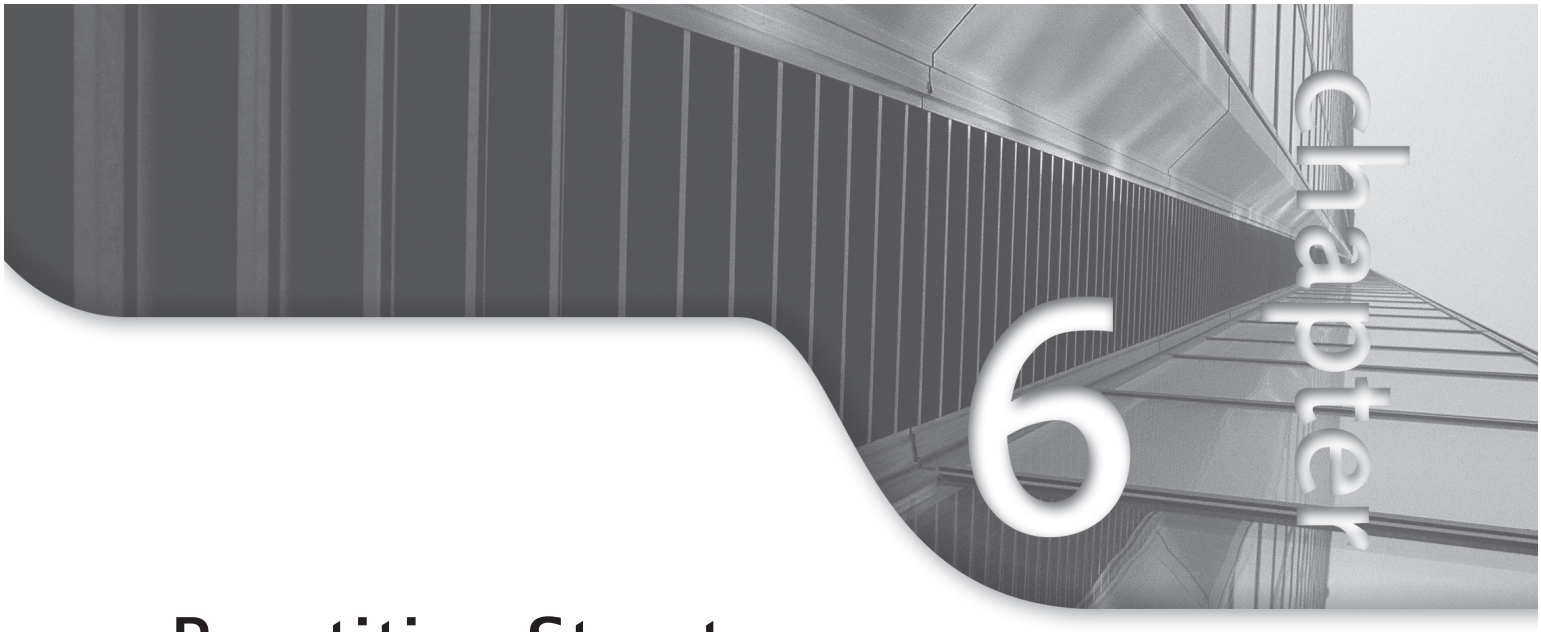


Repetition Structures

Goals

The applications examined so far have illustrated the programming concepts involved in input, output, assignment, and selection capabilities. By this time you should have gained enough experience to be comfortable with these concepts and the mechanics of implementing them using Visual Basic. However, many problems require a repetition capability, in which the same calculation or sequence of instructions is repeated, over and over, using different sets of data. Examples of such repetition include continual checking of user data entries until an acceptable entry, such as a valid password, is made; counting and accumulating running totals; and recurring acceptance of input data and recalculation of output values that only stop upon entry of a designated value.

This chapter explores the different methods that programmers use to construct repeating sections of code and how they can be implemented in Visual Basic. A repeated procedural section of code is commonly called a loop, because after the last statement in the code is executed, the program branches, or loops back to the first statement and starts another repetition. Each repetition is also referred to as an iteration or pass through the loop.

6.1 Introduction

The real power of most computer programs resides in their ability to repeat the same calculation or sequence of instructions many times over, each time using different data, without the necessity of rerunning the program for each new set of data values. This ability is realized through repetitive sections of code. Such repetitive sections are written only once, but include a means of defining how many times the code should be executed.

Constructing repetitive sections of code requires four elements:

1. A *repetition statement* that both defines the boundaries containing the repeating section of code and controls whether the code will be executed or not. There are three different forms of Visual Basic repetition structures: **Do While** structures, **For** structures, and **Do/Loop Until** structures.
2. A *condition* that needs to be evaluated. Valid conditions are identical to those used in selection statements. If the condition is **True**, the code is executed; if it is **False**, the code is not executed.
3. A *statement* that initially *sets the condition*. This statement must always be placed before the condition is first evaluated, to ensure correct loop execution the first time.
4. A *statement* within the repeating section of code that *allows the condition to become* ***False***. This is necessary to ensure that, at some point, the repetitions stop.

Pretest and Posttest Loops

The condition being tested can be evaluated either at the beginning or the end of the repeating section of code. Figure 6–1 illustrates the case where the test occurs at the beginning of the loop. This type of loop is referred to as a *pretest loop*, because the condition is tested before any statements within the loop are executed. If the condition is

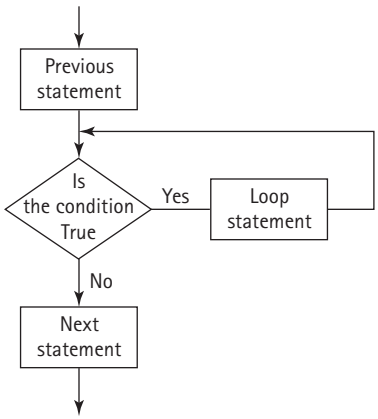


Figure 6–1 *A Pretest Loop*

True, the executable statements within the loop are executed. If the initial value of the condition is **False**, the executable statements within the loop are never executed, and control transfers to the first statement after the loop. To avoid infinite repetitions, the condition must be updated within the loop. Pretest loops are also referred to as *entrance-controlled loops*. Both the `Do While` and `For` loop structures are examples of such loops.

A loop that evaluates a condition at the end of the repeating section of code, as illustrated in Figure 6–2, is referred to as a *posttest* or *exit-controlled loop*. Such loops always execute the loop statements at least once before the condition is tested. Because

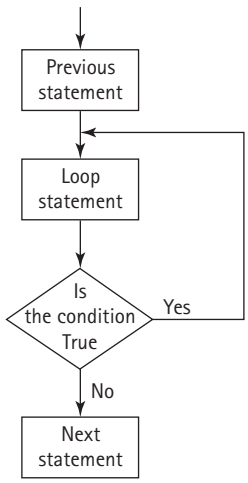


Figure 6–2 *A Posttest Loop*

the executable statements within the loop are continually executed until the condition becomes **False**, there must always be a statement within the loop that updates the condition and permits it to become **False**. The **Do /Loop Until** construct is an example of a posttest loop.

Fixed Count Versus Variable Condition Loops

In addition to where the condition is tested (pretest or posttest), repeating sections of code are also classified as to the type of condition being tested. In a *fixed-count loop*, the condition is used to keep track of how many repetitions have occurred. For example, we might want to produce a very simple fixed design such as:

```
**************************
**************************
**************************
**************************
```

In each of these cases, a fixed number of calculations is performed or a fixed number of lines is printed, at which point the repeating section of code is exited. All of Visual Basics repetition statements can be used to produce fixed-count loops.

In many situations, the exact number of repetitions is not known in advance or the items are too numerous to count beforehand. For example, if we are working with a large amount of market research data, we might not want to take the time to count the number of actual data items that must be entered and so we would use a variable-condition loop. In a *variable-condition loop*, the tested condition does not depend on a count being achieved, but rather on a variable that can change interactively with each pass through the loop. When a specified value is encountered, regardless of how many iterations have occurred, repetitions stop. All of Visual Basic's repetition statements can be used to create variable-condition loops. In this chapter we will encounter examples of both fixed-count and variable-condition loops.

6.2 Do While Loops

In Visual Basic, a `Do While` loop is constructed using the following syntax:

```
Do While expression
    statement(s)
Loop
```

The expression contained after the keywords `Do While` is the condition tested to determine if the statements provided before the `Loop` statement are executed. The expression is evaluated in exactly the same manner as that contained in an **If-Else** statement—the difference is in how the expression is used. As we have seen, when the expression is **True** in an **If-Else** statement, the statement or statements following the

expression are executed once. In a `Do While` loop, the statement or statements following the expression are executed repeatedly as long as the expression remains **True**. Considering the expression and the statements following it, the process used by the computer in evaluating a `Do While` loop is:

1. Test the expression.
2. If the expression is **True**:
   a. Execute all statements following the expression up to the `Loop` statement.
   b. Go back to step 1.

   Else

   Exit the `Do While` statement and execute the next executable statement following the `Loop` statement.

Note that step 2b forces program control to be transferred back to step 1. This transfer of control back to the start of a `Do While` statement, in order to reevaluate the expression, is what forms the loop. The `Do While` statement literally loops back on itself to recheck the expression until it becomes **False**. This means that somewhere in the loop, it must be possible to alter the value of the tested expression so that the loop ultimately terminates its execution.

This looping process produced by a `Do While` statement is illustrated in Figure 6–3. A diamond shape is used to show the two entry and two exit points required in the decision part of the `Do While` statement.

To make this process more understandable, consider the code segment below. Recall that the statement `lstDisplay.Items.Add(count)` will display each value of count into a List Box.

```
Do While count <= 10
  lstDisplay.Items.Add(count)
Loop
```

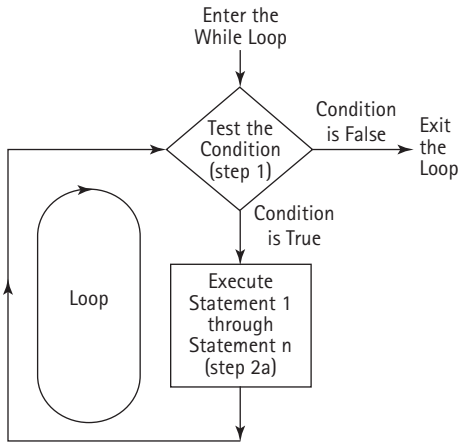


Figure 6–3 *Anatomy of a Do While Loop*

Although this loop structure is valid, the alert reader will realize that we have created a situation in which the `lstDisplay` statement either is called forever (or until we stop the program) or it is not called at all. Let us examine why this happens.

If `count` has a value less than or equal to 10 when the expression is first evaluated, the `lstDisplay` statement is executed. When the `Loop` statement is encountered, the structure automatically loops back on itself and retests the expression. Because we have not changed the value stored in `count`, the expression is still true and the `lstDisplay` statement is again executed. This process continues forever, or until the program containing this statement is prematurely stopped by the user. However, if count starts with a value greater than 10, the expression is false to begin with and the `lstDisplay` statement is never executed.

How do we set an initial value in `count` to control what the `Do While` statement does the first time the expression is evaluated? The answer is to assign values to each variable in the tested expression before the `Do While` statement is encountered. For example, the following sequence of instructions is valid:

```
Dim count As Integer
count = 1
Do While count <= 10
  lstDisplay.Items.Add(count)
Loop
```

Using this sequence of instructions, we ensure that `count` starts with a value of 1. We could assign any value to `count` in the assignment statement—the important thing is to assign a value. In practice, the assigned value depends upon the application.

We must still change the value of `count` so that we can finally exit the `Do While` loop. This requires an expression such as `count = count + 1` to increment the value of `count` each time the `Do While` statement is executed. All that we have to do is add a statement within the `Loop` structure that modifies count's value so that the loop ultimately terminates. For example, consider the following expanded loop:

```
count = 1   ' initialize count
Do While count <= 10
  lstDisplay.Items.Add(count)
  count = count +  1  ' increment count
Loop
```

Note that the `Do While` structure begins with the keywords `Do While` and ends with the keyword `Loop`. For this particular loop, we have included two statements within the `Loop` structure:

```
lstDisplay.Items.Add(count)
count = count +  1  ' increment count
```

Now we need to analyze the complete section of code to understand how it operates. The first assignment statement sets `count` equal to 1. The `Do While` structure is

then entered and the expression is evaluated for the first time. Because the value of `count` is less than or equal to 10, the expression is **True** and the two statements internal to the **Loop** structure are executed. The first statement within the loop is a `lstDisplay` statement that displays the value of `count`. The next statement adds 1 to the value currently stored in `count`, making this value equal to 2. The `Do While` statement now loops back to retest the expression. Since `count` is still less than or equal to 10, the `loop` statements are executed once again. This process continues until the value of `count` reaches 11. Event Procedure 6–1 illustrates these statements within the context of a complete **Button1_Click** event procedure:

Event Procedure 6–1

```
Private Sub frmMain_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles MyBase.Click
  Dim count As Integer

  count = 1   ' initialize count
  Do While count <= 10
    lstDisplay.Items.Add(count)
    count = count + 1  ' increment count
  Loop
End Sub
```

The output that will appear on the screen in the List Box when Event Procedure 6–1 is activated is:

```
1
2
3
4
5
6
7
8
9
10
```

There is nothing special about the name `count` used in Event Procedure 6–1. Any valid integer variable could have been used.

Note that if a text box had been used instead of a list box, the `lstDisplay.Items.Add(count)` statement would be replaced with `txtDisplay.AppendText(' ' & count)` and the output would be:

```
1 2 3 4 5 6 7 8 9 10
```

Before we consider other examples of the `Do While` statement, two comments concerning Event Procedure 6–1 are in order. First, the statement `count + 1` can be replaced with any statement that increases the value of `count`. A statement such as `count = count + 2`, for example, would cause every second integer to be displayed. Second, it is the programmer's responsibility to ensure that count is changed in a way that ultimately leads to a normal exit from the `Do While` statement. For example, if we replace the expression `count + 1` with the expression `count - 1`, the value of count will never exceed 10 and an infinite loop will be created. An *infinite loop* is a loop that never ends. With the above example, the computer continues to display numbers, until you realize that the program is not working as you expected.

Now that you have some familiarity with the `Do While` structure, see if you can read and determine the output of Event Procedure 6–2.

Event Procedure 6–2

```
Private Sub frmMain_Click(ByVal sender As Object, ByVal e As _
    System.EventArgs) Handles MyBase.Click
  Dim I As Integer

  I = 10    ' initialize I
  Do While I >= 1
    lstDisplay.Items.Add(I)
    I = I - 1        ' subtract 1 from I
  Loop
End Sub
```

The assignment statement in Event Procedure 6–2 initially sets the integer variable `I` to 10. The `Do While`statement then checks to see if the value of `I` is greater than or equal to 1. While the expression is **True**, the value of `I` is displayed by the `lstDisplay` statement and the value of `I` is decremented by 1. When `I` finally reaches zero, the expression is **False** and the program exits the `Do While` statement. Thus, the following display is obtained when Event Procedure 6–2 is activated:

```
10
9
8
7
6
5
4
3
2
1
```

To illustrate the power of the `Do While` loop, consider the task of printing a table of numbers from 1 to 10 with their squares and cubes. This can be done with a simple `Do While` statement, as illustrated by Event Procedure 6–3.

