Me.Controls
        MsgBox(Ctrl.Name & " " & TypeName(Ctrl))
    Next
```

Referencing Objects in a Collection by Index

Notice that you can also reference objects in a collection by index. Objects in the Controls collection, for example, can be referenced in a manner similar to the Items collection in the list box. The first object in the Controls has an index of zero. The Controls collection also has a Count property that gives the number of objects in the collection; therefore, the following code fragment can also count the number of text boxes on a form:

```
    Dim I As Integer
    Dim Count As Integer
    For I = 0 To Me.Controls.Count - 1
        If TypeName(Me.Controls(I)) = "TextBox" Then
            Count += 1
        End If
    Next I
```

7.3 Additional Notes on Coding Repetition

You have seen all the structures to construct code for repetition. Have you wondered under what situations you should use which structure (a question of *suitability*)? In addition, recall that statements within the loop will be repeated many times. Any performance efficiency/inefficiency associated with these statements can be magnified dramatically (a question of *speed* or efficiency). Are there tips that can enhance the efficiency of code for loops? Here are some notes in these regards.

Do Versus For

The key difference between the Do structure and the For structure is the absence or presence of the counter. This difference makes one structure more suitable than the other for certain situations. The following discussion provides additional details.

Suitability

The Do...Loop structure (and its variant, While…End While structure) has no counter; the For...Next structure does. In nearly all cases, one structure can be used in place of the other; however, one structure will appear to be more suitable than the other for a given situation. Basically, the For...Next structure is clearly intended for a situation in which the lower and the upper limit as well as the increment are known; the Do...Loop structure is designed for the situation in which such parameters are undeterminable.

For example, when reading an entire text file record by record, it will be difficult for the program to determine ahead of time how many records, or lines, there are in the file. In such a situation, it will be more appropriate to use the Do...Loop structure and check for the EOF condition to terminate the loop. Even if there is a need to use a counter while the input process is in progress, such as displaying the record number being read, a variable can be used to perform the count in the Do loop. You will be still better off not to use the For...Next structure by assuming an arbitrary upper limit and then check for the EOF condition inside the loop. Such an approach leaves open the possibility that in the future the arbitrary upper limit might be unexpectedly exceeded when the file grows too large.

On the other hand, the For...Next structure will be more suitable when its parameter values are known. Take the computation of the series of 1, 2, 3, to n, for example. At the time of computation, the value of n is certainly known, as are the starting value and the increment; therefore, it is natural that the For...Next structure is used. Although you can still accomplish the requirement by using the Do...Loop structure, you will find that more lines of code are needed. The code will not be as concise and easy to understand.

Speed

How does the For...Next structure compare with the Do...Loop structure in terms of speed? In general, the difference is negligible. In fact, the performance difference of the two structures depends more on the way the condition in the Do loop is coded and on the data types of the parameters in the For loop than on the structures themselves.

For example, similar to the condition in the If statement, a complex logical expression such as A >= B And C = D Or Not (X = Y And W < V) in the Do or Loop statement will be much slower than a simple expression such as A <> B. In the case of For loops, using variables of Decimal or Double type to serve as counter will certainly take more time to execute than using variables of the Short, Integer or Long type. Because loops are to be executed many times, their performance difference will be magnified and more noticeable. One of the Explore and Discover exercises at the end of this chapter illustrates the difference in speed between the Integer counter and the Decimal counter for the For...Next loop.

For Versus For Each

The For Each...Next structure is a convenient way to enumerate objects—especially of different types—in a collection, as you have seen in the example of enumerating all controls in a form. When you learn more about multiple form applications and databases in later chapters, you will find more opportunities to use this structure.

Note, however, that in many cases, the objects in a collection are of the same type. Usually, in such cases, the For...Next with a counter instead of with the keyword Each can also be used. When this is the case, the For...Next structure is usually more efficient than the For Each ...Next structure.

Tips for Efficiency

To repeat the point, because loops are to be executed many times, the efficiency or inefficiency of the group of statements in the loops is magnified. It is far more important to examine the code in the loops—especially the inner-most loops—very closely. Here are a few tips to improve the efficiency in coding the loops:

- *Move all constant assignments/operations out of the loop*. For example, consider the following loop:

```
For I = 1 To 20000
   A = 10 + B
   ‘ Other statements
Next I
```

  where none of the other statements inside the loop change the value of A or B. The code will be more efficient if it is rearranged as follows:

```
A = 10 + B
For I = 1 To 20000
    ‘ Other statements
Next I
```

- *Replace complex mathematical operations with simpler ones.* For example, the present value of an ordinary annuity of $1 for n years can be computed by the following formula:

$$P = (1 + r)^{-1} + (1 + r)^{-2} + \ldots + (1 + r)^{-n}$$

  Assuming the proper values for R and N have been obtained, a straightforward way to code the formula will be as follows:

```
Dim PresentValue As Double
Dim I As Integer
PresentValue = 0
For I = 1 To N
     PresentValue = PresentValue + (1 + R) ^ (-I)
Next I
```

The preceding code, however, is not efficient for two reasons. The value of (1 + R) will never change inside the loop. It will be more efficient to perform the computation before the loop. Also, the ^ operator takes much more time than a simple multiplication. The following code can accomplish the same and would be much more efficient:

```
Dim PresentValue As Double
Dim Factor1 As Double
Dim FactorN As Double
Dim I As Integer
PresentValue = 0
FactorN = 1 ' The present value of a dollar for n years
Factor1 = 1 / (1 + R) 'The inverse factor, i.e., (1 + r) ^ (-1)
For I = 1 To N
    FactorN *=  Factor1 'Compute V = V ^ I
    PresentValue += FactorN
Next I
```

The following execution table shows the state of each iteration.

I	FactorN	PresentValue
1	$(1 + r)^{-1}$	$(1 + r)^{-1}$
2	$(1 + r)^{-2}$	$(1 + r)^{-1} + (1 + r)^{-2}$
.		
N	$(1 + r)^{-n}$	$(1 + r)^{-1} + (1 + r)^{-2} + \ldots + (1 + r)^{-n}$

- *Avoid data type conversions inside the loop.* In the previous example, some might not see a need for the variable Factor1 or FactorN to be of the Double type; however, as long as PresentValue is of the Double type, making either Factor1 or FactorN (or both) as a Single type variable will force data type conversion inside the loop, creating additional work for the computer.
- *Replace reference to a control's property by a variable.* It takes longer to refer to a control's property than to a variable. For repetitive references in a loop, a variable should be used. For example, consider the following code:

```
For I = 1 To Len(txtName.Text)
    Ch = Mid(txtName.Text, I , 1)
    ' Other statements
Next I
```

The code will be more efficient if it is rearranged as follows, assuming TheText has been declared as a String variable:

```
TheText = txtName.Text
For I = 1 To Len(TheText)
    Ch = Mid(TheText, I, 1)
    ' Other statements
Next I
```

7.4 The W-4 Form Project: An Application Example

Loops are used extensively in programs but can be difficult for the beginner to understand. To illustrate how they can be used in a practical problem, consider a simplified aspect of a payroll system. When a new employee is hired, the payroll department needs to establish an employee record for the employee so that proper payroll withholdings can be handled. An aspect of this involves the completion of a W-4 form that requires the employee to divulge information concerning employee's tax status, such as the marital status and number of dependents. This simplified W-4 form example focuses on the programming steps, rather than the information required.

The W-4 Form Project

The project involves the entry of the employee W-4 form. Assume one or more new employees will be entered into the W-4 file each time the program runs and that the company has fewer than 100 employees. The program will require the entry of the following data:

- Employee Social Security number
- Name (last, first)
- Sex
- Number of dependents

The data will be saved in a file called W-4Info.txt as a fixed-length file (the length of each field remains constant), one line per record. The following lists additional specifications.

Design Considerations

One important consideration of the project is that there should be no duplicate employee records. To handle this, you will populate the existing employee records in a list box. A new employee record will be saved in the file and added to the list box if the new employee's Social Security number (SSN) is not found in the list box; otherwise, an error message will be displayed and the record will not be saved.

The Visual Interface

You will use the following controls to handle the data fields:

Data Field	Control
Employee Social Security number	Masked text box
Name (last, first)	Text box
Sex	Radio buttons
Number of dependents	Text box

To make the form appear uncluttered, you will use a group box to group all these fields. In addition to the VB controls for these data fields, you will add a list box to list the existing employees, as mentioned previously. Furthermore, you should also have two buttons: one to save the data, and another to end the program.

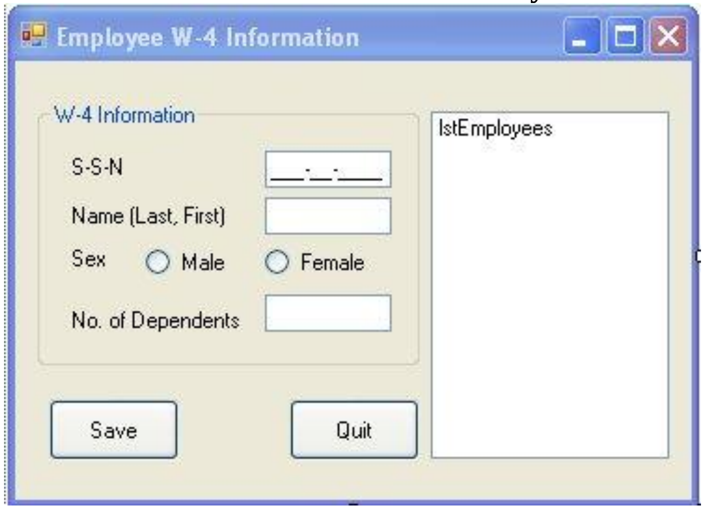
The properties for these controls should be set to proper values. Some selected ones are as listed in the following table.

Control (Field)	Property	Setting
Group box	Name	grpW4Info
	Text	W-4 Information
Masked text box (SSN)	Name	mskSSN
	Mask	"###-##-####"
Text box (Employ Name)	Name	txtName
	MaxLength	24
Radio button (for male)	Name	rdbMale

	Text	Male
	Checked	True
Radio button (for female)	Name	rdbFemale
	Text	Female
Text box (No. of dependents)	Name	txtNoOfDependents
	MaxLength	3
List box (to list employees)	Name	lstEmployees
Button	Name	btnSave
	Text	Save
Button	Name	btnQuit
	Text	Quit

The visual interface appears as in Figure 7-7.

Figure 7-7
The visual interface for W-4 form entry



Coding the Project

The code needs to handle the following situations:

- As soon as the program starts, the list box should be populated with employee records in the W-4 info file. This will involve:
    - associating the file with a StreamReader object
    - having the StreamReader read the records one at a time in a loop
    - adding each record to the list box as it is read
    - closing the StreamReader and reopening the file with a StreamWriter object so that data for additional employees can be saved during the remaining duration of the program.
- When the user clicks the Save button, several major steps should be carried out:
    - validating data entered. For simplicity, you will check only the employee's Social Security number to ensure that duplicate records for the same employee will not be allowed
    - saving the data in the W-4 Info file and adding the same record in the list box
    - clearing the screen to facilitate the entry of the next record
- When the user clicks the Quit button, the program should end.

Populating the List Box

The list box will be populated with all existing employee records. Assume all fields in each record will be kept in the list box and the file has been associated with a StreamReader W4FileReader. To populate the list box, you can set up a loop, and read the file one record at a time. As the record (named EmpRec) is read, it is added to the list box. The code in Example 2 can be easily modified to handle this:

```
    Do Until W4FileReader.Peek() = -1
        EmpRec = W4FileReader.ReadLine()
        lstEmployees.Items.Add(EmpRec)
    Loop
```

As already stated, this should be done as soon as the program starts. This routine should be placed in the Form Load event. This event procedure should also do the following:

- Open the file for input (associate the file with the StreamReader, W4FileReader).
- Close the StreamReader (W4FileReader) after all records have been read.
- Associate the file with a StreamWriter (W4FileWriter) with the Append specification because you will be adding new records for the remaining duration of the program.

The complete code for the Form Load event procedure appears as follows:

```
Dim W4FileWriter As System.IO.StreamWriter
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
  Handles MyBase.Load
    Dim EmpRec As String
    Dim W4FileReader As New System.IO.StreamReader("C:\Temp\w-4info.txt") 'Open the
      file to read
    ' Read employee w-4 info file and populate the list box
    Do Until W4FileReader.Peek() = -1
        EmpRec = W4FileReader.ReadLine()
        lstEmployees.Items.add(EmpRec)
    Loop
    ' Input complete; close the file
    W4FileReader.Close()
    ' Reopen the file for Append
    W4filewriter = New System.IO.StreamWriter("C:\temp\w-4info.txt", True)
End Sub
```

Note that the variable W4FileWriter is declared in the general declaration area at the class level. This is necessary because the variable will also be used in another event procedure in which the employee W-4 records are saved; however, the variable W4FileReader is declared inside the form load procedure because no other event procedure will need to use it to read the file. Note also that this routine will not run when it attempts to open the W-4 Info file for Input if it does not exist. You should create an empty file with Notepad before testing this program.

Handling the Save Button

As indicated previously, when the user clicks the Save button, the program should do the following:

- Ensure that the current entry is not a duplicate of an existing record.
- Save the record, if it is a new one.
- Clear the screen.

These steps can be coded in the btnSave click event procedure as follows:

```
Private Sub btnSave_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnSave.Click
    ' Make sure SSN does not already exist
    ' Don’t save the record if it does exist
    If SSNExists() Then Exit Sub
    ' Save the employee record
    ' Also add the record to the list box
    SaveRec()
    ' Clear screen and reset the default radio button
    ClearScreen()
    ' Set focus on the first control for the convenience of the user
    mskSSN.Focus()
End Sub
```

This coding design takes advantage of using general procedures for the top-down design. Each major step is coded either as a function or as a Sub and is invoked from this procedure. The code should be self-explanatory.

The first step is to make sure the Social Security number entered does not already exist in the file. To do this, the event procedure calls a function, SSNExists, which returns a value of True or False, depending on whether the Social Security number actually exists. How does the function carry out the checking? To understand how it is done, you need to understand the format of the records saved first.

Saving a W-4 Record.

As mentioned, the file is created as a fixed-length file; that is, each field in the file remains constant. Assume the lengths for SSN, name, sex, and number of dependents are 9, 24, 1, and 3, respectively. The lengths for SSN and sex should pose no problem because their field lengths appear fixed. For ease of browsing the file, you will also pad a blank space between each field.

How do you handle the length for Name? Note that the actual length of names can vary. You can solve this problem by creating a function (SetLength) that will take a string, and set its length according to the specified length. The function appears as follows:

```
Function SetLength(ByVal Text As String, ByVal L As Integer) As String
'This function returns a string of the specified length L,
‘given Text as the first parameter
    If Len(Text) >= L Then
        Return (Microsoft.VisualBasic.Left(Text, L))
    Else
        Return (Text & Space(L - Len(Text)))
    End If
End Function
```

As you can see from the function’s header, the function takes two parameters. The first parameter is the string itself, and the second parameter is used to specify the desired length. Inside the procedure, the length of the string (Text) is compared with the desired length. If the former is greater or equal to the latter, the first L (desired length) characters of the string are returned. On the other hand, if the string is shorter than the desired length, it is padded with additional spaces to make it of the exact desired length.

How do you ensure that the number of dependents will be always three characters long? You can handle this in several different ways. This is a numeric field, but you are not sure whether the user will always enter a number, or enter additional spaces after the number; therefore, the first step is to convert the content of the text to a number using the Val function, and convert it back to string using the CStr function. You can then pad on its left side with spaces. One way to do it without involving an explicit use of the If block is to always pad the result with three spaces and then take the only the three characters on the right. The code will appear as follows:

```
    ‘Generate a string (from a number) of length 3 with space padded on the left
    Microsoft.VisualBasic.Right(Space(3) & Cstr(Val(txtNoOfDependents.Text)) , 3)
```

The following code for Sub SaveRec should save the employee record in the format you want.

```
Sub SaveRec()
    ' This Sub saves an employee w-4 record
    ' It also adds the same to the list box
    Dim TheName As String
    Dim Sex As String
    Dim EmpRec As String
    Dim SSN As String
    TheName = SetLength(txtName.Text, 24)
    If rdbMale.Checked Then
        Sex = "M"
    Else
        Sex = "F"
    End If
    SSN = Replace(MskSSN.Text, "-", "") 'Remove - from SSN
    EmpRec = SSN & " " & TheName & " " & Sex & " " _
      & Microsoft.VisualBasic.Right(Space(3) & _
      CStr(Val(txtNoOfDependents.Text)), 3)
    W4FileWriter.WriteLine(EmpRec)
    ' Also add to the list box
    lstEmployees.Items.Add(EmpRec)
    ' Tell the user what has been done
    MsgBox("Employee data for " & Trim$(TheName) & " Saved")
End Sub
```

Notice that in the code, you also convert what the user specifies as the employee's sex by the radio buttons—rdbMale and rdbFemale—to a string variable, Sex, which is assigned a value "M" or "F", depending on which radio button is clicked. Notice also that the variable EmpRec is used to prepare the employee W-4 record to be saved. In addition, the Replace function is used to remove – from the masked text box's Text property. After the W-4 Info record is assembled, it is saved to the file. The same record is also added to the list box for content consistency and to facilitate the prevention of duplicate employee records.

Checking for Existence of an SSN

Now you know the Social Security number is the first field in the file and the list box. Searching for the SSN involves the comparison of the data entered in the masked text box with the first field of each item in the list box until a match is found or the list box runs out of items. To speed up the operation, a string variable SSN can be used to store the data extracted from the masked text box. (Recall that data stored in variables can be accessed much faster than those properties in controls.) The pseudo-code to search for the SSN can appear as follows:

1. Let SSN = the number extracted from the masked text box.
2. For each I = 0, 1, 2... item count in the list box minus 1, perform the following:
    a. Compare SSN with the SSN in the Ith item in the list box. If the two are equal, conclude that SSN exists and exit the function; otherwise, proceed to next I.
    b. When all items are compared, exit the function.

How do you extract data in the first field (SSN) of a list item? Each item (record) in the list box can be accessed by the list box's Items(I) property. To extract the first sub-string from a string, you can use the Left function. Because the SSN field has a length of 9, the expression Left(lstEmployees.Items(I), 9) should return the SSN of the Ith (0th being the first) employee. How do you set the value for I? The value can vary in the range of 0 through Items.Count –1. This range can be used to set up the counter of a For loop. The following code tests the existence of a given SSN and returns a value True when the SSN is found:

```
Function SSNExists() As Boolean
' This function checks if the SSN in the masked text box
' already exists; if so, it displays an error message
' Returns True; otherwise, it returns False
    Dim SSN As String
    Dim I As Integer
    SSN = Replace(mskSSN.Text, "-", "") 'Remove -
    For I = 0 To lstEmployees.Items.Count - 1
        If SSN = Microsoft.VisualBasic.Left(CStr(lstEmployees.Items(I)), 9) Then
            MsgBox("SSN already in file", vbInformation)
            Return (True)
        End If
    Next I
    Return (False) 'Entered SSN not found in the list box; return false
End Function
```

In the loop, when SSN (the entered SSN) is found to be equal to one of the SSNs in the list box, a message is displayed. The function then terminates with a return value True; otherwise, the search continues until the end of the loop. The function returns a value of False.

The FindString Method

As a side note, the list box and the combo box of VB 2008 have the *FindString method* that returns the position of the string in the box that starts with the specified string. If the string is not found, the method returns a value –1. The method has the following syntax:

```
    Object.FindString(StringToSearch, [BeginningPosition])
```

If the second parameter is specified, the search will start at this specified position. The preceding SSNExists function can be rewritten as follows:

```
Function SSNExists() As Boolean
    ' This function checks if the SSN in the masked text box
    ' already exists; if so, it displays an error message
    ' Returns True; otherwise, it returns False
    Dim SSN As String
    SSN = Replace(mskSSN.Text, "-", "")
    If lstEmployees.FindString(SSN) >= 0 Then
        MsgBox("SSN already in file", vbInformation)
        Return(True)
    End If
    Return(False)
End Function
```

The code here is definitely much simpler and more efficient. The previous code is used to illustrate how the For loop can be used. (And was also the typical way to find the matching string prior to VB.NET.) Also note that the list box and the combo box also have the FindStringExact method that returns the position of the string in the box that exactly matches the specified string. This method has exactly the same syntax as the FindString method.

Tip

Use the FindString and FindStringExact methods of the list box and combo box to search for a match in the box. They make your code much simpler.

Clearing the Screen

The third major step in the Save event procedure is to clear the screen. The previous code example to clear the text boxes and masked text boxes on the screen (form) can be copied and used here. Notice, however, that you have qualified Controls with grpW4Info (name for the group box which contains all data entry boxes) so that the routine will clear all text boxes and masked text boxes in that container. Can you write a routine that can clear the screen, regardless of whether the text boxes and masked text boxes are in any container control? Yes. This challenge is left to you as an exercise at the end of this chapter.

This W-4 info screen also has radio buttons. What you would like to accomplish is to reset the default button, such as set rdbMale on. To do this in the For Each loop appears to be a bit trickier. You will take the direct approach of setting the Checked property of rdbMale to True. The complete code for this Sub appears as follows:

```
Sub ClearScreen()
    Dim Ctrl As Control
    For Each Ctrl In grpW4Info.Controls 'Enumerate controls in the group box
        Select Case TypeName(Ctrl)
            Case "TextBox",”MaskedTextBox”
                ' If the control is a text box
                Ctrl.Text = "" 'Clear the text box
        End Select
        rdbMale.Checked = True
    Next Ctrl
End Sub
```

Don’t forget to Close the StreamWriter

As pointed out in Chapter 6, “Input, Output, and Procedures,” it is extremely important that the StreamWriter be closed before the program ends; otherwise, the data may not be written to the file. The output will be incomplete. As noted in Chapter 6, the most appropriate event to close the object is the form’s closing event. The code appears as follows:

```
Private Sub Form1_FormClosing(ByVal sender As System.Object, ByVal e As
  System.ComponentModel.CancelEventArgs) Handles MyBase.Closing
    W4FileWriter.Close()
End Sub
```

Finally, when the user clicks the Quit button, the program should end. This is left to you. This completes the code for the application example. The following table lists all the procedures developed for this application:

Procedure name	Procedure type	Remark/purposes
Form1_Load	Event	Open the W4 file for input
		Populate the list box with records read from the W4 file
		Close the W4 file
		Reopen open the W4 file for output
BtnSave_Click	Event	Call four other procedures related to the “save record” operations
SetLength	General function	Returns a string with the specified length
SaveRec	General sub	Saves the record (as entered on the screen) to the W4 file
		Add the new record to the list box

		Displays a “record saved” message
SSNExists	General function	Returns a Boolean value indicating whether the specified SSN already exists (Note that there are two versions. Include only one to test.)
ClearScreen	General sub	Clears the text boxes and masked text box and also resets rdbMale as the default selection
Form1_Closing	Event	Closes the W4 file
BtnQuit	Event	Terminates the program

Testing the Program

Run the program. You should be able to observe the following:

- As soon as the program starts, you should be able to see the existing employees in the list box. If you do not see all fields for the employee in the list box, stop the program and then set the list box’s *HorizontalScrollBar property* to True, or make the list box wider.
- After you have entered data for an employee and clicked the Save button:
    - If the SSN already exists, the program should display a message box and wait for your additional action.
    - If the SSN is new, the data will be saved and the screen will be cleared. In addition, the computer will display a message indicating that the data have been saved. Also, the new record is added to the list box.
- When you click the Quit button, the program terminates, assuming you have entered the proper code.
- When you run the program again, all data that you previously entered should appear in the list box. If not, check to see if you have the code to close the StreamWriter, W4FileWriter.

Additional Remarks

This application example was made as simple as possible. As simple as it is, however, it illustrates how useful and widely used loops are. You have seen that all the following activities involve the use of loops:

- Populating the list box
- Reading records from a file
- Looking for certain data in a list (list box is just an example)
- Clearing edit boxes in a container

You may have also noticed that each situation calls for a different loop structure. The implication is that you should not rely exclusively on only one structure. Become well acquainted with all of them, and you will become more proficient in VB programming.

If you compare this example with the one at the end of the preceding chapter for the phone file update, you may find some differences in design. For example, the current example involves only one file. Recall that a case was made in Chapter 6 that updating a text file sequentially calls for the use of two files (one for input and one for output). The current example imposes additional restrictions. Records can only be added to the file. No record can be recalled for editing, nor can a record be deleted from the file. In addition, the entire file is read into a list box. This is feasible only when the file is fairly small.

Summary

- There are basically two types of structures for repetition: one based on condition, and the other based on counter. The Do… Loop structure is based on condition; the For … Next structure is based on counter.
- In the Do…Loop structure, a condition expression can be placed either with Do or with Loop. The condition can be any expression that evaluates to a Boolean value. The condition must start with a keyword either While or Until. If the keyword is While, the repetition continues as long as the condition is True. If the keyword is Until, the repetition continues as long as the condition is False.
- In the Do…Loop structure, if neither the Do statement nor the Loop statement has a condition, the loop can only be terminated with an Exit Do statement or other statements that terminate the procedure inside the loop.
- The Do While…Loop structure is equivalent to the structure of While…End While. To exit the latter structure, use the Exit While statement.
- The For…Next structure has the following syntax:

```
For Counter = Starting Value To Ending Value [Step Increment]
    Statements to be executed
Next [Counter]
```

  When the loop starts, the counter is set to the starting value and compared with the ending value. If it passes the ending value, the loop is terminated. In each of the subsequent iteration, the current value of the counter is added by the increment and compared with the ending value. Again, if it passes the ending value, the loop is terminated. The comparison to determine whether the counter's value passes the ending value depends on the sign of the increment. If the increment is positive, the counter is considered to have passed the ending value when the counter is greater. If the increment is negative, the counter is considered to have passed the ending value when the counter is less.
- The default increment in the For…Next structure is 1.
- The For Each…Next structure relies on the enumerator to enumerate all the objects in a collection. The loop ends when all objects in the collection have been enumerated.
- The Do…Loop structure (and its variant, While…While End structure) is most proper when the repetition depends on certain known condition; for example, reaching the end of file. The For…Next structure is most suitable when the beginning and ending value of the counter are known before the execution of the loop. The For Each…Next can be very useful in enumerating objects in a collection.
- Many mathematical problems whose complexity in the equation defies analysis can be solved numerically by approximating the solution iteratively. These numerical methods can also be applied to many business problems. (For a sample, see exercise 7-33 and project 7-38 at the end of this chapter.)
- Loops execute statements within their structures many times; therefore, they magnify the efficiency of the code noticeably. Always analyze code within the repetition structure (especially the inner most loop) very carefully to ensure code efficiency. Tips to enhance code speed include:
    - Move constant assignment statement out of the loop.
    - Replace complex computation inside the loop with simpler one.
    - Assign object properties to variables outside of the loop, and use the variables inside the loop.

Explore and Discover

7-1. Conditions with Both the Do and Loop Statements. Enter the following code in the form click event:

```
    Dim I As Integer
    Dim C As Integer
    I = -10
    Do Until I > 0
        I = I + 1
        C = I - 1
    Loop While C < 0
```

You should notice that the keyword While is underlined. Place the mouse cursor on the word. What does the IDE say?

7-2. Changing the Parameters of a For Block. Enter the following code:

```
Private Sub Form1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
  Handles MyBase.Click
    Dim I As Integer
    Dim K As Integer
    Dim Z As Integer
    Dim Total As Integer
    K = 1
    Z = 10
    For I = K To Z
        Total = Total + I
        K = 10
    Next I
    MsgBox(Total)
End Sub
```

Run the project and then click the form. What is the value of Total? Does the change in value for K inside the loop affect the loop?

7-3. Changing the Parameters of a For Block. (continued from exercise 7-2). Test the following code:

```
Private Sub Form1_Click(ByVal sender As System.Object,  ByVal e As
  System.EventArgs) Handles MyBase.Click
    Dim I As Integer
    Dim K As Integer
    Dim Z As Integer
    Dim Total As Integer
    K = 1
    Z = 10
    For I = K To Z
        Total = Total + I
        Z = 1
    Next I
    MsgBox(Total)
End Sub
```

What is the value of Total? Does the change in value for Z inside the loop affect the loop?

7-4. Changing the Parameters of a For Block. (continued from exercise 7-3). Test the following code:

```
Private Sub Form1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
  Handles MyBase.Click
    Dim I As Integer
    Dim K As Integer
    Dim Z As Integer
    Dim Total As Integer
    Dim S as Integer
    K = 1
    Z = 10
    S = 1
    For I = K To Z Step S
        Total = Total + I
        S = 2
    Next I
    MsgBox(Total)
End Sub
```
What is the value of Total? Does the change in the value of S inside the loop affect the loop? What is your conclusion concerning the effect of changing the parameter values of the For loop?

7-5. Changing the Counter of a For Block. Test the following code:
```
Private Sub Form1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
  Handles MyBase.Click
    Dim I As Integer
    Dim K As Integer
    Dim Z As Integer
    Dim Total As Integer
    Dim S as Integer
    K = 1
    Z = 10
    S = 1
    For I = K To Z Step S
        Total = Total + I
        I = 10
    Next I
    MsgBox(Total)
End Sub
```
What is the value of Total? Why is the result different from that of previous exercises? The parameters for the loop are set the first time the loop is executed and remain the same. The counter value (which is allowed to be changed inside the loop) is always compared with the ending value each time the For statement is executed. (*Note:* Changing the counter value inside the loop can make it difficult to trace the code logic. Avoid such a coding practice.)

7-6. Fractional Progression. Type the following code in the form click event procedure. Run the project, and observe the result:
```
    Dim I As Single
    Dim C As Integer
    For I = 1 To 100 Step 0.1
        C = C + 1
    Next I
    MsgBox( "C = " & C)
```
Is the result what you expected? How do you explain it?

7-7. Fractional Progression. (continued from exercise 7-6). Make sure you have set Option Strict On. Change the declaration for I in the preceding problem to the following:
```
    Dim I As Integer
```
Do you see a portion of the following line highlighted?
```
    For I = 1 To 100 Step 0.1
```

Place the mouse point on the highlighted line to find out why.

7-8. Fractional Progression. (continued from exercise 7-7). Place the following code as the first line in the code window of the preceding project.

```
Option Strict Off
```

Now run the project. What result do you obtain? Why does the code result in an error? Click the Break button on the error message box. Place the mouse pointer on the variable C in the For loop. What value do you see? That value basically indicates that C has an overflow condition. Why does the program result in an overflow for C? The loop turns out to be an endless one. I is an integer. After I is added by 0.1, it is truncated to be 1; therefore, the counter never increases, regardless of how many iterations the loop executes. You can see this effect by setting the breakpoint on the For statement, pressing F8, and examining the value of I for a few iterations.

7-9. Counter Value After Exit For. Consider the following code:

```
    Dim I as Integer
    For I = 1 To 10
        If I = 5 Then Exit For
    Next I
    MsgBox (I)
    For I = 1 To 10
    Next I
    MsgBox (I)
```

Place the code in an event procedure, such as a button Click or Form Click event, and test it. What results do you obtain? When the loop is terminated by Exit For, the value of the counter is exactly what it is within the loop. When the loop is terminated from the For statement by comparing the counter value with the ending value, the counter value should be one increment value as specified in the Step parameter away from the ending value.

7-10. The Puzzling Fractional Increment. Test the following code in the form click event:

```
    Dim S As Single
    For S = 0 To 2 Step 0.2
    Next S
    MsgBox(S)
```

What is the last number in the message box? Is it 2, which is what you should expect? As explained in Chapter 4, Single type variables have a precision of approximately seven significant digits. Some numbers are hard for this type of variables to represent exactly. Repetitively adding 0.2 to S causes the variable minutely off its true value. Do you see the potential problem with using the Single type variable, especially as a counter? (Contributed by Wen-Hao Chuang via Ben Wu, both graduates of Indiana University.)

7-11. Use of the For Each...Next Structure. Draw a few different types of controls on a form. Give each a name of your favorite; then test the following code in the form click event:

```
    Dim V As Object
    For Each V In Me.Controls
        MsgBox(V.Name)
    Next V
```

Does it work properly? Set Option Strict Off if you see V.Name underlined. Do you see the flexibility of the For Each…Next? Notice also that V is declared as Object, which can also be any type of data. When you set Option Strict Off, you have much more flexibility in your code. But when your code does not produce the expected results, you may not know the exact statement that causes the problem.

7-12. Does It Really Matter What Data Type the Counter Is? Draw two buttons on a new form. Name one btnInteger and set its Text property to Integer. Name the other btnDecimal and set its Text property to Decimal. Enter the following code:

```
Private Sub btnInteger_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnInteger.Click
    Dim I As Integer
    Dim J As Integer
    Dim T As Double
    T = Microsoft.VisualBasic.Timer
    For I = 1 To 10000
        For J = 1 To 10000
        Next J
    Next I
    MsgBox(Microsoft.VisualBasic.Timer - T)
End Sub

Private Sub btnDecimal_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnDecimal.Click
    Dim I As Decimal
    Dim J As Decimal
    Dim T As Double
    T = Microsoft.VisualBasic.Timer
    For I = 1 To 10000
        For J = 1 To 10000
        Next J
    Next I
    MsgBox(Microsoft.VisualBasic.Timer - T)
End Sub
```

Notice that the only difference between the two procedures is the type of data declared for the counters. Click the Integer button. Observe the time displayed in the message box. Do the same with the Decimal button. Do you see a difference in the time it takes to execute each procedure? Decimal counters use much more time (more than 5 times) than Integer counters in getting an empty loop executed.

Exercises

7-13. Series. Write a routine to compute the total of 1+ 2 + 3 +. . . + n, where n is taken from a text box. The result is displayed in the Text property of the lblSeries label when the user clicks the btnCompute button.

7-14. Factorial. Write a routine to compute the product of 1 x 2 x 3 x . . . x n, where n is taken from a text box. The result is displayed in the text of a label called lblFactorial when the user clicks a button named btnCompute. Use the Double data type for the result. (*Remarks:* Recall that this problem was solved by the use of a recursive procedure. Most problems that can be solved by recursion can also be solved by iteration. Iteration can be easier for the computer, but recursion is typically easier for human beings to solve a complex problem.)

7-15. Setting the SelectedIndex Property of Combo Boxes in a Form. Write a Sub procedure and name it ComboSetter; set the SelectedIndex property of all combo boxes in your form to zero when the user clicks the button with a text Set Default. Your procedure should not explicitly refer to the names of the combo boxes. Test the program using a form that contains combo boxes mixed with other controls. Make sure each combo box contains at least one item. The Sub should be invoked by a call from a button's Click event procedure. (*Hint:* Use the For Each loop to enumerate all controls and the TypeName function to identify the combo boxes in the form.)

7-16. Are All Text Boxes Blank? Write a routine to check whether all text boxes in a form are blank. When the user clicks the button with the text, "All Blank?", your program will display either the message "Yes. All text boxes are blank," or the message "No. Some text boxes contain data," depending on the state of the text boxes. (*Hint:* Use the For Each loop to enumerate all controls and the TypeName to identify the text boxes in the form. *Note:* Write the routine as a function that returns a Boolean value; then use this function from the Click event procedure.)

7-17. Reversing a String. Develop the code to reverse a string; that is, given a string "abcde," your code will produce "edcba" as a result. Test your code by taking the string from a text box named txtOriginal into which the user enters the data and displaying the result in the Text property of a label named lblReversed. (*Note:* VB 2008 has a reverse string function—StrReverse—to handle this; however, the purpose of this exercise is for you to practice using the loop—don't allow yourself to use that built-in function.)

7-18. Encrypting a String. Write code to encrypt a string in a text box named txtUserName using the 13 rotation algorithm. This rotation algorithm replaces each letter in the string with a letter in the alphabet 13 positions from it. For example, the letter a (first position) will be replaced by n (14th position), b will be replaced by o, and so on; n is then replaced by a, o is replaced b, and so on. The result should be displayed in the same txtUserName text box when the user clicks the button named btnEncrypt with the text Encrypt/Decrypt. (*Hint:* Use a For loop to work on each character in the string at a time. Use the Mid function to extract a character from the string. Examine whether it is a letter. If it is, find its keycode value, and use it to find its position in the alphabet. Add 13 to the position, and obtain its remainder against 26. The remainder gives the encrypted position. Convert that position to the corresponding letter. You will need to use the ASC and Char functions.)

7-19. Counting the Number of Periods in a Paragraph. Write a function to count the number of periods in a text string (paragraph), which is passed to the function as a parameter. Test the function by calling it from an event procedure. (*Hint:* Use the InStr function in a Do loop. Take advantage of its first [optional] parameter.)

7-20. Counting the Number of Words in a Paragraph. Write a function that will return the number of words in a text string passed to the function as a parameter. (*Hint:* Trim off the leading and trailing spaces of the text. The number of words should be equal to the number of groups of spaces; that is, a group of consecutive spaces should be counted as one.)

7-21. Counting the Number of Radio Buttons in a Form. Write a routine to count the number of radio buttons in a form by using the For Each...Next structure. Test your code by drawing five radio buttons on a form in addition to one text box, one picture box, one list box, and one button. The button should have a name btnCount with the text, Count. After you click the Count button, your routine should display the message, "There are 5 radio buttons on the form."

7-22. Listing All Controls in a Form. Recall that if the form has a container control such as group box, Me.Controls will not enumerate any of the controls contained in the container in the For Each…Next enumeration. Write a Sub procedure that will display the names of all controls in the form, regardless of whether the controls are in a container or in the form itself. (*Hint*: Each control has a Controls collection that has the Count property. This property indicates whether this control contains any other controls. Your Sub should recursively go deeper into each layer of the container.)

7-23. Clearing the Contents of All Text Boxes and Masked Text Boxes in the Form. Modify exercise 7-22 so that it will clear all the text boxes and masked text boxes in the form.

7-24. Adding an Item to the Combo Box. Set up a new form such that it will contain the following fields for data entry:

Field	Control	Control Name
Name	Text box	txtName
Address	Text box	txtAddress
Zip code	Combo box	cboZipCode

There should be two buttons: one with the text, Save, and the other with the text, Quit.
When the user clicks the Save button, your program checks whether the Zip code entered in the combo box (cboZipCode.Text) already exists in its list. If not, this new Zip code is added to the list.

7-25. Populating and Saving the Content of a Combo Box. Modify exercise 7-24 in the following manner:

- When the program starts, the combo box will be populated with a list of Zip codes in a file called ZipCodeList.txt.
- When the user clicks the Save button, if the Zip code is new, it is added to the combo box's list and saved in the file. (*Hint:* In the Form Load event, after the combo box is populated, close the file; then reopen it for Append. Close this file in the form's Closing event.)

7-26. Generalized Routine for Series Computation. Write a function to compute the value of an infinite series of the form:

$$S = a + a(1/d) + a(1/d)^2 + a(1/d)^3 + \dots + a(1/d)^n$$
$$\text{where } d > 1$$

The function should have the following header:

```
   Function Series(A As Double, D as Double) As Double
```

Test your routine by calling it from a button Click event. The values of a and d should be taken from two text boxes whose values are entered by the user.

7-27. Computing the Present Value of an Annuity. The present value of an annuity can be computed with the following formula:

$$P = a(1 + r)^{-1} + a(1 + r)^{-2} + a(1 + r)^{-3} + \dots + a(1 + r)^{-n}$$

where a = the amount of payment per period,
r = the interest rate per period,
n = the number of periods, and
p = the present value of the annuity.

Write a routine to compute p and then display the result when the user clicks the button with the text, Compute. The values of a, r, and n are to be taken from the text boxes named txtAmount, txtRate, and txtPeriod. (*Note:* Do not use the PV built-in function for this purpose; instead, set up a For loop to perform the computation. Also, apply the best algorithm for efficiency.)

7-28. Computing the Present Value of a Perpetual Annuity. Consider the problem in exercise 7-27 again. Suppose n (the number of periods) is infinite, such an annuity is recognized as perpetual annuity. Modify your routine to compute the result.

7-29. The Present Value of a Bond. A coupon bond pays an interest semiannually at the stated rate of its coupon, such as 6%. On the date of maturity, it pays its face value of $1,000 per unit, in addition to its interest coupon. Because the coupon rate can seldom be the same as the market rate that fluctuates daily, the price or present value of the bond is seldom the same as its face value. The formula to compute the present value (price) of a coupon bond is as follows:

$$P = c\ (1 + r)^{-1} + c\ (1 + r)^{-2} + c\ (1 + r)^{-3} + . . . + c\ (1 + r)^{-2n} + f\ (1 + r)^{-2n}$$

where *P* = the present value (price) of the bond,

c = coupon rate x Face value/2,

r = market rate/2,

n = number of years to maturity, and

f = face value (usually $1,000).

Write a function to compute the price of a bond given the coupon rate, market rate, years to maturity, and face value. Set all of these as the function's parameters.

Your visual interface should include three text boxes for the user to enter the coupon rate, the market rate, and years to maturity. You can assume that the face value is $1,000. You should also have a text box (set its ReadOnly property to True) to display the result of computation (price). There should also be two buttons: one to initiate the computation, and the other to end the program. Invoke the function from the Click event of the button to initiate computation and display the result.

7-30. Computing the Square Root of a Number. The square root of a number can be computed by a set of iterative formulas:

$$D=(X_n^2 - A) / (2\ X_n)$$
$$X_{n+1} = X_n - D$$

where *A* = the number for which to compute the square root,

X = the square root of A, and

N= the subscript of X, denoting the number of iterations.

The formulas state that you can start with any positive number for X as an approximation for the square root of A. (You can start with A, except when it is zero or one. In the case of zero or one, there no need to compute. The root is the same as A.) A better approximation can then be computed by subtracting from the current approximation (X_n) a value D, which is computed by the formula just given.

Each approximation will move closer to the true square root of A. When D (the amount of adjustment/correction) is sufficiently small, the new X can be considered a good approximation to the root. You can then stop the iteration (repetition). You can consider D sufficiently small when Abs(D/X) < 0.0000001.

Write a function (call it SqareRoot) to compute the square root of any positive number using this method. Test your program by taking a number from a text box entered by the user and displaying the result in another text box (set its ReadOnly property to True). Compare your result with the result given by the Sqrt built-in function.

7-31. Computing the Cube Root of a Number. Refer to exercise 7-30. In a similar fashion, the cube root of a number can be computed by a set of iterative formulas:

$$D=(X_n - A / X_n^2) / 3$$
$$X_{n+1} = X_n - D$$

Write a function to compute the cube root of any positive number using the same iterative method. Test your results with the following numbers: 1, 8, 27, 64, and 125. The solution should be 1, 2, 3, 4, and 5.

7-32. Computing the Economic Order Quantity (EOQ): The Sequential Method. The relevant cost of an inventory replenishment policy has an optimum point with respect to the quantity reordered each time. The total inventory cost for such a purpose is given as follows:

```
    TCC = CC x (Q / 2)
    TOC = PC x (D / Q)
    TIC = TCC + TOC
```

where *Q* = the quantity to order each time,
CC = the cost to carry an item per period,
PC = the cost to place an order,
D = the total demand (units) per period for the product,
TCC = total carrying costs,
TOC = total ordering costs, and
TIC = total relevant inventory cost.

You can compute the total relevant inventory cost beginning with Q equals 1; then perform the cost computation when Q is increased by 1 each time. Initially, you will find the cost decreases as Q increases. Eventually, you will find the total cost increases as Q increases. The EOQ is the quantity at which the total cost is the lowest—before it starts to increase as Q increases.

Write a routine to compute the Q at which the total inventory cost is the lowest, using this method. Your visual interface should include three text boxes for the user to enter the total demand per period, the carrying cost per unit per period, and the cost to place an order. You should also have a text box (set its ReadOnly property to True) to display the result (EOQ). In addition, you should have two buttons: one to initiate the computation, and one to end the program.

7-33. Computing the EOQ with the Half Interval Method. The preceding problem can be solved using another, more efficient algorithm. If you draw a graph depicting the cost behavior of the total carrying costs and the total order costs, you will notice that a Q below the optimum EOQ, the total ordering costs are higher than the total carrying costs. A Q above the EOQ, however, has higher total carrying costs than the total ordering costs. You can start by setting a very low quantity, such as 1 and call it Low. Set the high quantity (call it High) by using the reverse half interval method explained in this chapter.

You can then approximate the EOQ by finding the half point of the Low and High; that is, Q = (High + Low)/2); then compute the total carrying costs and the total ordering costs. If the total carrying costs are higher than the total ordering costs, Q is too high (higher than the EOQ according to the previous explanation). You should then adjust the high bound by setting High = Q; otherwise, you should adjust the low bound. Repeat this computational process until High and Low are very close to each other; for example, a difference less than 0.01. There you have the solution. (This method of arriving at a numerical solution is recognized as the half interval method as explained in this chapter.)
Write a routine to compute the EOQ using this method. The visual interface should be the same as the preceding exercise. Compare your result with what you obtained from 7-32. Also, compare the time required between the two algorithms.

7-34. Listing All Available Lottery Numbers. Assume that a lottery involves the random selection (without replacement) of three balls from a box. The box contains 25 numbers labeled from 1 to 25.

Write a routine that will list all possible combinations of numbers for the lottery. Place the results in a list box.

7-35. Clearing Selections of a List Box. Write a Sub procedure that will take a list box as its parameter and clear all selections in the list box. (*Note:* Just deselect; do not remove the item[s].) Note that if the SelectionMode is set to 1, all you have to do is to set the SelectedIndex to −1; otherwise, you will also need to deselect all selected items.

7-36. Clearing All Controls. Write a Sub procedure that will clear all controls in a form that are used to accept the user input. These controls should include text boxes, masked text boxes, radio buttons, check boxes, combo boxes, and list boxes. Deselect items selected in combo boxes and list boxes, but do not clear the items from these controls. Refer to exercise 7-35 for the fine points concerning deselecting items in the list box.

7-37. Generalizing the Simulation Example. Modify the blackjack simulation program in example 10 in this chapter so that it can be used for a general class of problems. Your project should allow the user to specify the following:

- The probability of success per trial (winning a hand)
- The minimum desired number of successes per 100 trials, such as winning at least 60 out of 100 hands
- The number of experiment—this was assumed to be 1,000 in the example

Test your project by answering the following question: "Assume the probability that Mark McGwire hits a home run each time that he is at bat is 10%. He is at bat five times per game. What is the probability that he hits 70 home runs in a season of 162 games?"

Without revising the program, can you use it to answer the question, "For Mark McGwire to have a 50% probability to hit 70 home runs in a season, at what rate (probability) must he be able to hit a home run each time he is at bat?" (*Hint:* Try different values of the probability to win per trial until the probability of success is around 50%. You can minimize your trials by mimicking the half interval method.)

Projects

7-38. Finding the Effective Rate of a Bond. Refer to exercise 7-29 for the computation of the bond price (present value). In the financial market, the bond price is found through the bid and ask process (quote). When you are buying or selling a bond, you know the price first. As an investor, you, of course, would want to know the effective (market) interest rate of the bond given the price. You are to develop a project for such a purpose.

Input: The user will enter the coupon rate, the years to maturity, and the price of the bond. The price is quoted as if the face value were 100, so a quote of 101 actually means 101% of the face value.

Output: The computed market rate should be displayed in a text box with its ReadOnly property set to True.

Computation: The bond price changes inversely with the market rate of interest. The higher the market rate, the lower the bond price. You should first write a function to compute the bond price (present value) given the market rate, coupon rate, and years to maturity as the parameters (call this function BondPrice) (refer to the formula in exercise 7-29). The rate that makes the present value of the bond equal to the current bond price is the market (effective) rate on the bond. You can approximate the market rate by using the half interval method explained in this chapter.

7-39. Updating the Accounts Receivable File: Sequential File Processing. Develop a project that will update customer balances in an accounts receivable master file using a customer transaction file. Each record in the accounts receivable master file consists of two data fields and are structured as follows:

Field	Length	Description
Account number	5 digits	xxxxx
Account balance	10 digits	xxxxxxx.xx; begins at position 7

Sample records of the file appear as follows:

```
10001 8000.00
10002 10000.00
.
.
```

Each record in the transaction file consists of four data fields and is structured as follows:

Field	Length	Description
Date	10 characters	mm/dd/yyyy
Account number	five digits xxxxx;	begins at position 12
Charge or credit	one character	C or D; in position 18; C = Credit; D = Charge
Amount	nine digits	xxxxxx.xx; begins at position 20

Sample records of the file appear as follows:

```
01/10/1999 10002 D 7000.00
01/05/1999 10002 C 5000.00
.
.
```

Both files are text files created with the StreamWriter.WriteLine method (one record per line), and have been sorted in ascending order by the account number.

To update the account balances, your program needs to read both files and use the data to create a new account receivable (customer balance) file. Note that the customer master file contains all valid accounts. An account number in the transaction file that fails to match one in the master file represents an invalid number. The transaction record should be logged in an error log file and discarded (not processed).

Hints:

- You need to use four files for the duration of your program. These are the existing account receivable master file (for Input), the transaction file (for Input), the updated account receivable file (for Output), and the error log file. Note that the new master file should have the same format as the existing one.
- Use a button’s Click event to initiate the update process.
- The major steps involved in the update are as follows:
    1. Read transaction records for an account (TFAcctNo) until a different account or end of file (EOF) is encountered. For each record for the same account, accumulate the transaction amount. Note that you have an unprocessed transaction record here.
    2. Read a customer master record. If the account number (MFAcctNo) is smaller than the one from the transaction file (TFAcctNo), this master record has no current transactions. Simply output the record onto the new account receivable master file and repeat step 2. If MFAcctNo equals TFAcctNo, update the balance with the amount accumulated from step 1 and output the result onto the new master file. If MFAcctNo is greater than TFAcctNo, TFAcctNo is invalid; print an error log (message) in the error log file. Proceed to 3.

3. If it is not the EOF of the transaction file, set the TFAcctNo and the accumulated transaction amount to the last record read in step 1 and repeat steps 1 and 2; otherwise, copy all the remaining records in the old master file to the new master file and then quit.