Event Procedure 6–3

```
Private Sub frmMain_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles MyBase.Click
  Dim num As Integer

  num = 1
  lstDisplay.Items.Clear()  ' clear the List box
  lstDisplay.Items.Add("NUMBER SQUARE CUBE")
  Do While num < 11
    lstDisplay.Items.Add("      " & num & "              " & (num ^ 2) _
      & "         " & (num ^ 3))
    num = num + 1
  Loop
End Sub
```

When Event Code 6–3 is activated, the following display is produced in the List box:

```
NUMBER   SQUARE   CUBE
   1        1       1
   2        4       8
   3        9       27
   4        16       64
   5        25       125
   6        36       216
   7        49       343
   8        64       512
   9        81       729
   10        100       1000
```

Note that the expression used in Event Procedure 6–3 is `num < 11`. For the integer variable `num`, this expression is exactly equivalent to the expression `num <= 10`. The choice of which to use is entirely up to you.

If we want to use Event Procedure 6–3 to produce a table of 1,000 numbers, all that needs to be done is to change the expression in the `Do While` statement from `num < 11` to `num < 1001`. Changing the 11 to 1001 produces a table of 1,000 lines—not bad for a simple five-line `Do While` structure.

All the program examples illustrating the `Do While` statement are examples of fixed-count loops because the tested condition is a counter that checks for a fixed number of repetitions. A variation on the fixed-count loop can be made, where the counter is not incremented by one each time through the loop, but by some other value. For example, consider the task of producing a Celsius to Fahrenheit temperature conversion table. Assume that Fahrenheit temperatures corresponding to Celsius temperatures rang-

ing from 5 to 50 degrees are to be displayed in increments of five degrees. The desired display can be obtained with the following series of statements:

```
celsius = 5       ' starting Celsius value
Do While celsius <= 50
  fahren =  9.0/5.0  * celsius + 32.0
  lstDisplay.Items.Add(celsius & " " & fahren)
  celsius = celsius + 5
Loop
```

As before, the Do While loop consists of everything from the words Do While through the `Loop` statement. Prior to entering the Do While loop, we have made sure to assign a value to the counter being evaluated and there is a statement to alter the value of `celsius` within the loop (in increments of 5), to ensure an exit from the Do While loop. Event Procedure 6–4 illustrates the use of this code within the context of a complete `Form_Click` event.

Event Procedure 6–4

```
' A procedure to convert Celsius to Fahrenheit

Private Sub frmMain_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles MyBase.Click
  Const MAX_CELSIUS As Integer = 50
  Const START_VAL As Integer = 5
  Const STEP_SIZE As Integer = 5

  Dim celsius As Integer
  Dim fahren As Single

  lstDisplay.Items.Clear()
  lstDisplay.Items.Add("Degrees" & "     Degrees")
  lstDisplay.Items.Add("Celsius" & "     Fahrenheit")

  celsius = START_VAL
  Do While celsius <= MAX_CELSIUS
    fahren = (9.0 / 5.0) * celsius + 32.0
    lstDisplay.Items.Add("   " & celsius & "              " & fahren)
    celsius = celsius + STEP_SIZE
  Loop

End Sub
```

After this code is activated, the list box will display:

```
Degrees     Degrees
Celsius     Fahrenheit
 5            41
 10           50
 15           59
 20           68
 25           77
 30           86
 35           95
 40           104
 45           113
 50           122
```

Exercises 6.2

1. Rewrite Event Procedure 6–1 to print the numbers 2 to 10 in increments of two. The list box in your program should display the following:

   2

   4

   6

   8

   10

2. Rewrite Event Procedure 6–4 to produce a table that starts at a Celsius value of −10 and ends with a Celsius value of 60, in increments of ten degrees.

3. a. Using the following code, determine the total number of items displayed. Also determine the first and last numbers printed.

```
Dim num As Integer
Num = 0
 Do While Num <= 20
    Num = Num + 1
    lstDisplay.Items.Add(Num)
 Loop
```

   b. Enter and run the code from Exercise 3a as a `Button_Click` event procedure to verify your answers to the exercise.

   c. How would the output be affected if the two statements within the compound statement were reversed; i.e., if the `lstDisplay` statement were made before the n = n + 1 statement?

4. Write a Visual Basic program that converts gallons to liters. The program should display gallons from 10 to 20 in one-gallon increments and the corresponding liter equivalents. Use the relationship of 3.785 liters to a gallon.
5. Write a Visual Basic program to produce the following display within a multiline text box.

```
0
 1
  2
   3
    4
     5
      6
       7
        8
         9
```

6. Write a Visual Basic program to produce the following displays within a list box.

a.
```
****
 ****
  ****
   ****
```

b.
```
 ****
  ****
 ****
****
```

7. Write a Visual Basic program that converts feet to meters. The program should display feet from 3 to 30 in three-foot increments and the corresponding meter equivalents. Use the relationship of 3.28 feet to a meter.
8. A machine purchased for $28,000 is depreciated at a rate of $4,000 a year for seven years. Write and run a Visual Basic program that computes and displays in a suitably sized list box a depreciation table for seven years. The table should have the following form:

Year	Depreciation	End-of-year value	Accumulated depreciation
1	4000	24000	4000
2	4000	20000	8000
3	4000	16000	12000
4	4000	12000	16000
5	4000	8000	20000
6	4000	4000	24000
7	4000	0	28000

9. An automobile travels at an average speed of 55 miles per hour for four hours. Write a Visual Basic program that displays in a list box the distance driven, in miles, that the car has traveled after 0.5, 1.0, 1.5 hours, and so on. until the end of the trip.

10. An approximate conversion formula for converting temperatures from Fahrenheit to Celsius is:

    Celsius = (Fahrenheit − 30) / 2

    a. Using this formula, and starting with a Fahrenheit temperature of zero degrees, write a Visual Basic program that determines when the approximate equivalent Celsius temperature differs from the exact equivalent value by more than four degrees. (*Hint:* Use a `Do While` loop that terminates when the difference between approximate and exact Celsius equivalents exceeds four degrees.)

    b. Using the approximate Celsius conversion formula given in Exercise 10a, write a Visual Basic program that produces a table of Fahrenheit temperatures, exact Celsius equivalent temperatures, approximate Celsius equivalent temperatures, and the difference between the exact and approximate equivalent Celsius values. The table should begin at zero degrees Fahrenheit, use two-degree Fahrenheit increments, and terminate when the difference between exact and approximate values differs by more than four degrees. Use a list box to display these values.

6.3 Interactive Do While Loops

Combining interactive data entry with the repetition capabilities of the `Do While` loop produces very adaptable and powerful programs. To understand the concept involved, consider Program 6–1, where a `Do While` statement is used to accept and then display four user-entered numbers, one at a time. Although it uses a very simple idea, the program highlights the flow of control concepts needed to produce more useful programs. Figure 6–4 shows the interface for Program 6–1.

For this application the only procedure code is the **Click** event, which is listed in Program 6–1's event code.

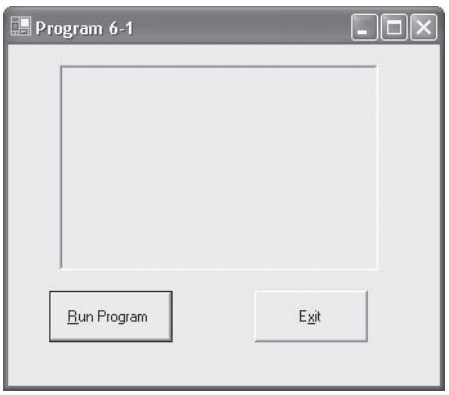


Figure 6–4 *Program 6–1's Interface*

Table 6–1 The Properties Table for Program 6–1

Object	Property	Setting
Form	Name	frmMain
	Text	Program 6–1
ListBox	Name	lstDisplay
Button	Name	btnRun
	Text	&Run Program
Button	Name	btnExit
	Text	E&xit

Program 6–1's Event Code

```
Private Sub btnRun_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) _Handles btnRun.Click
  Const MAXNUMS As Integer = 4
  Dim count As Integer
  Dim num As Single

  lstDisplay.Items.Clear()
  lstDisplay.Items.Add("This Program will ask you to enter " & MAXNUMS _
      & " numbers")

  count = 1
  Do While count <= MAXNUMS
    num = Val(InputBox("Enter a number", "Input Dialog", 0))
    lstDisplay.Items.Add("The number entered is " & num)
    count = count + 1
  Loop
End Sub

Private Sub btnExit_Click(ByVal sender As Object, ByVal e As _
    System.EventArgs) Handles btnExit.Click
  Beep()
  End
End Sub
```

Figure 6–5 illustrates a sample run of Program 6–1 after four numbers have been input. The `InputBox` control that is displayed by the program for the data entry is shown in Figure 6–6.

Let us review the program to clearly understand how the output illustrated in Figure 6–5 was produced. The first message displayed is caused by execution of the first

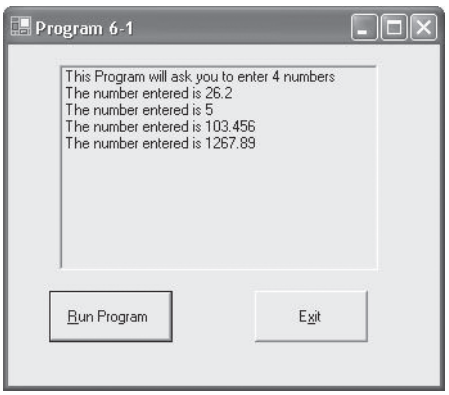


Figure 6–5 *A Sample Run of Program 6–1*

Figure 6–6 *The InputBox Displayed by Program 6–1*

`lstDisplay` statement. This statement is outside and before the `Do While` loop, so it is executed once before any statement within the loop.

Once the `Do While` loop is entered, the statements within the loop are executed while the test condition is **True**. The first time through the loop, the following statement is executed:

```
num = Val(InputBox("Enter a number", "Input Dialog", "0"))
```

The call to the `InputBox` function displays the `InputBox` shown in Figure 6–6, which forces the computer to wait for a number to be entered at the keyboard. Once a number is typed and the Enter key is pressed, the **lstDisplay** statement within the loop displays on a new line the number that was entered. The variable `count` is then incremented by one. This process continues until four passes through the loop have been made and the value of `count` is 5. Each pass causes the `InputBox` to be displayed with the message `The number entered is`. Figure 6–7 illustrates this flow of control.

Program 6–1 can be modified to use the entered data rather than simply displaying it. For example, let us add the numbers entered and display the total. To do this we must be very careful about how we add the numbers, because the same variable, `num`, is used

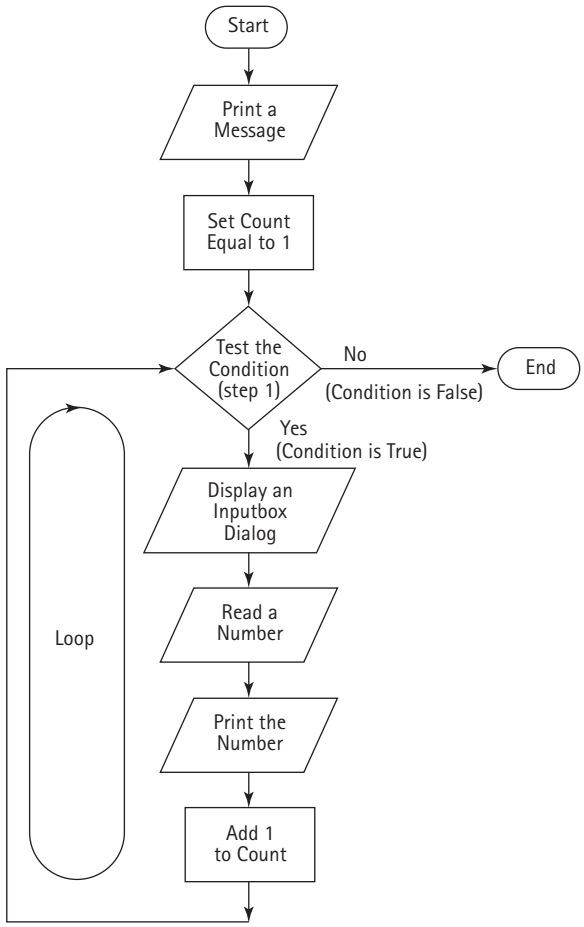


Figure 6–7 *Flow of Control Diagram for the btnRun Click Event Procedure*

for each number entered. As a result, the entry of a new number in Program 6–1 automatically causes the previous number stored in `num` to be lost. Thus, each number entered must be added to the total before another number is entered. The required sequence is:

```
Enter a number
Add the number to the total
```

How do we add a single number to a total? A statement such as `total = total + num` is the solution. This is the accumulating statement introduced in Section 3.3. After each number is entered, the accumulating statement adds the number to the total, as

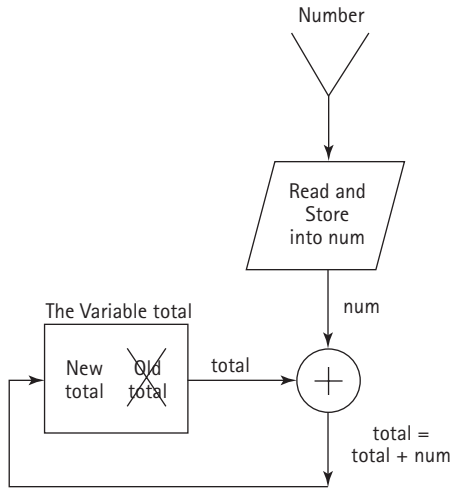


Figure 6–8 *Accepting and Adding a Number to a Total*

illustrated in Figure 6–8. The complete flow of control required for adding the numbers is illustrated in Figure 6–9.

Observe that in Figure 6–9 we have made a provision for initially setting the total to zero before the `Do While` loop is entered. If we cleared the total inside the `Do While` loop, it would be set to zero each time the loop was executed and any value previously stored would be erased. As indicated in the flow diagram shown in Figure 6–9, the statement `total = total + num` is placed immediately after the call to the **InputBox** function. Putting the accumulating statement at this point in the program ensures that the entered number is immediately added to the total.

Program 6–2 incorporates the necessary modifications to Program 6–1 to total the numbers entered. The significant difference between Program 6–1 and Program 6–2 is the `Run Program` button event code. The event code for Program 6–2 is shown below.

Program 6–2's Event Code

```
Private Sub btnRun_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles btnRun.Click
  Const MAXNUMS As Integer = 4
  Dim count As Integer
  Dim num, total As Single

  lstDisplay.Items.Clear()
  lstDisplay.Items.Add("This Program will ask you to enter " & MAXNUMS & _
      " numbers")
  count = 1
  Do While count <= MAXNUMS
    num = Val(InputBox("Enter a number", "Input Dialog", 0))
```

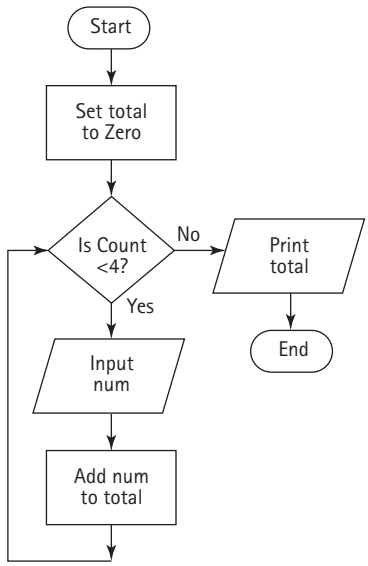


Figure 6–9 *Accumulation Flow of Control*

```
    total = total + num
    lstDisplay.Items.Add("The number entered is " & num)
    lstDisplay.Items.Add("The total is now " & total)
    count = count + 1
  Loop
  lstDisplay.Items.Add("The final total is " & total)

End Sub
```

Let us review this event code. The variable `total` was created to store the total of the numbers entered. Prior to entering the `Do While` statement, the value of `total` is set to zero. This ensures that any previous value present in the storage locations assigned to the variable `total` is erased. Within the `Do While` loop, the statement `total = total + num` is used to add the value of the entered number into `total`. As each value is entered, it is added into the existing `total` to create a new `total`. Thus, `total` becomes a running subtotal of all the values entered. Only when all numbers are entered does `total` contain the final sum of all the numbers. After the `Do While` loop is finished, the last `lstDisplay` statement displays the final sum.

Using the same data we entered in the sample run for Program 6–1, the sample run of Program 6–2 produces the total shown in Figure 6–10.

Having used an accumulating assignment statement to add the numbers entered, we can now go further and calculate the average of the numbers. However, first we have to determine whether we calculate the average within the `Do While` loop or outside of it.

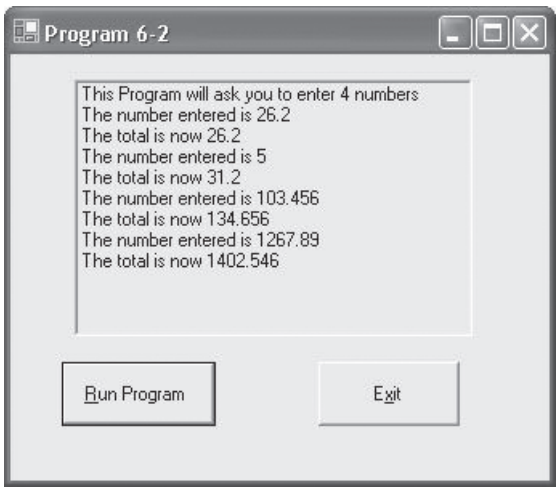


Figure 6–10 *A Sample Run of Program 6–2*

In the case at hand, calculating an average requires that both a final sum and the number of items in that sum be available. The average is then computed by dividing the final sum by the number of items. We must now answer the questions "At what point in the program is the correct sum available, and at what point is the number of items available?" In reviewing Program 6–2's event code, we see that the correct sum needed for calculating the average is available after the `Do While` loop is finished. In fact, the whole purpose of the `Do While` loop is to ensure that the numbers are entered and added correctly to produce a correct sum. With this as background, see if you can read and understand the event code used in Program 6–3.

Program 6–3's Event Code

```
Private Sub btnRun_Click(ByVal sender As Object, _
  ByVal e As System.EventArgs) Handles btnRun.Click
Const MAXNUMS As Integer = 4
Dim count As Integer
Dim num, total, average As Single

lstDisplay.Items.Clear()
lstDisplay.Items.Add("This Program will ask you to enter " & MAXNUMS & _
    " numbers")
count = 1
Do While count <= MAXNUMS
  num = Val(InputBox("Enter a number", "Input Dialog", 0))
  total = total + num
  lstDisplay.Items.Add("The number entered is " & num)
  count = count + 1
```

```
 Loop
 Average = total/MAXNUMS
 lstDisplay.Items.Add("The average of these numbers is " & average)
End Sub
```

Program 6–3 is almost identical to Program 6–2, except for the calculation of the average. We have also removed the constant display of the total within and after the `Do While` loop. The loop in Program 6–3 is used to enter and add four numbers. Immediately after the loop is exited, the average is computed and displayed.

A sample run of Program 6–3 is illustrated in Figure 6–11.

The Do Until Loop Structure

In a `Do While Loop` structure, the statements within the loop are executed as long as the condition is **True**. A variation of the `Do While` loop is the `Do Until` loop which executes the statements within the loop as long as the condition is **False**. The syntax of a `Do Until` loop is:

```
Do Until condition
  statement(s)
Loop
```

The `Do Until` loop, like its `Do While` counterpart, is an entrance-controlled loop. Unlike the `Do While` loop, which executes until the condition becomes **False**, a `Do Until` loop executes until the condition becomes **True**. If the condition tested in a `Do Until` loop is **True** to begin with, the statements within the loop will not execute at all. `Do Until` loops are not used extensively for two reasons. First, most practical programming problems require performing a repetitive set of tasks while a condition is **True**. Second, if an entrance-controlled loop is required until a condition becomes **True**, it can always be

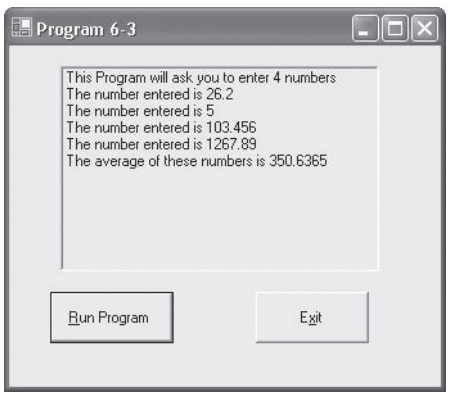


Figure 6–11 *A Sample Run of Program 6–3*

restated as a loop that needs to be executed while the same condition remains **False**, using the following syntax:

```
Do While Not condition
  statement(s)
Loop
```

Sentinels

All of the loops we have created thus far have been examples of fixed-count loops, where a counter controls the number of loop iterations. By means of a `Do While` statement, variable-condition loops may also be constructed. For example, when entering grades we may not want to count the number of grades that will be entered, preferring to enter the grades one after another and, when finished, typing in a special data value to signal the end of data input.

In computer programming, data values used to signal either the start or end of a data series are called *sentinels*. The sentinel values selected must not conflict with legitimate data values. For example, if we were constructing a program to process a student's grades, assuming that no extra credit is given that could produce a grade higher than 100, we could use any grade higher than 100 as a sentinel value. Program 6–4 illustrates this concept. In Program 6–4's event procedure, the grades are serially requested and accepted until a number larger than 100 is entered. Entry of a number higher than 100 alerts the program to exit the `Do While` loop and display the sum of the numbers entered. Figure 6–12 shows the interface for Program 6–4.

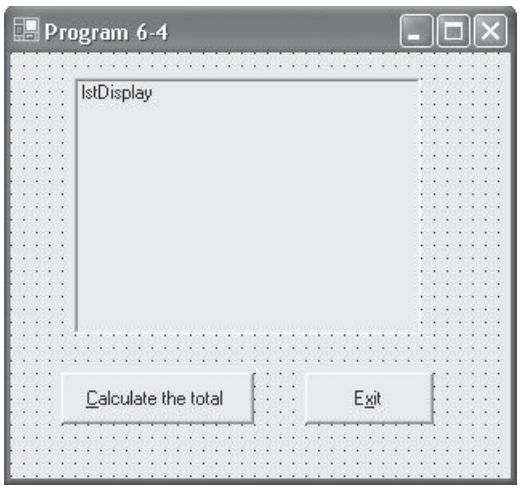


Figure 6–12 *Program 6–4's Interface*

Table 6–2 The Properties Table for Program 6–4

Object	Property	Setting
Form	Name	frmMain
	Text	Program 6–4
ListBox	Name	lstDisplay
	TabStop	False
Button	Name	btnRun
	Text	&Calculate the total
Button	Name	btnExit
	Text	E&xit

Program 6–4's Event Code

```
Private Sub btnRun_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles btnRun.Click
  Const HIGHGRADE As Integer = 100
  Dim grade, total As Single

  grade = 0
  total = 0
  lstDisplay.Items.Clear()
  lstDisplay.Items.Add("To stop entering grades, type in")
  lstDisplay.Items.Add("any number greater than 100.")

  Do While grade <= HIGHGRADE
    grade = Val(InputBox("Enter a grade", "Input Dialog", "0"))
    lstDisplay.Items.Add("The grade just entered is " & grade)
    total = total + grade
  Loop
  lstDisplay.Items.Add("The total of the valid grades is " & total - grade)
End Sub

Private Sub btnExit_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles btnExit.Click
  Beep()
  End
End Sub
```

A sample run using Program 6–4 is illustrated in Figure 6–13. As long as grades less than or equal to 100 are entered, the program continues to request and accept additional data. For example, when a number less than or equal to 100 is entered, the pro-

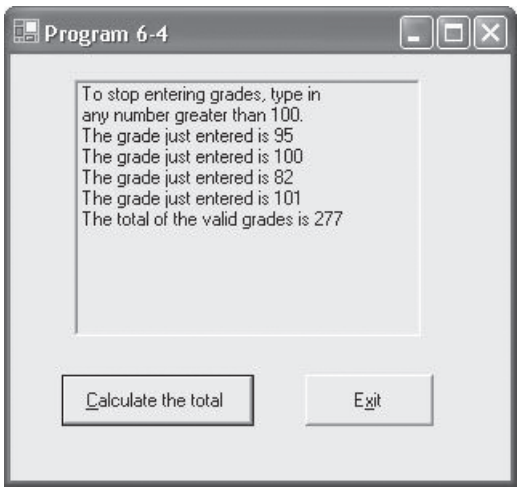


Figure 6–13 *A Sample Run Using Program 6–4*

gram adds this number to the total. When a number greater than 100 is entered, the program exits the loop and displays the sum of the grades.

Note that the event procedure used in Program 6–4 differs from previous examples in that termination of the loop is controlled by an externally supplied value rather than a fixed count condition. The loop in Program 6–4 will continue indefinitely until the sentinel value is encountered.

Breaking Out of a Loop

It is sometimes necessary to prematurely break out of a loop when an unusual error condition is detected. The **Exit Do** statement allows you to do this. For example, execution of the following `Do While` loop is immediately terminated if a number greater than 76 is entered.

```
count = 1
Do While count <= 10
  num = Val(InputBox("Enter a number", "Input Dialog", "0"))
  If num > 76 Then
    lstDisplay.Items.Add("You lose!")
    Exit Do      ' break out of the loop
  End If
  lstDisplay.Items.Add("Keep on trucking!")
  count = count + 1
Loop
' break jumps to here
```

The `Exit Do` statement violates pure structured programming principles because it provides a second, nonstandard exit from a loop. This statement should be used with extreme caution since it makes debugging more difficult.

Exercises 6.3

1. Write a `Do While` loop to achieve the following:
   a. Display the multiples of 3 backward from 33 to 3, inclusive.
   b. Display the capital letters of the alphabet backward from Z to A.
2. Rewrite Program 6–2 to compute the average of eight numbers.
3. Rewrite Program 6–2 to display the following prompt in an InputBox:

   ```
   Please type the total number of data values to be averaged
   ```

   In response to this prompt, the program should accept a user-entered number from the `InputBox` and then use this number to control the number of times the **Do While** loop is executed. Thus, if the user enters 5 in the `InputBox`, the program should request the input of five numbers, and display the total after five numbers have been entered.
4. a. Write a Visual Basic program to convert Celsius degrees to Fahrenheit. The program should request the starting Celsius value, the number of conversions to be made, and the increment between Celsius values. The display should have appropriate headings and list the Celsius value and the corresponding Fahrenheit value. Use the relationship: Fahrenheit = 9.0 / 5.0 * Celsius + 32.0.
   b. Run the program written in Exercise 4a on a computer. Verify that your program starts at the correct Celsius value and contains the exact number of conversions specified in your input data.
5. a. Modify the program written in Exercise 4b, to request the starting Celsius value, the ending Celsius value, and the increment. Instead of the condition checking for a fixed count, the condition will check for the ending Celsius value.
   b. Run the program written in Exercise 5a on a computer. Verify that your output starts and ends at the correct values.
6. Rewrite Program 6–3 to compute the average of ten numbers.
7. Rewrite Program 6–3 to display the following prompt in an InputBox:

   ```
   Please type the total number of data values to be averaged
   ```

   In response to this prompt, the program should accept a user-entered number from the `InputBox` and then use this number to control the number of times the **Do While** loop is executed. Thus, if the user enters 6, the program should request the input of six numbers and display the average of the next six numbers entered.
8. By mistake, a programmer put the statement `average = total / count` within the `Do While` loop immediately after the statement `total = total + num` in Program 6–3. Thus, the `Do While` loop becomes:

   ```
   count = 1
   ```

```
total = 0
Do While count <= MAXNUMS
  num = Val(InputBox("Enter a number", "Input Dialog", "0"))
  total = total + num
  lstDisplay.Items.Add("The number just entered is " & num)
  average = total / count
  count = count + 1
Loop
lstDisplay.Items.Add("The average of these numbers is " & average)
```

Will the program yield the correct result with this `Do While` loop?

From a programming perspective, which `Do While` loop is better and why?

9. a. The following data was collected on a recent automobile trip.

	Mileage	Gallons
Start of trip:	22495	Full tank
	22841	12.2
	23185	11.3
	23400	10.5
	23772	11.0
	24055	12.2
	24434	14.7
	24804	14.3
	25276	15.2

Write a Visual Basic program that accepts a mileage and gallons value and calculates the miles-per-gallon (mpg) achieved for that segment of the trip. The miles-per-gallon is obtained as the difference in mileage between fill-ups divided by the number of gallons of gasoline used.

b. Modify the program written for Exercise 9a to additionally compute and display the cumulative mpg achieved after each fill-up. The cumulative mpg is calculated as the difference between the mileage at each fill-up and the mileage at the start of the trip, divided by the sum of the gallons used to that point in the trip.

10. a. A bookstore summarizes its monthly transactions by keeping the following information for each book in stock:

International Standard Book Number (ISBN)

Inventory balance at the beginning of the month

Number of copies received during the month

Number of copies sold during the month

Write a Visual Basic program that accepts this data for each book, and then displays the ISBN and an updated book inventory balance using the relationship:

New balance = Inventory balance at the beginning of the month
+ Number of copies received during the month
− Number of copies sold during the month

Your program should use a `Do While` loop with a fixed count condition, so that information on only three books is requested.

b. Run the program written in Exercise 10a on a computer. Review the display produced by your program and verify that the output produced is correct.

11. Modify the program you wrote for Exercise 10a to keep requesting and displaying results until a sentinel identification value of 999 is entered. Run the program on a computer.

6.4 For/Next Loops

As we have seen, the condition used to control a `Do While` loop can be used either to test the value of a counter or to test for a sentinel value. Loops controlled by a counter are referred to as *fixed-count* loops, because the loop is executed a fixed number of times. The creation of fixed-count loops always requires initializing, testing, and modifying the counter variable. The general form we have used for these steps has been:

```
Initialize counter
Do While counter <= final value
          statement(s)
          counter = counter + increment
Loop
```

The need to initialize, test, and alter a counter to create a fixed-count loop is so common that Visual Basic provides a special structure, called the **For/Next** loop, that groups all of these operations together on a single line. The general form of a **For/Next** loop is:

```
For variable = start To end Step increment
  statement(s)
Next variable
```

Although the `For/Next` loop looks a little complicated, it is really quite simple if we consider each of its parts separately. A `For/Next` loop begins with a `For` statement. This statement, beginning with the keyword `For`, provides four items that control the loop: a variable name, a starting value for the variable, an ending value, and an increment value. Except for the increment, each of these items must be present in a `For` statement, including the equal sign and the keyword `To`, used to separate the starting and ending

values. If an increment is included, the keyword **Step** must also be used to separate the increment value from the ending value.

The variable name can be any valid Visual Basic name and is referred to as the *loop counter* (typically the counter is chosen as an integer variable); *start* is the starting (initializing) value assigned to the counter; *end* is the maximum or minimum value the counter can have and determines when the loop is finished; and *increment* is the value that is added to or subtracted from the counter each time the loop is executed. If the increment is omitted, it is assumed to be 1. Examples of valid `For` statements are as follows:

```
For count = 1 To 7 Step 1
For I = 5 To 15 Step 2
For kk = 1 To 20
```

In the first `For` statement, the counter variable is named `count`, the initial value assigned to `count` is 1, the loop will be terminated when the value in `count` exceeds 7, and the increment value is 1. In the next `For` statement, the counter variable is named `I`, the initial value of `I` is 5, the loop will be terminated when the value in `I` exceeds 15, and the increment is 2. In the last `For` statement, the counter variable is named `kk`, the initial value of `kk` is 1, the loop will be terminated when the value of `kk` exceeds 20, and a default value of 1 is used for the increment.

For each `For` statement there must be a matching `Next` statement. The `Next` statement both defines where the loop ends and is used to increment the counter variable by the increment amount defined in the `For` statement. If no increment has been explicitly listed in the `For` statement, the counter is incremented by one.

The `Next` statement formally marks the end of the loop and causes the counter to be incremented. It then causes a transfer back to the beginning of the loop. When the loop is completed, program execution continues with the first statement after the `Next` statement.

Consider the loop contained within Event Procedure 6–5 as a specific example of a **For/Next** loop.

Event Procedure 6–5

```
Private Sub frmMain_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles MyBase.Click
  Dim count As Integer

  lstDisplay.Items.Clear()
  lstDisplay.Items.Add("NUMBER   SQUARE")

  For count = 1 To 5
    lstDisplay.Items.Add("       " & count & "            " & count*count)
  Next count

End Sub
```

When Event Procedure 6–5 is executed, the following display is produced in the List box lstDisplay:

```
NUMBER    SQUARE
  1         1
  2         4
  3         9
  4         16
  5         25
```

The first line displayed by the program is produced by the lstDisplay statement placed before the For statement. The statements within the For/Next loop produce the remaining output. This loop begins with the For statement and ends with the Next statement.

The initial value assigned to the counter variable count is 1. Because the value in count does not exceed the final value of 5, the statements in the loop, including the Next statement, are executed. The execution of the lstDisplay.Items.Add statement within the loop produces the following display:

```
1         1
```

The Next statement is then encountered, incrementing the value in count to 2, and control is transferred back to the For statement. The For statement then tests whether count is greater than 5 and repeats the loop, producing the following display:

```
2         4
```

This process continues until the value in count exceeds the final value of 5, producing the complete output of numbers and squares displayed above. For comparison purposes, a Do While loop equivalent to the For/Next loop is contained in Event Procedure 6–5:

```
count = 1
Do While count <= 5
  lstDisplay.Items.Add("       " & count & "            " & count*count)
  count = count + 1
Loop
```

As seen in this example, the difference between the For/Next and Do While loops is the placement of the initialization, condition test, and incrementing items. The grouping together of these items in a For statement is very convenient when fixed-count loops must be constructed. See if you can determine the output produced by Event Procedure 6–6.

Event Procedure 6–6

```
Private Sub frmMain_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles MyBase.Click
  Dim count As Integer
```

```
  For count = 12 To 20 Step 2
    lstDisplay.Items.Add(count)
  Next count
End Sub
```

The loop starts with count initialized to 12, stops when count exceeds 20, and increments count in steps of 2. The actual statements executed include all statements following the **For** statement, up to and including the `Next` statement. The output produced by Event Procedure 6–6 is:

```
12
14
16
18
20
```

For/Next Loop Rules

Now that we have seen a few simple examples of `For/Next` loop structures, it is useful to summarize the rules that all `For/Next` loops must adhere to:

1. The first statement in a `For/Next` loop must be a `For` statement and the last statement in a `For/Next` loop must be a `Next` statement.
2. The `For/Next` loop counter variable may be either a real or integer variable.
3. The initial, final, and increment values may all be replaced by variables or expressions, as long as each variable has a value previously assigned to it and the expressions can be evaluated to yield a number. For example, the `For` statement:

   ```
   For count = begin To begin + 10 Step Augment
   ```

   is valid and can be used as long as values have been assigned to the variables `begin` and `augment` before this `For` statement is encountered in a program.
4. The initial, final, and increment values may be positive or negative, but the loop will not be executed if any one of the following is true:

a. The initial value is greater than the final value and the increment is positive.

b. The initial value is less than the final value and the increment is negative.

5. An infinite loop is created if the increment is zero.
6. An `Exit For` statement may be embedded within a `For/Next` loop to cause a transfer out of the loop.

Once a `For/Next` loop is correctly structured, it is executed as follows:[1]

---

[1]The number of times that a For/Next loop is executed is determined by the expression:

```
INT((final value - initial value + increment)/increment)
```

Step 1. The initial value is assigned to the counter variable.

Step 2. The value in the counter is compared to the final value.

For positive increments, if the value is less than or equal to the final value, then:

All loop statements are executed; and

The counter is incremented and step 2 is repeated.

For negative increments if the value is greater than or equal to the final value, then:

All loop statements are executed.

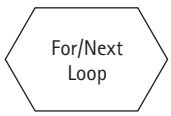
The counter is decremented and step 2 is repeated,

Else

The loop is terminated.

It is extremely important to realize that no statement within the loop should ever alter the value in the counter because the `Next` statement increments or decrements the loop counter automatically. The value in the counter may itself be displayed, as in Event Procedures 6–5 and 6–6, or used in an expression to calculate some other variable. However, it must never be used on the left-hand side of an assignment statement or altered within the loop. Also note that when a `For/Next` loop is completed, the counter contains the last value that exceeds the final tested value.

Figure 6–14 illustrates the internal workings of the `For/Next` loop for positive increments. To avoid the necessity of always illustrating these steps, a simplified flowchart symbol has been created. Using the following flowchart symbol to represent a `For/Next` statement



complete `For/Next` loops alternatively can be illustrated as shown on Figure 6–15.

To understand the enormous power of `For/Next` loops, consider the task of printing a table of numbers from 1 to 10, including their squares and cubes, using a `For/Next` statement. Such a table was previously produced using a `Do While` loop in Event Procedure 6–3. You may wish to review Event Procedure 6–3 and compare it to Event Procedure 6–7 to get a further sense of the equivalence between `For/Next` and `Do While` loops. Both Event Procedures 6–3 and 6–7 produced the same output.

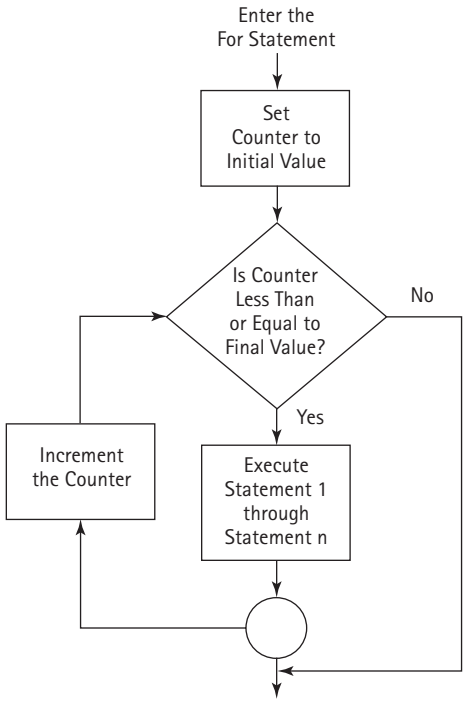


Figure 6–14 *For/Next Loop Flowchart for Positive Increments*

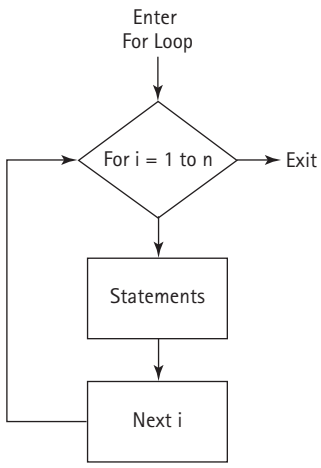


Figure 6–15 *Simplified For/Next Loop Flowchart*

Programmer Notes

Which Loop Should You Use?

Beginning programmers frequently ask which loop structure they should use—a **For** or **Do While** loop?

In Visual Basic, the answer is relatively straightforward, because the **For** statement can be used only to construct fixed-count loops. Although both **For** and **Do While** loops in Visual Basic create pretest loops, generally you should use a **For/Next** loop when constructing fixed-count loops and **Do While** loops when constructing variable-condition loops.

Event Procedure 6–7

```
Private Sub frmMain_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles MyBase.Click
  Dim num As Integer

  lstDisplay.Items.Add("NUMBER SQUARE CUBE")
  For num = 1 To 10
    lstDisplay.Items.Add("     " & num & "          " & num ^ 2 & "         " _
        & num ^ 3)
  Next num
End Sub
```

When Event Code 6–6 is activated, the following display is produced in the `list` box `lstDisplay`:

```
NUMBER   SQUARE      CUBE
  1        1          1
  2        4          8
  3        9          27
  4        16          64
  5        25          125
  6        36          216
  7        49          343
  8        64          512
  9        81          729
  10        100          1000
```

Simply changing the number 10 in the `For/Next` statement of Program 6–11 to 1000 creates a loop that is executed 1,000 times and produces a table of numbers from 1 to

1000. As with the `Do While` loop, this small change produces an immense increase in the processing and output provided by the program.

Exercises 6.4

Note: in all the problems listed below, output should be displayed in a ListBox control.

1. Write individual **For** statements for the following cases:
   a. Use a counter named `I` that has an initial value of 1, a final value of 20, and an increment of 1.
   b. Use a counter named `count` that has an initial value of 1, a final value of 20, and an increment of 2.
   c. Use a counter named `J` that has an initial value of 1, a final value of 100, and an increment of 5.
   d. Use a counter named `count` that has an initial value of 20, a final value of 1, and an increment of −1.
   e. Use a counter named `count` that has an initial value of 20, a final value of 1, and an increment of −2.
   f. Use a counter named `count` that has an initial value of 1.0, a final value of 16.2, and an increment of 0.2.
   g. Use a counter named `xcnt` that has an initial value of 20.0, a final value of 10.0, and an increment of −0.5.
2. Determine the number of times that each **For/Next** loop is executed for the statements written for Exercise 1.
3. Determine the value in `total` after each of the following loops is executed.
   a.
```
total = 0
For I = 1 To 10
  total = total + I
Next I
```
   b.
```
total = 1
For count = 1 To 10
  total = total * 2
Next count
```
   c.
```
total = 0
For I = 10 To 15
  total = total + I
Next I
```
   d.
```
total = 50
For I = 1 To 10
  total = total - I
Next I
```

e.
```
total = 1
For icnt = 1 To 8
  total = total * icnt
Next icnt
```

f.
```
total = 1.0
For J = 1 To 5
  total = total / 2.0
Next J
```

4. Determine the errors in the following **For/Next** statements:

   a. `For I = 1,10`

   b. `For count 5,10`

   c. `For JJ = 1 To 10 Increment 2`

   d. `For kk = 1, 10, -1`

   e. `For kk = -1, -20`

5. Determine the output of the following **For** loop:

```
Dim I As Integer
For I = 20 To 0 Step -4
  lstDisplay.Items.Add(I)
Next I
```

6. Modify Event Procedure 6–7 to produce a table of the numbers 0 through 20 in increments of 2, with their squares and cubes.

7. Modify Event Procedure 6–7 to produce a table of numbers from 10 to 1, instead of 1 to 10 as it currently does.

8. Write a Visual Basic program that uses a `For/Next` loop to accumulate the sum 1 + 2 + 3 + ... + N, where N is a user-entered integer number. Then evaluate the expression N* (N + 1) /2 to verify that this expression yields the same result as the loop.

9. a. An old Arabian legend has it that a fabulously wealthy but unthinking king agreed to give a beggar one cent and double the amount for 64 days. Using this information, write, run, and test a Visual Basic program that displays how much the king must pay the beggar on each day. The output of your program should appear as follows:

```
Day         Amount Owed
 1              0.01
 2              0.02
 3              0.04
 .              .
 .              .
 .              .
 64             .
```

b. Modify the program you wrote for Exercise 9a to determine on which day the king will have paid a total of one million dollars to the beggar.

10. Write and run a program that calculates and displays the amount of money available in a bank account that initially has $1,000 deposited in it and earns 8% interest a year. Your program should display the amount available at the end of each year for a period of ten years. Use the relationship that the money available at the end of each year equals the amount of money in the account at the start of the year plus 0.08 times the amount available at the start of the year.

11. A machine purchased for $28,000 is depreciated at a rate of $4000 a year for seven years. Write and run a Visual Basic program that uses a `For/Next` loop to compute and display a seven-year depreciation table. The table should have the form:

```
Depreciation Schedule
                          End-of-year     Accumulated
Year    Depreciation        value        depreciation
----    ------------      -----------    ------------
 1          4000             24000            4000
 2          4000             20000            8000
 3          4000             16000            12000
 4          4000             12000            16000
 5          4000             8000             20000
 6          4000             4000             24000
 7          4000             0                28000
```

12. A well-regarded manufacturer of widgets has been losing 4% of its sales each year. The annual profit for the firm is 10% of sales. This year, the firm has had $10 million in sales and a profit of $1 million. Determine the expected sales and profit for the next 10 years. Your program should complete and produce a display as follows:

```
Sales and Profit Projection
Year        Expected sales         Projected profit
----        --------------         ----------------
  1         $10000000                $1000000
  2         $ 9600000                $ 960000
  3             .                      .
  .             .                      .
  .             .                      .
  .             .                      .
  10            .                      .
-------------------------------------------------
Totals:    $    .                 $    .
```

13. According to legend, the island of Manhattan was purchased from its Native American population in 1626 for $24. Assuming that this money was invested in a Dutch bank paying 5 percent simple interest per year, construct a table showing how much money the Indians would have at the end of each twenty year period, starting in 1626 and ending in 2006.

6.5 Nested Loops

There are many situations in which it is very convenient to have a loop contained within another loop. Such loops are called *nested loops*. A simple example of a nested loop is:

```
For I = 1 To 4    '<-- Start of Outer Loop
  lstDisplay.Items.Add("I is now " & I)
  For J = 1 To 3  '<-- Start of Inner Loop
    lstDisplay.Items.Add("  J = " & J)
  Next J          '<-- End of Inner Loop
Next I            '<-- End of Outer Loop
```

The first loop, controlled by the value of `I`, is called the *outer loop*. The second loop, controlled by the value of `J`, is called the *inner loop*. Note that all statements in the inner loop are contained within the boundaries of the outer loop, and that we have used a different variable to control each loop. For each single trip through the outer loop, the inner loop runs through its entire sequence. Thus, each time the `I` counter increases by one, the inner `For/Next` loop executes completely. This situation is illustrated in Figure 6–16.

To understand the concept involved, consider Program 6–5, which uses a nested `For/Next` loop. Figure 6–17 shows the interface for Program 6–5.

For this application, the procedure code for the Button **Click** events is listed in Program 6–5's event code.

Program 6–5's Event Code

```
Private Sub btnRun_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles btnRun.Click
  Dim I, J As Integer

  lstDisplay.Items.Clear()
  For I = 1 To 4    '<-- Start of Outer Loop
    lstDisplay.Items.Add("I is now " & I)
    For J = 1 To 3 '<-- Start of Inner Loop
      lstDisplay.Items.Add("  J = " & J)
    Next J      '<-- End of Inner Loop
  Next I        '<-- End of Outer Loop

  End Sub

  Private Sub btnExit_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles btnExit.Click
  Beep()
  End
End Sub
```

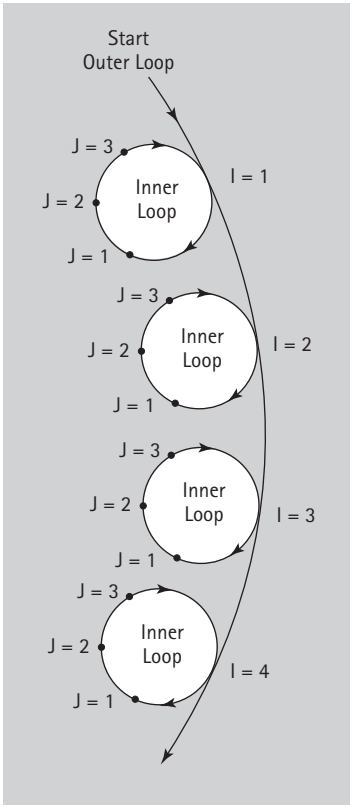


Figure 6–16 *For Each I, J Makes a Complete Loop*

Figure 6–18 illustrates the display produced when Program 6–5 is executed.

When you are creating nested loops using any of Visual Basic's loop structures (and the various structures can be nested within one another), the only requirements that must be adhered to are the following:

1. An inner loop must be fully contained within an outer loop.
2. The inner loop and outer loop control variables cannot be the same.
3. An outer loop control variable must not be altered within an inner loop.

Let us use a nested loop to compute the average grade for each student in a class of 10 students. Each student has taken four exams during the course of the semester. The final grade for each student is calculated as the average of the four examination grades. The pseudocode for this example is:

```
Do 10 times
  Set student total to zero
  Do 4 times
    Read in a grade
    Add the grade to the student total
```

Figure 6–17 *Program 6-5's Interface*

Table 6–3 The Properties Table for Program 6–5

Object	Property	Setting
Form	Name	frmMain
	Text	Program 6–5
ListBox	Name	lstDisplay
Button	Name	btnRun
	Text	&Show Nested Loop
Button	Name	btnExit
	Text	E&xit

```
  End inner Do
Calculate student's average grade
Print student's average grade
End outer Do
```

As described in the pseudocode, an outer loop consisting of 10 passes will be used to calculate the average for each student. The inner loop will consist of four passes, with one examination grade entered in each inner loop pass. As each grade is entered, it is added to the total for the student, and at the end of the loop, the average is calculated

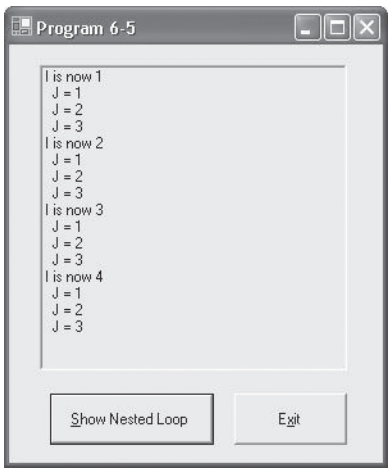


Figure 6–18 *The Display Produced by Program 6–5*

and displayed. Program 6–6 uses a nested loop to make the required calculations. Figure 6–19 shows the interface for Program 6–6.

For this application, the procedure code for the `Button Click` event is listed in Program 6–6's event code.

Program 6–6's Event Code

```
Private Sub btnRun_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles btnRun.Click
  Const MAXSTUDENTS As Integer = 10
  Const NUMGRADES As Integer = 4
  Dim i, j As Integer
  Dim grade, total, average As Single

  ' This is the start of the outer loop
  For i = 1 To MAXSTUDENTS
    total = 0
    ' This is the start of the inner loop
    For j = 1 To NUMGRADES
      grade = Val(InputBox("Enter an exam grade for this student", _
          "Input Dialog", "0"))
      total = total + grade
    Next j   ' End of inner loop
    average = total / NUMGRADES
    lstDisplay.Items.Add("The average for student " & i & " is " & _
                         average)
  Next i 'End of outer loop
End Sub
```

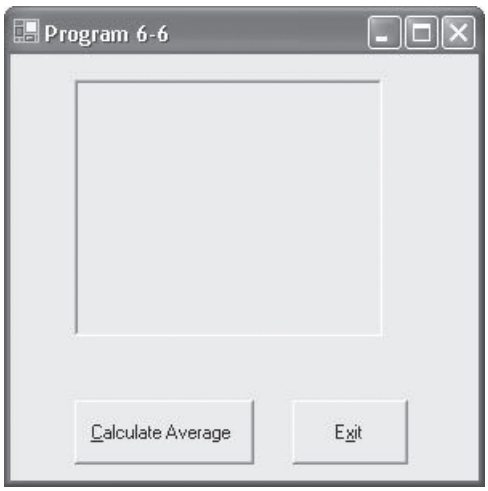


Figure 6–19 *Program 6–6's Interface*

Table 6–4 The Properties Table for Program 6–6

Object	Property	Setting
Form	Name	frmMain
	Text	Program 6–6
ListBox	Name	lstDisplay
Button	Name	btnRun
	Text	&Calculate Average
Button	Name	btnExit
	Text	E&xit

```
Private Sub btnExit_Click(ByVal sender As Object, ByVal e As _
    System.EventArgs) Handles btnExit.Click
  Beep()
  End
End Sub
```

In reviewing Program 6–6, pay particular attention to the initialization of `total` within the outer loop before the inner loop is entered; `total` is initialized 10 times, once for each student. Also note that the average is calculated and displayed immediately after the inner loop is finished. Because the statements that compute and print the average are also contained within the outer loop, 10 averages are calculated and displayed. The entry and addition of each grade within the inner loop uses summation techniques we have seen before that should now be familiar to you.

It may be useful to provide a way out of this program to avoid having to enter four grades for 10 students. With the code above, clicking the `Cancel` button in the `InputBox` dialog box just enters a 0 for that particular grade and continues. However, to have the `Cancel` terminate the program, the code must examine the return value of `InputBox`. When the user clicks on Cancel, the return value for InputBox is set to the null string "". An **If** statement can check for this value and if it is equal to the null string, then you want to exit from the **For** loop. However, just doing that only exits from the inner **For** loop. To completely exit from the program, you need to put another check in the outer **For** loop with a second **Exit For**. It is also important to note that the conversion of the return value to an integer using the **Val** function needs to be done after the return value is checked for equality with the null string. So a string variable `returnval` has been added to the program. The following code provides a way to terminate Program 6–6:

```
Private Sub cmdRun_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles cmdRun.Click
  Const MAXSTUDENTS As Integer = 10
  Const NUMGRADES As Integer = 4

  Dim i, j As Integer
  Dim grade, total, average As Single
  Dim returnval As String

  ' This is the start of the outer loop
  For i = 1 To MAXSTUDENTS
    total = 0
    ' This is the start of the inner loop
    For j = 1 To NUMGRADES
      returnval = InputBox("Enter an exam grade for this student", _
          "Input Dialog", "0")
      If returnval = "" Then 'InputBox canceled
        Exit For  ' exit inner loop
      End If
      grade = Val(returnval)
      total = total + grade
    Next j   ' End of inner loop
    If returnval = "" Then
      Exit For  ' exit outer loop
```

```
    End If
    average = total / NUMGRADES
    lstDisplay.Items.Add("The average for student " & i & " is " & _
                         average)
  Next i   ' End of outer loop
End Sub
```

Exercises 6.5

Note: in all the problems listed below, output should be displayed in a ListBox control.

1. Four experiments are performed, with each experiment consisting of six test results. The results for each experiment follow. Write a program using a nested loop to compute and display the average of the test results for each experiment.

1st experiment results:	23.2	31.5	16.9	27.5	25.4	28.6
2nd experiment results:	34.8	45.2	27.9	36.8	33.4	39.4
3rd experiment results:	19.4	16.8	10.2	20.8	18.9	13.4
4th experiment results:	36.9	39.5	49.2	45.1	42.7	50.6

2. Modify the program written for Exercise 1 so that the user enters the number of test results for each experiment. Write your program so that a different number of test results can be entered for each experiment.

3. a. A bowling team consists of five players. Each player bowls three games. Write a Visual Basic program that uses a nested loop to enter each player's individual scores and then computes and displays the average score for each bowler. Assume that each bowler has the following scores:

1st bowler:	286	252	265
2nd bowler:	212	186	215
3rd bowler:	252	232	216
4th bowler:	192	201	235
5th bowler:	186	236	272

   b. Modify the program written for Exercise 3a to calculate and display the average team score. (*Hint:* Use a second variable to store the total of all the players' scores.)

4. Rewrite the program written for Exercise 3a to eliminate the inner loop. To do this, you will have to input three scores for each bowler rather than one at a time.

5. Write a program that calculates and displays values for *Y* when:

   `Y = X * Z/(X - Z)`

   Your program should calculate *Y* for values of *X* ranging between 1 and 5, and values of *Z* ranging between 2 and 10. *X* should control the outer loop and be incremented in steps of one, and *Z* should be incremented in steps of two. Your program should also display the message `Value Undefined` when the *X* and *Z* values are equal.

6. Write a program that calculates and displays the yearly amount available if $1,000 is invested in a bank account for 10 years. Your program should display the amounts available for interest rates from 6% to 12% inclusively, in 1% increments. Use a nested loop, with the outer loop controlling the interest rate and the inner loop controlling the years. Use the relationship that the money available at the end of each year equals the amount of money in the account at the start of the year, plus the interest rate times the amount available at the start of the year.
7. In the Duchy of Penchuck, the fundamental unit of currency is the Penchuck Dollar (PD). Income tax deductions are based on Salary in units of 10,000 PD and on the number of dependents the employee has. The formula, designed to favor low-income families, is:

   Deduction PD = Dependents 500 + 0.05 *(50,000 − Salary)*

   Beyond 5 dependents and beyond 50,000 PD, the Deduction does not change. There is no tax—hence no deduction—on incomes of less than 10,000 PD. Based on this information, create a table of Penchuck income tax deductions, with Dependents 0 to 5 as the column headings and salary 10000, 20000, 30000, 40000, and 50000 as the rows.

6.6 Exit-Controlled Loops

The `Do While` and `For/Next` loops are both entrance-controlled loops, meaning that they evaluate a condition at the start of the loop. A consequence of testing a condition at the top of the loop is that the statements within the loop may not be executed at all.

There are cases, however, where we always require a loop to execute at least once. For such cases, Visual Basic provides two exit-controlled loops, the `Do/Loop Until` and `Do/Loop While` structures. Each of these loop structures tests a condition at the bottom of a loop, ensureing that the statements within the loop are executed at least one time.

The Do/Loop Until Structure

The syntax for the most commonly used exit-controlled loop is:

```
Do
  statement(s)
Loop Until condition
```

The important concept to note in the `Do/Loop Until` structure is that all statements within the loop are executed at least once before the condition is tested, and the loop is repeated until the condition becomes **True** (in other words, the loop executes while the condition is **False**). A flowchart illustrating the operation of the `Do/Loop Until` loop is shown in Figure 6–20.

As illustrated in Figure 6–20, all statements within the `Do/Loop Until` loop are executed once before the condition is evaluated. Then, if the condition is **False**, the statements within the loop are executed again. This process continues until the condition becomes **True**.

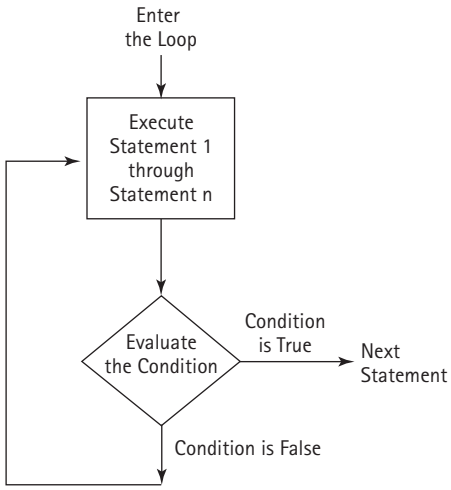


Figure 6–20 *The Do/Loop Until Loop's Flowchart*

As an example of a **Do/Loop Until** structure, consider the following loop:

```
i = 0
Do
  lstDisplay.Items.Add(i)
  i = i + 5
Loop Until i > 20
```

The output produced in the list box lstDisplay by this code is:

```
0
5
10
15
20
```

The important point to note here is that the condition is tested after the statements within the loop structure have been executed, rather than before. This ensures that the loop statements are always executed at least once. Also note that exit-controlled loops require you to correctly initialize all variables used in the tested expression before the loop is entered, in a manner similar to that used in entrance-controlled loops. Similarly, within the loop itself, these same variables must be altered to ensure that the loop eventually terminates.

The Do/Loop While Structure

In a `Do/Loop Until` structure, the statements within the loop are executed as long as the condition is **False**. A variation of the `Do/Loop Until` structure is the `Do/Loop While` structure, which executes the statements within the loop as long as the condition is **True**. The syntax of a `Do/Loop While` loop is:

```
Do
  statement(s)
Loop While condition
```

The `Do/Loop While` structure, like its `Do/Loop Until` counterpart, is an exit-controlled loop. Unlike the `Do/Loop Until` loop, which executes until the condition becomes **True**, a `Do/Loop While` structure executes until the condition becomes **False**. If the condition tested in a `Do/Loop While` loop is **False** to begin with, the statements within the loop will execute only once.

Validity Checks

Exit-controlled loops are particularly useful for filtering user-entered input and validating that the correct type of data has been entered. For example, assume that a program is being written to provide the square root of any number input by the user. For this application, we want to ensure that a valid number has been entered. Any invalid data, such as a negative number or nonnumeric input, should be rejected and a new request for input should be made until the user actually enters a valid number. Program 6–7 illustrates how this request can be accomplished easily using a `Do/Loop Until` structure. Figure 6–21 shows the interface for Program 6–7.

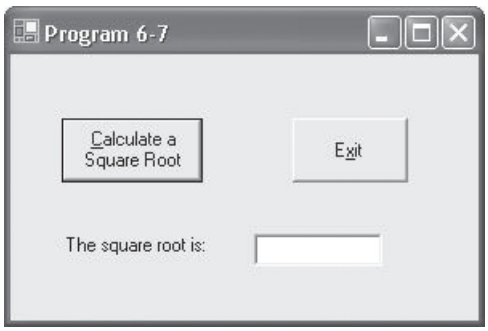


Figure 6–21 *Program 6–7's Interface*

Program 6–7's Event Code

```
Private Sub btnRun_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles btnRun.Click
  Dim num As String

  Do
    txtSquare.Clear()
    num = InputBox("Enter a number", "Input Request", "0")
    If IsNumeric(num) And Val(num) >= 0 Then
      txtSquare.Text = Math.Sqrt(num)
    Else
      MessageBox.Show("Invalid data was entered", "Error Message", _
          MessageBoxButtons.OK, MessageBoxIcon.Error)
    End If
  Loop Until IsNumeric(num) And Val(num) >= 0
End Sub

Private Sub btnExit_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles btnExit.Click
  Beep()
  End
 End Sub
```

Note that in the run button **Click** code, a request for a number is repeated until a valid number is entered. This code is constructed so that a user cannot "crash" the program by entering either a nonnumeric string or a negative number. This type of user input validation is essential for programs that are to be used extensively by many people, and is a mark of a professionally written program.

Although this example is showing the validity check on exit in a **Do/Loop Until** structure, a validity check could also be done on entrance in a **Do/While** structure.

Exercises 6.6

Note: in all the problems listed below, output should be displayed in a ListBox control.

1. a. Using a `Do/Loop Until` structure, write a program to accept a grade. The program should continuously request a grade until a valid grade is entered. A valid grade is any grade greater than or equal to 0 and less than or equal to 101. After a valid grade has been entered, your program should display the value of the grade entered.

   b. Modify the program written for Exercise 1a so that it allows the user to exit the program by entering the number 999.

   c. Modify the program written for Exercise 1b so that it allows the user to exit the program by entering the number 999.

   d. Modify the program written for Exercise 1c so that it automatically terminates after five invalid grades are entered.

2. a. Modify the program written for Exercise 1a as follows: If the grade is less than 0 or greater than 100, your program should print an appropriate message informing the user that an invalid grade has been entered; otherwise, the grade should be added to a total. When a grade of 999 is entered, the program should exit the repetition loop and compute and display the average of the valid grades entered.

   b. Run the program written in Exercise 2a on a computer and verify the program using appropriate test data.

3. a. Write a program to reverse the digits of a positive integer number. For example, if the number 8735 is entered the number displayed should be 5378. *Hint:* Use an exit-controlled loop, and continuously strip off and display the units digit of the number. If the variable `num` initially contains the number entered, the units digit is obtained as `num Mod 10`. After a units digit is displayed, dividing the number by 10 sets up the number for the next iteration. Thus, `8735 Mod 10` is `5` and `8735 / 10` is `873`. The loop should repeat as long as the remaining number is not zero.

   b. Run the program written in Exercise 3a on a computer and verify the program using appropriate test data.

4. a. The outstanding balance on Rhona Karp's car loan is $8,000. Each month, Rhona is required to make a payment of $300, which includes both interest and principal repayment of the car loan. The monthly interest is calculated as 0.10/12 of the outstanding balance of the loan. After the interest is deducted, the remaining part of the payment is used to pay off the loan. Using this information, write a Visual Basic program that produces a table indicating the beginning monthly balance, the interest payment, the principal payment, and the remaining loan balance after each payment is made. Your output should resemble and complete the entries in the following table until the outstanding loan balance is zero.

```
Beginning       Interest    Principal  Ending Loan
 Balance        Payment      Payment     Balance
---------       --------    ---------  ----------
  8000.00         66.67       233.33      7766.67
  7766.67         64.73       235.28      7531.39
 75531.39           .           .           .
      .             .           .           .
      .             .           .           .
      .             .           .          0.00
```

   b. Modify the program written in Exercise 4a so that the total of the interest and principal paid is displayed at the end of the table produced by your program.

5. Write, run, and test a Visual Basic program that prompts the user for the amount of a loan, the annual percentage rate, and the number of years of the loan, using `InputBox` functions that display the following prompts:

```
What is the amount of the loan?
What is the annual percentage rate?
How many years will you take to pay back the loan?
```

From the input data, produce a loan amortization table similar to the one shown below:

Amount	Annual % Interest	Years	Monthly Payment
1500	14	1	134.68

Payment Number	Interest Paid	Principal Paid	Cumulative Interest	Total Paid to Date	New Balance Due
1	17.50	117.18	17.50	134.68	1382.82
2	16.13	118.55	33.63	269.36	1264.27
3	14.75	119.93	48.38	404.04	1144.34
4	13.35	121.33	61.73	538.72	1023.01
5	11.94	122.75	73.67	673.40	900.27
6	10.50	124.18	84.17	808.08	776.09
7	9.05	125.63	93.23	942.76	650.46
8	7.59	127.09	100.81	1077.45	523.37
9	6.11	128.57	106.92	1212.13	394.79
10	4.61	130.07	111.53	1346.81	264.72
11	3.09	131.59	114.61	1481.49	133.13
12	1.55	133.13	116.17	1616.17	0

In constructing the loop necessary to produce the body of the table, the following initializations must be made:

```
New balance due = Original loan amount
Cumulative interest = 0.0
Paid to date = 0.0
Payment number = 0
Monthly Interest Rate = Annual Percentage Rate / 1200
                        (Loan Amount) * (Monthly Interest Rate)
Monthly Payment = ----------------------------------------------
                  1 - (1 + Monthly Interest Rate) -(Number of Months)
```

Within the loop, the following calculations and accumulations should be used:

Payment number = Payment number + 1

Interest paid = New balance due * Monthly interest rate

Principal paid = Monthly payment - Interest paid

Cumulative interest = Cumulative interest + Interest paid

Paid to date = Paid to date + Monthly payment

New balance due = New balance due - Principal paid

6. Modify the program written for Exercise 5, to prevent the user from entering an illegal value for the interest rate. That is, write a loop that asks the user repeatedly for the annual interest rate, until a value between 1.0 and 25.0 is entered.

6.7 Focus on Program Design and Implementation:[2] After-the-Fact Data Validation and Creating Keyboard Shortcuts[3]

In the last Focus Section, keystroke input validation was applied to the Rotech Walk In order entry forms quantity text box. This type of validation consists of checking each entered character and only permitting selected keys to be entered as valid input data. In addition to this keystroke-by-keystroke data validation, overall validation of a completed input entry is also used to ensure that: (1) a required field has not been skipped over; and (2) that data contained within a field is reasonable. This second type of data validation is referred to as after-the-fact validation.

As a practical example of this type of data validation, consider Rotech's Mail In form, a copy of which is shown in Figure 6–22.

In an extreme situation, a user could enter no data at all into any of the input text boxes shown in Figure 6–22 and still attempt to print out a packing slip. Even using keystroke validation for the quantity text box won't help, because a user can simply skip this box without entering any quantity into it. After-the-fact validation can be used to correct this type of situation. Specifically, for the Mail In form, we will apply after-the-fact validation to ensure the following requirements are met:

Figure 6–22 *The Rotech Mail In Form*

[2]This section can be omitted with no loss of subject continuity, unless the Rotech project is being followed.

[3]The Rotech project, as it should exist at the start of this section (which includes all of the capabilities built into it through the end of Section 5.6) can be found at http://computerscience.jbpub.com/bronsonvbnet in the ROTECH5 folder as project rotech5.

A packing slip will not be prepared if any name or address fields are empty or if there is not a positive number in the quantity text box. We will also assume that each customer is limited to a maximum of 5 free disk packages, so the quantity cannot exceed this value.

In addition, we will add after-the-fact data validation to ensure that the Walk In form adheres to the following requirements:

A bill will be printed only if the quantity field contains a positive number. Because the customer is purchasing disk packs, there will be no limit check on the maximum number that can be entered for this quantity. Additionally, if both the first and last name fields have been left blank, the bill will be made out to Cash.[4]

There are two predominant techniques for informing a user that a particular data entry item has either been filled in incorrectly or is missing completely. In the first method, each individual field is examined and a message displayed, one at a time, for each incorrectly completed item. For more than two fields, this method tends to annoy users and should be avoided. For example, assume that you inform the user that the Address item has not been filled in. Then, after the user corrects this field, you provide another message that the City item is missing. After this item is entered, another message appears indicating that the State has not been entered. From a user's standpoint, this is extremely annoying. A much better scheme is to provide a single message that gives a complete list of all missing items.

Figure 6–23 shows how such a message would look, assuming that all of the Mail In form's name and address fields were left empty when the `Print Packing Slip` button was pressed. Note that the message box lists all of the missing address items. The quantity input, which is not part of the address data, will be given its own set of error messages. Figures 6–24 and 6–25 show the two possible error messages that a user could receive for this input. Thus, a user will receive at most two separate error messages (one for the address data and one for the quantity input), which is within most user's tolerance zone. Clearly, these error messages can and will recur if the user does not correct the problem. However, this recurrence will not be blamed on the system, but rather on the user's inattention to detail.

Figure 6–23 *A Sample Address Information Error Message*

[4]This is a common custom in business. For customers who do not wish to give their names or addresses, the transaction is simply posted as a cash transaction.

Figure 6–24 *The Error Message for a Missing Quantity*

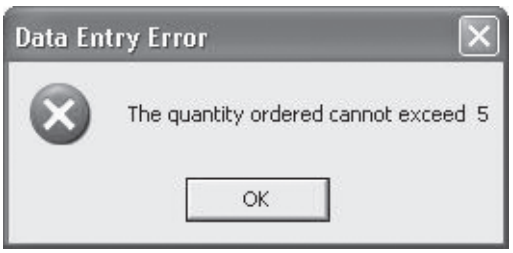


Figure 6–25 *The Error Message for Too Large a Quantity*

Event Procedure 6–8 illustrates the code used in the Rotech application to validate the Mail In form's input before a packing slip is printed. The first item to note is that, from a user's viewpoint, the procedure is meant to print a packing slip. However, from a programming viewpoint, the majority of the procedure is concerned with validating data. This is true for most real-world code; the code itself should validate all data and ensure that any data required for processing cannot inadvertently cause the system to crash or produce inappropriate results. For example, it is the program's responsibility not to print a packing slip with a blank address or an unreasonable quantity of product. All the validation in the world cannot stop a user from entering a misspelled name or a nonexistent address, but that is an error that cannot be fixed by the application. Note that the control character CrLf is used in this procedure. This stands for Carriage-return Linefeed and is equivalent to the NewLine character.

Event Procedure 6–8

```
Private Sub btnPrint_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles btnPrint.Click
  Dim strErrorPrompt, strErrorMsg, strErrorMin, strErrorMax As String
  Dim strListEmptyBoxes As String  ' the list of empty Text boxes

  Const MAXALLOWED As Integer = 5

  'Set the standard strings that will be printed
  strErrorPrompt = "Data Entry Error"
  strErrorMsg = "The following data needs to be entered: " & _
      ControlChars.CrLf
```

```
  strListEmptyBoxes = ""
  strErrorMin ="You must enter the No. of disk packages ordered."
  strErrorMax = "The quantity ordered cannot exceed " & Str(MAXALLOWED)

  'Locate all empty name and address text boxes
  If txtFname.Text.Length = 0 Then
    strListEmptyBoxes = strListEmptyBoxes & "First Name " & ControlChars.CrLf
  End If
  If txtLname.Text.Length = 0 Then
    strListEmptyBoxes = strListEmptyBoxes & "Last Name " & ControlChars.CrLf
  End If
  If txtAddr.Text.Length = 0 Then
    strListEmptyBoxes = strListEmptyBoxes & "Address " & ControlChars.CrLf
  End If
  If txtCity.Text.Length = 0 Then
    strListEmptyBoxes = strListEmptyBoxes & "City " & ControlChars.CrLf
  End If
  If txtState.Text.Length = 0 Then
    strListEmptyBoxes = strListEmptyBoxes & "State " & ControlChars.CrLf
  End If
  If txtZip.Text.Length = 0 Then
    strListEmptyBoxes = strListEmptyBoxes & "Zip Code "
  End If

  ' Inform the user of all empty text boxes
  If strListEmptyBoxes.Length > 0 Then
    MessageBox.Show(strErrorMsg & strListEmptyBoxes, strErrorPrompt, _
        MessageBoxButtons.OK, MessageBoxIcon.Error)

  ElseIf Val(txtQuantity.Text) < 1 Then
    MessageBox.Show(strErrorMin, strErrorPrompt, MessageBoxButtons.OK, _
        MessageBoxIcon.Error)
  ElseIf Val(txtQuantity.Text) > MAXALLOWED Then
    MessageBox.Show(strErrorMax, strErrorPrompt, MessageBoxButtons.OK, _
        MessageBoxIcon.Error)

  Else ' Print the packing slip
    PrintDocument1.Print()
  End If
End Sub

Private Sub PrintDocument1_PrintPage(ByVal sender As System.Object, _
    ByVal e As System.Drawing.Printing.PrintPageEventArgs) _
    Handles PrintDocument1.PrintPage
  Dim font As New Font("Courier New", 12)
```

```
    Dim x, y As Single

    x = e.MarginBounds.Left
    y = e.MarginBounds.Top

    e.Graphics.DrawString("", font, Brushes.Black, x, y)
    y = y + font.GetHeight
    e.Graphics.DrawString("", font, Brushes.Black, x, y)
    y = y + font.GetHeight

    e.Graphics.DrawString(Space(20) + FormatDateTime(Now, _
        DateFormat.GeneralDate), font, Brushes.Black, x, y)
    y = y + font.GetHeight

    e.Graphics.DrawString("", font, Brushes.Black, x, y)
    y = y + font.GetHeight
    e.Graphics.DrawString("", font, Brushes.Black, x, y)
    y = y + font.GetHeight

    e.Graphics.DrawString(Space(5) & "To: " & txtFname.Text + _
        " " & txtLname.Text, font, Brushes.Black, x, y)
    y = y + font.GetHeight
    e.Graphics.DrawString(Space(9) & txtAddr.Text, font, _
        Brushes.Black, x, y)
    y = y + font.GetHeight
    e.Graphics.DrawString(Space(9) & txtCity.Text & " " _
        & txtState.Text & " " & txtZip.Text, font, Brushes.Black, x, y)
    y = y + font.GetHeight

    e.Graphics.DrawString("", font, Brushes.Black, x, y)
    y = y + font.GetHeight
    e.Graphics.DrawString("", font, Brushes.Black, x, y)
    y = y + font.GetHeight

    e.Graphics.DrawString("Quantity:" & Space(1) & txtQuantity.Text, _
        font, Brushes.Black, x, y)
End Sub
```

Now look at the first section of code printed in black in Event Procedure 6–8. Note that the maximum allowable disk order quantity and all of the standard messages used as prompts in Figures 6–23 through 6–25 have been assigned at the top of this section, near the beginning of the procedure. The advantage of this is that if any change must be made to any of these values, the value can be easily located and modified.

Now note that the name and address validation code at the end of this first black section consists of a series of six individual **If** statements. Within each statement an

individual text box string is checked using the Length method. The *Length method* returns a string's length, consisting of the total number of characters in the string. A length of zero means that the string has no characters, indicating that no data has been entered for it. For each text box that has no data, the name of the equivalent data field and a carriage return-line feed combination are appended to the string variable named `strListEmptyBoxes`. Thus, after the last **If** statement is executed, this string variable will contain a list of all empty name and address items.

Tips from the Pros

Checking for Completed Inputs

The method used in Event Procedure 6–8 individually tests and identifies each text box that contains an empty string. When this individual identification is not necessary, a much quicker validation can be obtained by multiplying the length of all the input strings together. Then, if one or more of the strings is empty, the result of the computation will be zero.

As a specific example, assume that you need to check that three text boxes, named `txtPrice`, `txtQuantity`, and `txtDescription`, have all been filled in. The following code does this:

```
If (txtPrice.Text.Length * txtQuantity.Text.Length * _
    txtDescription.Text.Length) = 0 Then
  MessageBox.Show("One or more of the fields has not been filled in!")
Endif
```

Note that this code does not identify which, or how many, of the input items are blank. However, if one or more of the items is blank, at least one of the operands in the multiplication will be zero, and the result of the computation is zero. Note that this is the only way a zero will result, because if all of the inputs are filled in the result of the multiplication will be a positive number. Thus, a zero result ensures that at least one of the inputs has not been filled in.

This programming trick is very prevalent in real-world applications, and you are certain to see many applications of it in your programming work.

Finally, look at the second section of code printed in black, containing the validation code for the quantity of disk packages ordered. Here there are two possibilities—either the quantity has been left blank or the number of ordered packages exceeds the value in the MAXALLOWED named constant. (We don't have to check for either a negative or fractional value being entered, because the key stroke validation will prevent a minus sign or period from being entered.) The two `ElseIf` statements within this second black region check for each of these conditions.

The Walk In form's after-the-fact validation is listed as Event Procedure 6–9. As most of the code listed in this procedure was previously described in Section 4.4, we will only comment on the code specifically added for data validation. The first black section provides the prompts that we will use for an error Message box, which is the same as that previously illustrated in Figure 6–24 for the Mail In form. The second black section

contains two individual `If` statements. The first `If` statement checks that both a first name and last name have been entered; if not, the string `Cash` is substituted for the first name. (See this sections Tips From the Pros box in this section for an alternative way of making this check.) The second `If` statement validates the number of disks ordered is greater than zero.

Event Procedure 6–9

```
Private Sub btnPrint_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles btnPrint.Click
  Dim stotal, staxes, sfinal As Single
  Dim strErrorPrompt As String
  Dim strErrorMin As String

  'Set the standard strings that will be printed
  strErrorPrompt = "Data Entry Error"
  strErrorMin = "You must enter the No. of disk packages ordered."

  'Clear out the text boxes
  txtTotal.Clear()
  txtTax.Clear()
  txtFinal.Clear()

  'Calculate and display new values
  sTotal = Val(txtQuantity.Text) * CDec(txtPrice.Text)
  sTaxes = Val(txtTrate.Text) * sTotal
  sFinal = sTotal + sTaxes
  txtTotal.Text = FormatCurrency(sTotal)
  txtTax.Text = FormatCurrency(sTaxes)
  txtFinal.Text = FormatCurrency(sFinal)

  ' Validate the Name and Quantity
  If txtFname.Text.Length = 0 AND txtLname.Text.Length = 0 Then
    txtFname.Text = "Cash"
  End If
  If Val(txtQuantity.Text) < 1 Then
    MessageBox.Show(strErrorMin, strErrorPrompt, MessageBoxButtons.OK, _
        MessageBoxIcon.Error)
  Else ' Print a Bill
    PrintDocument1.Print()
  End If
End Sub

Private Sub PrintDocument1_PrintPage(ByVal sender As System.Object, _
    ByVal e As System.Drawing.Printing.PrintPageEventArgs) _
    Handles PrintDocument1.PrintPage
```

```
Dim font As New Font("Courier New", 12)
Dim x, y As Single

x = e.MarginBounds.Left
y = e.MarginBounds.Top

e.Graphics.DrawString("", font, Brushes.Black, x, y)
y = y + font.GetHeight
e.Graphics.DrawString("", font, Brushes.Black, x, y)
y = y + font.GetHeight

e.Graphics.DrawString(Space(20) + FormatDateTime(Now, _
    DateFormat.ShortDate), font, Brushes.Black, x, y)
y = y + font.GetHeight
e.Graphics.DrawString(Space(20) + FormatDateTime(Now, _
    DateFormat.LongTime), font, Brushes.Black, x, y)
y = y + font.GetHeight

e.Graphics.DrawString("", font, Brushes.Black, x, y)
y = y + font.GetHeight
e.Graphics.DrawString("", font, Brushes.Black, x, y)
y = y + font.GetHeight

e.Graphics.DrawString("Sold to: " & txtFname.Text + _
    " " & txtLname.Text, font, Brushes.Black, x, y)
y = y + font.GetHeight
e.Graphics.DrawString(Space(9) & txtAddr.Text, font, _
    Brushes.Black, x, y)
y = y + font.GetHeight
e.Graphics.DrawString(Space(9) & txtCity.Text & " " _
    & txtState.Text & " " & txtZip.Text, font, Brushes.Black, x, y)
y = y + font.GetHeight

e.Graphics.DrawString("", font, Brushes.Black, x, y)
y = y + font.GetHeight
e.Graphics.DrawString("", font, Brushes.Black, x, y)
y = y + font.GetHeight

e.Graphics.DrawString("Quantity:" & Space(1) & txtQuantity.Text, _
    font, Brushes.Black, x, y)
y = y + font.GetHeight
e.Graphics.DrawString("Unit Price:" & Space(2) & _
    FormatCurrency(txtPrice.Text), font, Brushes.Black, x, y)
y = y + font.GetHeight
```

```
  e.Graphics.DrawString("Total:" & Space(6) & txtTotal.Text, _
      font, Brushes.Black, x, y)
  y = y + font.GetHeight
  e.Graphics.DrawString("Sales tax:" & Space(3) & txtTax.Text, _
      font, Brushes.Black, x, y)
  y = y + font.GetHeight
  e.Graphics.DrawString("Amount Due:" & Space(1) & txtFinal.Text, _
      font, Brushes.Black, x, y)
End Sub
```

Using Function Keys as Keyboard Shortcuts

Many applications reserve the Function keys (F1, F2, and so on) for use as keyboard shortcuts. A keyboard shortcut is a single key, or combination of keys, that performs a distinct task no matter when or where it is pressed in the application.

For example, a common practice is to allocate the F1 key as the keyboard shortcut for calling up a Help screen, so that whenever this key is pressed a Help screen appears. Another use is to allocate a function key as a "Return to Main Menu" keyboard shortcut. Using the designated key (no matter where it is pressed in an application) guarantees hiding/unloading of the currently displayed form and display of the Main Menu form. For applications with long chains of forms, where one form is used to call another, a specific function key can also be used as a "Return to Main Menu" keyboard shortcut. By pressing the designated key, the user recalls the previous form. This permits a user to back up as many forms as necessary, even to the point of recalling the first screen displayed by the application.

The first requirement for capturing a pressed function key, independent of which specific control has the focus when the key is pressed, is to set a form's **KeyPreview** property to **True**. Doing this ensures that no matter which control has focus, the keyboard events **KeyDown**, **KeyUp**, and **KeyPress** will be invoked as Form events before they are invoked for any control placed on a form. Because the form keyboard events supercede all other control keyboard events, the desired function key is detected as a `Form` event first and is recognized as such no matter where on the form or within a control that the key is pressed.

As a specific example, assume that we want to make the F11 key a "Return to Main Menu" keyboard shortcut. The following event code does this:

```
Private Sub Form1_KeyDown(ByVal sender As Object, _
    ByVal e As System.Windows.Forms.KeyEventArgs) Handles MyBase.KeyDown
  If e.KeyCode = System.Windows.Forms.Keys.F11 Then  ' check for F11 key
    Dim frmMainRef as New frmMain()
    e.Handled = True
    Me.Hide()
    frmMainRef.Show()
  End If
End Sub
```

In this event code, we check for the constant `System.Windows.Forms.Keys.F11`, which is the named constant for the **KeyCode** returned by the F11 key. When the F11

function key is pressed, the `If` statement first sets e.Handled to **True** to ensure that the key is not passed on to the specific control currently having the focus. Then the current Form is hidden, and the frmMain form is displayed. Assuming this is the name of the Main Menu form, the user now sees this form displayed.

All that remains is to explicitly set each form's **KeyPreview** property to **True**, which is necessary to ensure that the **Form-level Keyboard event** is triggered. This is easily done within the code that displays each form for which we want the function keys to be active at the form level. For example, the following event code, which is used to display the Walk In form from Rotech's Main Menu screen, illustrates how this form's **KeyPreview** property is set before the form is displayed.

```
Private Sub btnWalkins_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnWalkins.Click
  Dim frmWalkRef as New frmWalkIn()

  frmWalkRef.KeyPreview() = True
  Me.Hide
  frmWalkRef.Show()
End Sub
```

Although the example presented here applies to the F11 key, it is easily extended to any key for which you know the **KeyCode**. Exercise 4 in the exercise set for Section 6.7 shows how to obtain these codes using the Help facility.

Finally, as a technical note, a **KeyCode** is recognized within all keyboard event procedures, while the **e.Keychar** codes used in last chapter's Focus section are only recognized within **KeyPress** events.

Exercises 6.7

*(**Note:** The Rotech Systems project, at the stage of development begun in this section, can be found at http://computerscience.jbpub.com/bronsonvbnet in the ROTECH5 folder as project rotech5. The project, as it exists at the end of this section, can be found in the ROTECH6 folder as project rotech6. If you are developing the system yourself, following the procedures given in this section, we suggest that you first copy all of the files in the ROTECH5 folder onto your system, and then work out of this latter folder. When you have finished your changes you can compare your results to the files in the ROTECH6 folder.)*

1. a. Either add the data validation code described in this section to Rotech Mail In and Walk In forms or locate and load the project at the JBWEB website.
   b. Test that the data validation procedures work correctly for each form.
2. a. Make the F11 function key a keyboard shortcut for both the Mail In and Walk In forms, as described in this section.
   b. Test the keyboard shortcut created in Exercise 2a to ensure that it works correctly for both the Mail In and Walk In forms.
3. Using the Help facilities' Index tab, obtain information on the **KeyPreview** method used to create the keyboard shortcut described in this section.

4. Using the Help facility, obtain information on the **KeyCode** constant names and values used for all of the function keys. *Hint:* Enter the words `Keycode Constants` in the Index tabs text box.
5. The key stroke validation used for the quantity text box in both the Mail In and Walk In forms is a bit too restrictive because it does not permit entry of the backspace key to delete an entered digit (a user can still use the arrow keys and the delete key to edit the text box's data). Using the fact that the named constant for the backspace key is `vbKeyBack`, which you can verify by completing Exercise 4, modify the keystroke validation for the txtQuantity text box to permit entry of the backspace key.
6. Event Procedure 6–8 uses the following series of six individual `If` statements to check each name and address text box:

```
If txtFname.Text.Length = 0 Then
  strListEmptyBoxes = strListEmptyBoxes & "First Name " & ControlChars.CrLf
End If
If txtLname.Text.Length = 0 Then
  strListEmptyBoxes = strListEmptyBoxes & "Last Name " & ControlChars.CrLf
End If
If txtAddr.Text.Length = 0 Then
  strListEmptyBoxes = strListEmptyBoxes & "Address " & ControlChars.CrLf
End If
If txtCity.Text.Length = 0 Then
  strListEmptyBoxes = strListEmptyBoxes & "City " & ControlChars.CrLf
End If
If txtState.Text.Length = 0 Then
  strListEmptyBoxes = strListEmptyBoxes & "State " & ControlChars.CrLf
End If
If txtZip.Text.Length = 0 Then
  strListEmptyBoxes = strListEmptyBoxes & "Zip Code "
End If
```

Determine the effect of replacing this set of statements by the following single **If-then-ElseIf** structure:

```
If txtFname.Text.Length = 0 Then
  strListEmptyBoxes = strListEmptyBoxes & "First Name " & ControlChars.CrLf
ElseIf txtLname.Text.Length = 0 Then
  strListEmptyBoxes = strListEmptyBoxes & "Last Name " & ControlChars.CrLf
ElseIf txtAddr.Text.Length = 0 Then
  strListEmptyBoxes = strListEmptyBoxes & "Address " & ControlChars.CrLf
ElseIf txtCity.Text.Length = 0 Then
```

```
  strListEmptyBoxes = strListEmptyBoxes & "City " & Control1Chars.CrLf
ElseIf txtState.Text.Length = 0 Then
  strListEmptyBoxes = strListEmptyBoxes & "State " & Control1Chars.CrLf
ElseIf txtZip.Text.Length = 0 Then
  strListEmptyBoxes = strListEmptyBoxes & "Zip Code "
End If
```

7. Event Procedure 6–9 uses the following **If** statement to check that neither a first nor last name has been entered in the respective text boxes:

   ```
   If txtFname.Text.Length = 0 AND txtLname.Text.Length = 0 Then
     txtFname.Text = "Cash"
   End If
   ```

   Determine the effect of replacing this statement with the following:

   ```
   If (txtFname.Text.Length * txtLname.Text.Length) = 0 Then
     txtFname.Text = "Cash"
   End If
   ```

8. (Case study) For your selected project (see project specifications at the end of Section 1.5), complete all order entry forms by adding appropriate after-the-fact data validation code to the appropriate event procedures.

6.8 Knowing About: Programming Costs

Any project that requires a computer incurs both hardware and software costs. The costs associated with the hardware consist of all costs relating to the physical components used in the system. These components include the computer itself, peripherals, and any other items, such as air conditioning, cabling, and associated equipment, required by the project. The software costs include all costs associated with initial program development and subsequent program maintenance. As illustrated in Figure 6–26, software costs represent the greatest share of most computer projects.

The reason that software costs contribute so heavily to total project costs is that these costs are labor-intensive that is, they are closely related to human productivity, while hardware costs are more directly related to manufacturing technologies. For example, microchips that cost over $500 per chip 15 years ago can now be purchased for under $1 per chip.

It is far easier, however, to dramatically increase manufacturing productivity by a thousand, with the consequent decrease in hardware costs, than it is for people to double either the quantity or the quality of their thought output. So as hardware costs have plummeted, the ratio of software costs to total system costs (hardware plus software) has increased dramatically. As was previously noted in Section 1.2 (see Figure 1–11, repeated as Figure 6–27 for convenience), maintenance of existing programs accounts for the majority of software costs.

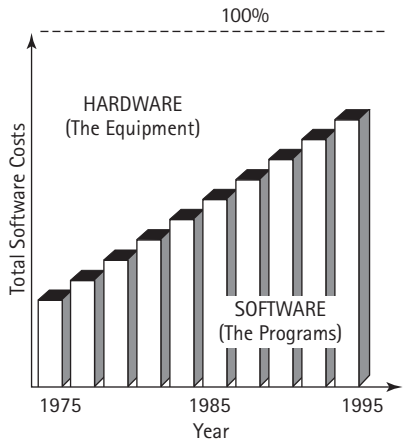


Figure 6–26 *Software is the Major Cost of Most Engineering Projects*

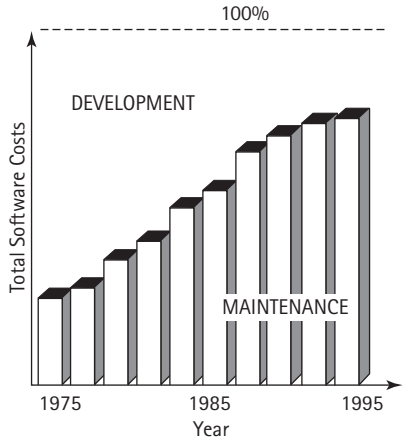


Figure 6–27 *Maintenance is the Predominant Software Cost*

How easily a program can be maintained (debugged, modified, or enhanced) is related to the ease with which the program can be read and understood, which is directly related to the modularity with which the program was constructed. Modular programs are constructed using procedures, each of which performs a clearly defined and specific task. If each procedure is clearly structured internally and the relationship between procedures is clearly specified, each procedure can be tested and modified with a minimum of disturbance or undesirable interaction with the other procedures in the program.

Just as hardware designers frequently locate the cause of a hardware problem by using test methods designed to isolate the offending hardware subsystem, modular software permits the software engineer to similarly isolate program errors to specific software units.

Once a bug has been isolated, or a new feature needs to be added, the required changes can be confined to appropriate procedures without radically affecting other procedures. Only if the procedure in question requires different input data or produces different outputs are its surrounding procedures affected. Even in this case, the changes to the surrounding procedures are clear: they must either be modified to output the data needed by the changed procedure or be changed to accept the new output data. Procedures help the programmer determine where the changes must be made, while the internal structure of the procedure itself determines how easy it will be to make the change.

Although there are no hard and fast rules for well-written procedures, specific guidelines do exist. Programming teams usually set standards that dictate the way source code should be written and formatted. Each procedure should have one entrance point and one exit point, and each control structure in the unit, such as a **Do** loop, should also contain a single entry and exit point. This makes it easy to trace the flow of data when errors are detected. All the Visual Basic selection and repetition statements that alter the normal sequential program flow conform to this single-input-single-output model.

As we have stressed throughout the text, instructions contained within a procedure should use variable names that describe the data and are self-documenting. This means that they tell what is happening without a lot of extra comments. For example, the statement

```
x = (a - b) / (c - d)
```

does not contain intelligent variable names. A more useful set of instructions, assuming that a slope is being calculated, is:

```
slope = (y2 - y1) / (x2 - x1)
```

Here, the statement itself tells what the data represents, what is being calculated, and how the calculation is being performed. Always keep in mind that the goal is to produce programs that make sense to any programmer reading them, at any time. The use of mnemonic data names makes excessive comments unnecessary. The program should contain a sufficient number of comments explaining what a procedure does and any other information that would be helpful to other programmers.

Another sign of a good program is the use of indentation to alert a reader to nested statements and indicate where one statement ends and another begins. Consider the following pseudocode listing the algorithm for determining "What to Wear":

If it is below 60 degrees
If it is snowing
wear your lined raincoat

```
    Else
    wear a topcoat
    If it is below 40 degrees
    wear a sweater also
    If it is below 30 degrees
    wear a jacket also
    ElseIf it is raining
wear an unlined raincoat
```

Because the **If** and **Else** statement matches are not clearly indicated, the instructions are difficult to read and interpret. For example, consider what you would wear if the temperature is 35 degrees and it is raining. Now consider the following version of "What to Wear":

```
If it is below 60 degrees
    If it is snowing
      wear your lined raincoat
    Else
      wear a topcoat
    If it is below 40 degrees
      wear a sweater also
    If it is below 30 degrees
      wear a jacket also
    ElseIf it is raining
      wear an unlined raincoat
```

The second version is indented, making it clear that we are dealing with one main **If-ElseIf** statement. If it is below 60 degrees, the set of instructions indented underneath the first **If** will be executed; otherwise, the **ElseIf** condition will be checked. Although a Visual Basic program essentially ignores indention and will always pair an **ElseIf** with the closest matching **If**, indention is extremely useful in making code understandable to the programmer.

6.9 Common Programming Errors and Problems

Beginning Visual Basic programmers commonly make three errors when using repetition statements:

1. Testing for equality in repetition loops when comparing floating point or double-precision operands. For example, the condition `num = 0.01` should be replaced by a test requiring that the absolute value of `num - 0.01` be less than an acceptable

amount. The reason for this is that all numbers are stored in binary form. Using a finite number of bits, decimal numbers such as .01 have no exact binary equivalent, so that tests requiring equality with such numbers can fail.

2. Failure to have a statement within a **Do** loop that alters the tested condition in a manner that terminates the loop.
3. Modifying a **For** loop's counter variable within the loop.

6.10 Chapter Review

Key Terms

counter loop
Do/Loop Until loop
Do/Loop While loop
Do Until loop
Do While loop
Exit Do
fixed count loop
For/Next loop
infinite loop
nested loop
posttest loop
pretest loop
sentinel values
variable condition loop

Summary

1. A section of repeating code is referred to as a *loop*. The loop is controlled by a repetition statement that tests a condition to determine whether the code will be executed. Each pass through the loop is referred to as a *repetition* or *iteration*. The tested condition must always be explicitly set prior to its first evaluation by the repetition statement. Within the loop there must always be a statement that permits altering the condition so that the loop, once entered, can be exited.
2. There are three basic types of loops:
   a. `Do While`
   b. `For/Next`, and
   c. `Do/Loop Until`

   The `Do While` and `For/Next` loops are *pretest* or *entrance-controlled loops*. In this type of loop, the tested condition is evaluated at the beginning of the loop, requiring that the tested condition be explicitly set prior to loop entry. If the condition is **True**, loop repetitions begin; otherwise the loop is not entered. Iterations continue as long as the condition remains **True**.

   The `Do/Loop Until` loop is a *posttest* or *exit-controlled loop*, where the tested condition is evaluated at the end of the loop. This type of loop is always executed at least once. `Do/Loop Until` loops continue to execute as long as the tested condition is **False** and terminate when the condition becomes **True**.
3. Loops are also classified according to the type of tested condition. In a *fixed-count loop*, the condition is used to keep track of how many repetitions have occurred. In a *variable-condition loop*, the tested condition is based on a variable that can change interactively with each pass through the loop.

4. In Visual Basic, the most commonly used form for a **While** loop is:

```
Do While condition
   statement(s)
Loop
```

The *condition* is tested to determine if the statement or statements within the loop are executed. The condition is evaluated in exactly the same manner as a condition contained in an **If-Else** statement; the difference is how the condition is used. In a **Do While** statement, the statement(s) following the condition is(are) executed repeatedly, as long as the expression is **True**. An example of a **While** loop is:

```
count = 1                ' initialize count
Do While count <= 10
 lstDisplay.Items.Add(count)
  count = count + 1    ' increment count
Loop
```

The first assignment statement sets `count` equal to 1. The `Do While` loop is then entered and the condition is evaluated for the first time. Since the value of count is less than or equal to 10, the condition is **True** and the statements within the loop are executed. The first statement displays the value of `count`. The next statement adds 1 to the value currently stored in `count`, making this value equal to 2. The `Do While` structure now loops back to retest the condition. Because `count` is still less than or equal to 10, the two statements within the loop are again executed. This process continues until the value of `count` reaches 11.

The `Do While` loop always checks a condition at the top of the loop. This requires that any variables in the tested expression must have values assigned before the `Do While` is encountered. Within the `Do While` loop there must be a statement that alters the value of the tested condition.

5. Sentinels are prearranged values used to signal either the start or end of a series of data items. Typically, sentinels are used to create `Do While` loop conditions that terminate the loop when the sentinel value is encountered.
6. The `For/Next` structure is extremely useful in creating loops that must be executed a fixed number of times. The initializing value, final value, and increment used by the loop are all included within the `For` statement. The general syntax of a `For/Next` loop is:

```
For counter = start value To end value Step increment value
   statement(s)
Next counter
```

7. Both `Do While` and `For/Next` loops evaluate a condition at the start of the loop. The `Do/Loop Until` structure is used to create an exit-controlled loop, because it checks its expression at the end of the loop. This ensures that the body of a **Do** loop is executed at least once. The syntax for this loop structure is:

```
Do
   statement(s)
Loop Until condition
```

The important concept to notice in the **Do/Loop Until** structure is that all statements within the loop are executed at least once before the condition is tested, and the loop is repeated until the condition becomes **True** (another way of viewing this is that the loop executes while the condition is **False**). Within the loop, there must be at least one statement that alters the tested expression's value.

Test Yourself—Short Answer

1. A section of repeating code is referred to as a _____.
2. A prearranged value used to signal either the start or end of a series of data items is called a _____.
3. In the **For Next** loop statement, `For X = 1 To 100 Step 5`, what is the purpose of the Step clause?
4. List the four elements required in the construction of repetitive sections of code.
5. Explain the difference between a pretest loop and a posttest loop.

Problems

Note: in all the problems listed below, output should be displayed in a **List Box** control.

1. Determine the output of the following program segment, assuming that all variables have been declared as `single`.

```
sngNum = 1734527
sngComputedValue = 0
sngCounter = 0

Do While sngComputedValue < 20
    sngSmallNum = sngNum / 10
    sngLargeNum = Int(sngSmallNum)
    sngDigit = sngNum - sngLargeNum * 10
    sngComputedValue = sngComputedValue + sngDigit
    sngNum = sngLargeNum

    lstOutput.Items.Add("sngCounter = " & sngCounter)
    lstOutput.Items.Add("sngComputedValue  = " & _
      sngComputedValue)
    sngCounter = sngCounter + 1
Loop
```

2. Write a loop that displays the numbers 1 to 10 on two lines.
3. Write a loop that displays the even numbers from 2 to 20 on two lines.
4. Write a loop that displays the first 50 numbers in the following sequence: 1, 2, 4, 7, 11, 16, 33, ...
5. Write a loop that displays all integers less than 1000 whose square roots are integers (i.e., 1, 4, 16, 25, . . .).

6. Write a loop that displays the following table:

```
1 2 3 4
2 3 4 5
3 4 5 6
4 5 6 7
5 6 7 8
6 7 8 9
7 8 9 10
```

7. Write a loop that displays the following table:

```
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
5 6 7 8 9
6 7 8 9 10
```

8. Write a loop that displays the following table:

```
1
2 4
3 6 9
4 8 12 16
5 10 15 20 25
```

9. Write a loop that displays the following:

```
     *
    **
   ***
  ****
 *****
******
```

10. Write a loop that displays the following:

```
******
 *****
  ****
   ***
    **
     *
```

Programming Projects

1. **a.** Create an application that determines the most efficient change to be given from a one dollar bill. The change should be given in terms of quarters, dimes, nickels, and pennies. For example, a purchase of 44 cents would require change of two quarters, a nickel, and a penny (two quarters and six pennies would not be correct). Your program should:
   i. Accept the amount of purchase in terms of the number of cents (integers, no decimals).
   ii. Calculate the change required.
   iii. Determine the number of quarters, dimes, nickels, and pennies necessary for the *most efficient* change.
   iv. Add a Clear button to clear all the quarters, dimes, nickels, and pennies and set the focus to the input text box.

   A suggested form is shown in Figure 6–28.

   b. Change your code so that if the amount of purchase is over one dollar or less than 0, an error message is given and the input text box receives the focus.

2. Credit scoring is a common practice for businesses that offer credit to their customers. Some of the factors that are generally used in credit scoring include whether the customer owns a home, has a telephone in their own name, has a savings or checking account, the years employed at their present job, and so on. In this project you are to construct a program that determines a credit score. The credit scoring algorithm used by the company requesting the program is as follows:
   i. Add 5 points for each of the following: phone (in customer's name), owns home, has a savings or checking account.
   ii. If the customer has been at the same job for less than two years, add no points; if they have been at the same job for two years but less than four years, add 3 points, and if they have been at the same job for four or

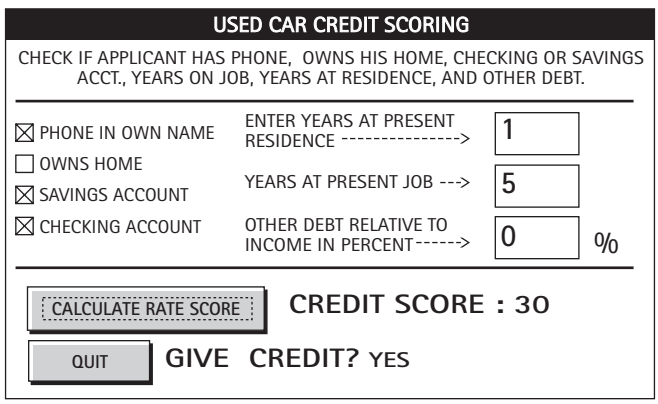


Figure 6–28 *Form for Project 1a*

more years, add 5 points. For example, for three years at the same job, add 3 points.

iii. If the customer has been at the same residence for less than two years, add no points; if they have been at the same residence for two years but less than four years, add 3 points; and if they have been at the same residence for four or more years, add 5 points. For example, for three years at the same residence, add 3 points.

iv. If the customer has other debt, the percent of this debt relative to total income is evaluated as follows:

no debt, add 10 points

up to 5% of income, add 5 points

5% to <25% of income, no points

25% or more of income, subtract 10 points

Your program should permit a user to enter all the above information. It should then display the credit score (number of points) based on the entered data. A form for this application is shown in Figure 6–29.

3. Wind chill is calculated by subtracting wind speed times 1.5 from the current temperature. For example, if the current temperature is 40 degrees and the wind speed is 10 miles per hours, then Wind Chill = 40 − 1.5 * 10 = 25 degrees. For this project, the input for your program should include current temperature and a minimum value for wind speed (value should be at least 0). Your program should display a table of 10 values for wind speed and corresponding wind chill, starting with the input value for wind speed. For each pair of values, increment wind speed by 2. Use a **Do While-Loop** structure to handle the repetition. Output the pairs of values in a list box in a second form. Express wind chill as an integer value. Figure 6–30 illustrates the input form you should construct for this project.

For the form shown, the Display button should display a second form with the table values. The Clear button should clear all input entries on the input

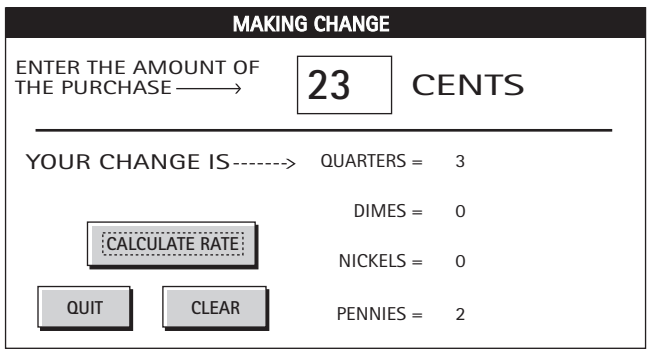


Figure 6–29 *Form for Project 2*

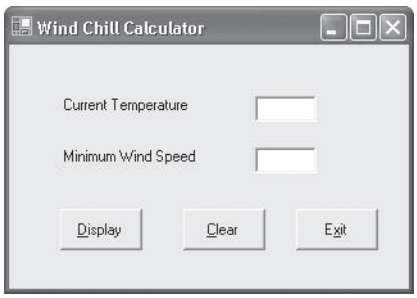


Figure 6–30 *Form for Project 3*

form, and the Exit button should end the program. The second form should have a Return that hides the form and redisplays the input form. For this project, use the following properties table.

Property Table for Project 3

Object	Property	Setting
Form	Name	frmWindSpeed
	Text	Wind Chill Calculator
Label	Name	lblCurrentTemp
	Text	Current Temperature
TextBox	Name	txtCurrentTemp
	Text	(blank)
Label	Name	lblMinWindSpeed
	Text	Minimum Wind Speed
TextBox	Name	txtMinWindSpeed
	Text	(blank)
Button	Name	btnDisplay
	Text	&Display
Button	Name	btnClear
	Text	&Clear
Button	Name	btnExit
	Text	E&xit
Form	Name	frmWindChill
	Text	Wind Chill Factor
Label	Name	lblCurrentTemp
	Text	For a temperature of:
ListBox	Name	lstCurrentTemp
Button	Name	btnReturn
	Text	&Return

4. Modify your solution to Project 3 to have your program display a two-way table of wind chill factors for pairs of wind speed and temperature. Input for the program should include a minimum value for temperature and a minimum value for wind speed. Output should be displayed on a separate form. All numeric values should be integers. Column headings should represent incremented values of temperature and row headings should represent incremented values of wind speed. Increase wind speed by 2 and temperature by 5.